2 Mathematical and Cryptographic Foundation

> There are two types of encryption: one that will prevent your sister from reading your diary and one that will prevent your government.
>
> Bruce Schneier

2.1 Preliminaries and Notation

This section introduces cryptographic primitives and their foundation. The presentation follows common notations and definitions. To keep these preliminaries short and readable, we omit some special cases and slightly simplify certain definitions when the omitted exceptions are of no interest here. We assume the reader is familiar with general probability theory and basic group theory, even though we give a short introduction into the latter in Sect. 2.1.2.

2.1.1 Functions and Algorithms

Deterministic and Probabilistic Algorithms

We start with the notion of deterministic and probabilistic algorithms. While *deterministic algorithms* behave predictably, and thus, always compute the same output given a particular input using the same procedure, *probabilistic algorithms* use a source of randomness. Therefore either the running time or the output of probabilistic algorithms will be affected by randomness. This behaviour can be useful to avoid "malicious input", which would result in bad running times of the algorithm (see Worst-Case and Average-Case Complexity below) or to create algorithms producing output influenced by randomness. Allowing randomness to influence the algorithm's output can be useful to create algorithms solving problems with some (hopefully small) error rate or for algorithms giving different outputs

S. Pape, *Authentication in Insecure Environments,* DOI 10.1007/978-3-658-07116-5_2,

when run on the same input, which is useful in cryptographic encryption schemes, for example.

Definition 2.1. (Deterministic Functions and Algorithms) Let $f : X \to Y$ be a *deterministic function* or *algorithm* with input x from domain X and output y from codomain Y, then we denote f results with input x in output y with:

$$y = f(x) \quad \text{or} \quad y := f(x).$$

Definition 2.2. (Probabilistic Functions and Algorithms) Let $f : X \to Y$ be a *probabilistic function* or *algorithm* with input x from domain X and output y from codomain Y, then we denote f generates output y with input x with:

$$y \in_R Y_{f(x)} \quad \text{or} \quad y \leftarrow f(x).$$

Definition 2.3. (Equality of Probabilistic Functions and Algorithms) Let $f, g : X \to Y$ be probabilistic functions or algorithms with input x, x' from domain X and output y, y' from codomain Y. By the equality $f(x) = g(x')$ we require that both functions or algorithms return the same values with the same probability $\Pr$:

$$f(x) = g(x') \quad \Leftrightarrow \quad \{y | y \leftarrow f(x)\} = \{y | y \leftarrow g(x')\} \wedge \Pr[y \leftarrow f(x)] = \Pr[y \leftarrow g(x')]$$

To compare algorithms with respect to running time or memory usage, we introduce the Big O notation which describes the behaviour of a function when its argument tends to infinity. A common use case is to express an algorithm's running time as a function of its input length using the Big O notation. The Big O notation makes a statement about a function's growth rate and allows to simplify these functions by suppressing multiplicative constants and low order terms. This allows an easy comparison of functions' growth, and thus can be used to conveniently compare algorithms' running times for "large input lengths" but sacrifices predictions for concrete instances.

Definition 2.4. (Big O notation) Let $f, g : X \to \mathbb{R}$ be functions defined on the same totally ordered set X with codomain $\mathbb{R}$ and let be $c \in \mathbb{R}$ and $x_0, x \in X$. Then we define the *Big O notation* or *Landau notation*:

$$f \in \mathcal{O}(g) \stackrel{\text{def}}{=} \exists\, c > 0\ \exists\, x_0\ \forall\, x > x_0 : |f(x)| \leqslant c \cdot |g(x)|$$

Following common conventions we also mean $f \in \mathcal{O}(g)$ when writing $f = \mathcal{O}(g)$.

In other words if $f \in \mathcal{O}(g)$ then, beginning from a certain value x_0, $f(x)$ is not significantly larger[1] than $g(x)$.

[1] In this case, not significantly larger means maximum a constant c times larger.

Remark 2.5. **(Security Parameter)** In cryptography it is common practise to view the running time as a function of n, often called the *security parameter*. But as previously mentioned the standard convention in complexity theory is to measure the running time of an algorithm as a function of the length of its input. To be consistent with the latter notion, we provide the algorithms with the security parameter in unary as 1^n (for example a string of n 1's) when necessary.

Now that we have a comfortable way of comparing algorithms' resources, we are able to elaborate which algorithms are feasible (for large instances).

Efficient Algorithms and Negligible Probability

Kerckhoff is mostly known for his principle saying that the security of the cryptosystem should not rely on its secrecy (see Sect. 2.2). It is less known that he formulated six principles in total, one of them saying "*The system must be practically, if not mathematically, indecipherable*" [Ker83]. That means, when considering the security of a cryptosystem, an attacker should not be able to break the cryptosystem in "reasonable time" with "reasonable success probability". Thus, before proceeding with the definition of cryptosystems, we need to give appropriate definitions of "reasonable". At first we equate "doing something in reasonable time" with the definition of *efficient computation*.

Definition 2.6. (Efficient Computation) We define *efficiently computable algorithm* as algorithm with access to a fair coin, solving the underlying problem in polynomial time with an error rate of at most $\frac{1}{3}$.

Note that the choice of $\frac{1}{3}$ is arbitrary, as long as it is a constant, independent of the input, and below $\frac{1}{2}$. The idea is to repeat the algorithm several times, and thus, increase the accuracy by taking a majority vote. We refer to Gill [Gil77] for an exact definition of "Bounded-Error Probabilistic Polynomial-Time"-algorithms and to Aaronson [AKG13] for an overview about its relations to other complexity classes.

Having a definition of efficient computation now, we continue by defining "reasonable success probability" as a *not negligible* success probability.

Definition 2.7. (Negligible function) A function $f : \mathbb{N} \to \mathbb{R}$ is *negligible* iff

$$\forall c \in \mathbb{N}\ \exists k_c \in \mathbb{N}\ \forall k \geqslant k_c\ :\ |f(k)| \leqslant \frac{1}{k^c}$$

In other words, a function that increases slower than any inverse polynomial is called *negligible*.

Note that k_c is a natural number, since in complexity-based cryptography, the input of the negligible function is usually the cryptographic key length or a corresponding security parameter. For example, an encryption scheme is defined as secure (see indistinguishability in Sect. 2.2.1), if the probability of an attacker's success is negligible in terms of cryptographic key length. Therefore, we continue with the definition of negligible success probability.

Definition 2.8. (Negligible success probability) The success probability of an algorithm is *negligible* iff the success probability, as a function of its input length, is bounded by a negligible function.

This complements our definition of efficient computation, since repeating an algorithm with negligible success probability polynomially (in the input length) many times, results in a new algorithm with also negligible success probability.

One-Way and Trapdoor Functions

Informally speaking, one-way functions have the property that they are easy to compute, but hard to invert given the function's result of a random input.

Definition 2.9. (One-Way Function) Let $f : \{0,1\}^* \to \{0,1\}^*$ be an efficient computable function. f is a *one-way function* if for every probabilistic polynomial-time algorithm $\mathcal{A}$ the success probability Pr of finding a x' for a given $y := f(x)$, so that $f(x') = y$, is negligible:

$$\forall c \in \mathbb{N}\ \exists k_c \in \mathbb{N}\ \forall k \geqslant k_c\ :\ Pr[f(\mathcal{A}(f(x))) = f(x)] < \frac{1}{k^c}$$

Note that any one-way function can be inverted if the adversary is given enough time and thus able to simply try all values $x \in \{0,1\}^*$ until a value x' is found, such that $f(x') = y = f(x)$. Using this algorithm an adversary always succeeds, but would require exponential running time. Hence, the assumption that a function is one-way is strongly related to computational complexity and computational hardness assumptions [KL08]. Consequently, the existence of one-way functions is still an open question. A proof of their existence would also imply that the complexity classes P and NP are distinct, which is one of the most important unsolved problems in theoretical computer science [Gol01]. An up-to-date survey on the current status of P versus NP has been published recently by Fortnow [For09]. However, it is also unknown if P unequals NP – which is widely believed – is a sufficient condition for the existence of one-way functions.

Candidates for one-way functions (cf. Sect. 2.3) are:

Multiplication and factoring Let p and q be two prime numbers in binary notation with length k. Then the multiplication of p and q can be computed in time $\mathcal{O}(k^2)$, while reverting it requires to find the factors of a given integer N with length n and the best factoring algorithm known, the general number field sieve [Cop93, Pom96], runs in time $2^{\mathcal{O}(n^{1/3}(\log n)^{2/3})}$ on average [KL08, p. 298].

Modular squaring and square roots Let N be the product of two prime numbers p and q. The modular squaring of $N > k \in \mathbb{Z}$ means to compute the remainder of k^2 divided by N, which can be done in time $\mathcal{O}(k^2)$ [HyOY96]. Obviously inverting it requires computing a square root modulo N, which has been shown to be computationally equivalent to factoring N [CGG86].

Discrete exponential function and logarithm Let p be a prime number in binary notation with length n and $k \in \mathbb{Z}$ between 0 and $p-1$. The remainder of 2^k divided by p (the discrete exponential function) can be computed in time $\mathcal{O}(n^3)$, while inverting this function requires computing the discrete logarithm modulo p. Currently, the best known algorithm, the general number field sieve – which shares many of the underlying ideas with its factoring counterpart – runs also in time $2^{\mathcal{O}(n^{1/3}(\log n)^{2/3})}$ on average [AH99, KL08].

Cryptographically secure hash functions Even though earlier versions of hash functions [MvOV97, p. 33] (see also Def. 2.76) like MD5 and SHA-1 have been successfully attacked by Wang et al. [WFLY04, WY05, WYY05], recent attacks on the latest family of hash functions – SHA-2 – by Sanadhya et al. work only on a reduced version [SS08a, SS08b]. Thus, as of today, the SHA-2 family offers practical and efficient computable one-way functions.

A special form of one-way functions with additional properties are one-way permutations.

Definition 2.10. (One-Way Permutation) Let $f : \{0,1\}^* \to \{0,1\}^*$ be a one-way function. f is a *one-way permutation* if f is injective and surjective.

Since one-way permutations are bijective, it follows, that any value $f(x)$ uniquely determines its preimage $f^{-1}(x)$: $\forall x, x' \in \{0,1\}^* : f(x) = f(x') \Leftrightarrow x = x'$. Note the difference when talking about inversion referring to one-way functions and one-way permutations, respectively. While the inversion is uniquely determined for one-way permutations, it is sufficient for the inversion of one-way functions to find any appropriate preimage value.

Strongly related to one-way permutations are trapdoor permutations. One can imagine trapdoor permutation as one-way permutation, but with the aid of special information (the trapdoor), it becomes easy to invert the permutation.

Definition 2.11. (Trapdoor Permutation) Let $f : \{0,1\}^* \to \{0,1\}^*$ be an efficiently computable function. f is a *trapdoor permutation* if for every probabilistic polynomial-time algorithm $\mathcal{A}$ the success probability of finding x for a given $y := f(x)$ is negligible and there exists an efficient algorithm $\mathcal{I}$, which outputs x on input $f(x)$, using trapdoor td:

$$\forall c \in \mathbb{N}\ \exists k_c \in \mathbb{N}\ \forall k \geqslant k_c\ :\ Pr[\mathcal{A}(f(x) = x] < \frac{1}{k^c}$$

and

$$\exists\, \mathcal{I} :\ \mathcal{I}(f(x), td)\ =\ x$$

Trapdoor functions are an important component for public-key encryption techniques (see Sect. 2.41) and the best known trapdoor function candidates are based on the RSA [RSA78] and Rabin [Rab79] families of functions, which both rely on the problem of prime factorisation.

The term of "families of functions" is quite intuitive, but there is a formal definition [KL08] and we show their application on one-way functions and one-way and trapdoor permutations. Most known candidates of one-way functions (or one-way permutations or trapdoor permutations, respectively) do not naturally fit in the concept of families of functions. Instead, there is an algorithm generating some parameters I which define some function f_I and the requirement then is that except with negligible probability f_I is a one-way function (or one-way permutation or trapdoor permutation, respectively). Because each value of I defines a different function, we refer to families of one-way functions, one-way permutations and trapdoor permutations.

Definition 2.12. (Family of Functions) A tuple $(Gen, Samp, f)$ of probabilistic polynomial-time algorithms is a *family of functions* if the following holds:

- Each run of the *parameter-generation algorithm* Gen outputs a parameter I with $|I| > n$ on input 1^n. Each value of I defines the domain $\mathcal{D}_I$ and the range $\mathcal{R}_I$ of a function f_I defined below.
- The *sampling algorithm* $Samp$ outputs a uniformly distributed element of $\mathcal{D}_I$ on input I (except possibly with probability negligible in $|I|$).
- The deterministic *evaluation algorithm* f outputs an element $y \in \mathcal{R}_I$ on input I and $x \in \mathcal{D}_I$. We write this as $y := f_I(x)$.

Definition 2.13. (Family of Permutations) A tuple $\Pi = (Gen, Samp, f)$ of probabilistic polynomial-time algorithms is a *family of permutations* if Π is a family of functions and for each value of I output by $Gen(1^n)$, it holds that $\mathcal{D}_I = \mathcal{R}_I$ and the function $f_I : \mathcal{D}_I \to \mathcal{D}_I$ is bijective.

Definition 2.14. (Families of One-Way Functions and Permutations) A family of functions or permutations is *one-way* if for all I, output by $\mathsf{Gen}(1^n)$, the deterministic evaluation function respective permutation f_I is one-way (except possibly with probability negligible in $|I|$).

Definition 2.15. (Families of Trapdoor Permutations) A tuple $(\mathsf{Gen}, \mathsf{Samp}, f, \mathcal{I})$ of probabilistic polynomial-time algorithms is a *family of trapdoor functions* if the tuple $(\mathsf{Gen}, \mathsf{Samp}, f)$ is a family of one-way permutations and for each value of I output by $\mathsf{Gen}(1^n)$, there exists an efficient algorithm $\mathcal{I}_I$, which outputs x on input $f_I(x)$ by using trapdoor td_I.

As already indicated above, most candidates of one-way functions and one-way or trapdoor permutations belong to families of functions and permutations. Thus, even though each value of I defines a different function f_I, all members of a specific one-way function or one-way or trapdoor permutation rely on the same computational hardness assumption.

Pseudorandom Functions

Pseudorandomness is a computational relaxation of true randomness. Computational relaxation means, a polynomial bound observer is unable to distinguish a pseudorandom distribution from the uniform distribution. On an intuitive level, pseudorandomness helps in the construction of encryption schemes, since an adversary faced with ciphertexts which seem random to her (he cannot distinguish their distribution from the uniform distribution), should not be able to learn any information from them about the underlying plaintexts. Thus, many secure encryption schemes fundamentally rely on the use of pseudorandom functions as cryptographic primitives.

It is possible to construct pseudorandom functions based on one-way functions with the intermediate step of a pseudorandom generator. Therefore this section starts with the definition of pseudorandom generators before defining pseudorandom functions.

Definition 2.16. (Pseudorandom Generator) Let $\ell(\cdot)$ be a polynomial and let G be an efficient, deterministic algorithm: $G : \{0,1\}^n \to \{0,1\}^{\ell(n)}$. G is a cryptographically secure *pseudorandom generator (*PRG*)*, if G outputs a long pseudorandom string $s' \in \{0,1\}^{\ell(n)}$ on the input of a short truly random seed $s \in \{0,1\}^n$, for every n it holds that $\ell(n) > n$ and the pseudorandom string s' is (except with negligible probability in n) computationally indistinguishable from a truly random string. The function $\ell(\cdot)$ is denoted as *expansion factor* of G.

Note that the output of a pseudorandom generator is not even close to real randomness, if regarded from a higher level of abstraction. This can be seen easily since the pseudorandom generator with input $s \in \{0,1\}^n$ has at most 2^n different outputs due to its deterministic nature. Since the length of its output $s' \in \{0,1\}^{\ell(n)}$ is $\ell(n)$ and $\ell(n) > n$, there is a gap of $2^{\ell(n)} - 2^n$ strings which will never occur as output of the regarded pseudorandom generator, but would be output of a uniform distribution over $\{0,1\}^{\ell(n)}$. That means an adversary with unlimited computational power could on an output s' simply brute-force all seeds and see if there is a seed $s \in \{0,1\}^n$ such that $G(s) = s'$. Since only for the fraction $\frac{2^n}{2^{\ell(n)}} = 2^{n-\ell(n)}$ of G's outputs there will exist a seed s, the adversary is able to distinguish between a pseudorandom and an uniform distribution. However, this procedure cannot be done by an efficient algorithm, which runs polynomial in n. From the above argument also follows that the seed s for a pseudorandom generator must be kept secret, chosen uniformly at random, and must be long enough that it is unfeasible for the distinguisher to brute-force all possible seeds.

Unfortunately it is not known, if pseudorandom generators exist, even if it is strongly believed. The belief is based on the fact that Levin et al. proved that a a cryptographically secure pseudorandom bit generator can be constructed from a one-way function [Lev87, ILL89] and one-way functions are believed to exist as stated earlier in the section.

Definition 2.17. (Pseudorandom Function) Let F be an efficient, deterministic function: $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$. F outputs a string $s' \in \{0,1\}^*$ on the input of a key $k \in \{0,1\}^*$ and a string $s \in \{0,1\}^*$. In general the key k will be chosen uniformly at random and then fixed. We denote this with $F_k : \{0,1\}^* \to \{0,1\}^*$ with $F_k(x) = F(k,x)$. F_k is called a cryptographically secure *pseudorandom function (*PRF*)*, if F_k is (except with negligible probability in n) computationally indistinguishable from an uniformly at random chosen function of the set of functions mapping strings of F_k's domain to strings of F_k's codomain.

Since F_k is a set of functions, F_k may also be denoted as pseudorandom function family if k is not fixed. In analogy to the reasoning in the above paragraph, the distribution over pseudorandom functions has a smaller range than the distribution over all functions with the same domain and range. For a more detailed description we refer to Katz and Lindel [KL08, p. 86-87]. Likewise, the consequence is that the key k for a pseudorandom function must be kept secret, chosen uniformly at random, and must be long enough that it is unfeasible for the distinguisher to brute-force all possible keys.

Goldreich, Goldwasser and Micali show how to construct pseudorandom function families from pseudorandom generators [GGM86]. In theory it is known that pseudorandom functions exist iff pseudorandom generators exist [KL08]. In prac-

tise, block ciphers like AES [DR00] are widely believed to work as pseudorandom functions.

For the sake of clarity, the difference between pseudorandom generators and pseudorandom functions is that a pseudorandom generator's output appears random if the input was chosen uniformly at random, while the output of a pseudorandom function appears random, regardless of its input, as long as the function was chosen randomly from the pseudorandom function family.

If a pseudorandom function is a length-preserving bijection it is also a pseudorandom permutation:

Definition 2.18. (Pseudorandom Permutation) Let P be an efficient, deterministic permutation: $P : \{0,1\}^* \times \{0,1\}^n \to \{0,1\}^n$. P outputs a string $s' \in \{0,1\}^n$ on the input of a key $k \in \{0,1\}^*$ and a string $s \in \{0,1\}^n$. In general the key k will be chosen uniformly at random and then fixed. We denote this with $P_k : \{0,1\}^n \to \{0,1\}^n$ with $P_k(x) = P(k,x)$. P_k is called a cryptographically secure *pseudorandom permutation (*PRP*)*, if P_k is (except with negligible probability in n) computationally indistinguishable from an uniformly at random chosen permutation of the set of permutations on n-bit strings.

It may be necessary for one or more parties participating in an encryption scheme to compute not only the pseudorandom permutation P_k, but also its inverse P_k^{-1}, which may introduce security issues not covered by cryptographically secure pseudorandom permutations. If P_k is indistinguishable from a random permutation, although the adversary is granted oracle access to the inverse of the permutation, we call P_k a strong pseudorandom permutation.

Definition 2.19. (Strong Pseudorandom Permutation) Let PRP_k be an pseudorandom permutation, PRP_k is called a *strong pseudorandom permutation* if PRP_k is (except with negligible probability) computationally indistinguishable from an uniformly at random chosen permutation of the set of permutations, even if the distinguisher is granted oracle access to the inverse of the permutation.

Worst-Case and Average-Case Complexity

Studies on *computational complexity theory* began in the fifties and sixties with publications by Trakhtenbrot [Tra56, Tra67], Rabin [Rab59, Rab60], and Hartmanis and Stearns [HS65]. The aim of computational complexity theory is to analyse the difficulty of computational problems and to classify problems concerning the required resources (time, memory, ...) needed to solve all instances of a certain problem. Naturally, if problems are divided into classes of problems which can be solved with a certain amount of resources, the most significant instances of the problem are the worst-cases referring to a specific algorithm. An efficient algorithm

would be required to solve *all instances* of a certain problem efficiently, even if there is only one extreme case resulting in a poor performance of the algorithm. That means in many cases worst-case analysis is too pessimistic, because there may exist efficient algorithms, which perform quite good on random (average) instances.

Thus, for example, in mathematical optimisation theory, the simplex algorithm, created by DANTZIG [Dan63], is an algorithm to numerically solve linear programming tasks. Although its worst-case running time is exponential [KM72, Zad80] and even so KHACHIYAN [Kha79] presented an algorithm with polynomial running time, the simplex algorithm is widely used because it performs well in practise (on average). Besides the previously mentioned extreme cases, it is important to notice that comparing algorithms asymptotically does not necessarily allow conclusions regarding relevant instances in practise. The reason is that exponential functions with "small coefficients" in the exponent may increase slower up to a certain value than polynomials of a "high degree" with "large coefficients".

In regard to complexity-based cryptography it is necessary to point out the difference between *worst-case complexity* and *average-case complexity* related to the security of a cryptosystem. In public-key encryption schemes (see Def. 2.41) the problem of finding the secret key sk from the public key pk is based on the hardness (cf. Sect. 2.3) of a certain problem. It is obvious, that computing sk from pk should be hard on average and thus the underlying problem should be hard on average. Otherwise an adversary confronted with a problem that is only hard in the worst-case may succeed with reasonable probability.

The emerging gap between average-case and worst-case analysis regarding the security of a cryptosystem may be further amplified if the algorithm's runtime is given in Big O notation. Since Big O notation describes the limiting behaviour of an algorithm on the one hand algorithms with exponential runtime may nevertheless be feasible regarding relevant instances, for example if the exponent's coefficient is very small, while on the other hand algorithms with polynomial runtime may likewise not be feasible regarding relevant instances, for example if the degree and the leading coefficient of the polynomial are very large.

2.1.2 Basic Group Theory

Divisibility and Modular Arithmetic

Definition 2.20. (Division) Let $a, b, c \in \mathbb{Z}$ be integers. We say that a *divides* b if there exists an integer c such that $ac = b$ and denote this by $a|b$. If a does not divide b we write $a \nmid b$. If a is positive and divides b, we call a a *divisor of* b. If furthermore $a \notin \{1, b\}$ holds, we call a a *factor* or *non-trivial divisor of* b.

http://www.springer.com/978-3-658-07115-8

Authentication in Insecure Environments
Using Visual Cryptography and Non-Transferable
Credentials in Practise

Pape, S.

2014, XVI, 362 p. 46 illus., 6 illus. in color., Softcover
ISBN: 978-3-658-07115-8