Chapter 4

Computational Problems and Reductions

4.1 Preliminaries

Computational problems and reductions between problems have been studied and formalized in complexity theory. The types of problems arising in cryptography are more general, in several ways, than those usually considered in complexity theory.

4.1.1 Decision Problems

In classical complexity theory (where classes such as P and NP are defined), an important type of computational problem are *decision problems*. A decision problem is characterized by an instance set $\mathcal{X}$ and a predicate $P\colon \mathcal{X} \to \{0,1\}$. The problem consists of deciding, for a given $x \in \mathcal{X}$, whether $P(x) = 1$.

A specific type of decision problem is characterized by a predicate $Q : \mathcal{X} \times \mathcal{W} \to \{0,1\}$, where $\mathcal{X}$ is an instance set and $\mathcal{W}$ is a witness set. The problem consists of deciding, for a given $x \in \mathcal{X}$, whether there *exists* a witness $w \in \mathcal{W}$ such that $Q(x,w) = 1$, i.e., $P(x) := \exists w\ Q(x,w)$.

Example 4.1. Decide whether a given graph with n vertices contains a Hamiltonian cycle. The instance set is (some subset of) the set of Boolean $n \times n$ matrices (the adjacency matrix of a graph), the witness set is the set of (somehow encoded) paths of length n, and the predicate Q consisting of verifying that a path is a cycle in the graph. This problem is known to be NP-complete.[1]

[1] see M.R. Garey and D.S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness*, New York: W.H. Freeman and Co., 1979.

Example 4.2. Decide whether a given number is a prime number. This problem has been shown to be in P.[2]

4.1.2 Search Problems

The search problem corresponding to a decision problem (of the above type) is the problem of *finding* a witness w (which can be called a solution) for a given x.

Definition 4.1. A *search problem* is a 4-tuple $(\mathcal{X}, \mathcal{W}, Q, \mathsf{P}_X)$ consisting of an *instance set* $\mathcal{X}$, a *solution set* (or *witness set*) $\mathcal{W}$, a predicate $Q : \mathcal{X} \times \mathcal{W} \to \{0,1\}$, as well as a probability distribution P_X over the instance set.[3]

The search problem $(\mathcal{X}, \mathcal{W}, Q, \mathsf{P}_X)$ consists of finding, for a given instance $x \in \mathcal{X}$ drawn according to P_X, a $w \in \mathcal{W}$ such that $Q(x,w) = 1$.[4] Typically one considers cases where every instance x has at least one solution w, but other cases can also be considered by allowing a special output "no solution".

Example 4.3. Finding a Hamiltonian cycle in a graph with n vertices (see Example 4.1). A random graph with n nodes containing a Hamiltonian cycle can, for example, be generated by starting from an n-cycle and then adding random edges according to some distribution.

Exercise 4.1. Suppose that for a search problem there exists a probabilistic algorithm with average success probability 1%. Can one amplify the success probability by repeating the algorithm several times?

Example 4.4. The discrete logarithm (DL) problem (see Definition 3.8) for a group $G = \langle g \rangle$ is a search problem where the instance set is G, the solution set is $\mathbb{Z}_q$, and the predicate Q is defined as

$$Q(x,w) = 1 \iff g^w = x,$$

and the instance distribution P_X is the uniform distribution.

Example 4.5. *Function inversion.* The problem of inverting a function $f : \{0,1\}^n \to \{0,1\}^m$ will be denoted as $\mathcal{I}^f$ ($\mathcal{I}$ stands for inversion.). The instance set is $\mathcal{X} = \{0,1\}^m$, the solution set is $\mathcal{W} = \{0,1\}^n$, and the predicate Q is defined as

$$Q(x,w) = 1 \iff f(w) = x.$$

The distribution P_X is usually assumed to be the distribution implied by the uniform distribution on the input of f, i.e., $X = f(U)$, where U is a uniform n-bit string. Note that the DL problem is also a function inversion problem. A function f for which the function inversion problem is computationally hard is called a *one-way function*.

[2] see M. Agrawal, N. Kayal, and N. Saxena, PRIMES is in P, *Annals of Mathematics*, 160 (2): 781–793, 2004.

[3] Often, but not always, $\mathcal{X} \subseteq \{0,1\}^*$, $\mathcal{W} \subseteq \{0,1\}^*$, and P_X is the uniform distribution.

[4] One can distinguish two types of search problems, depending on whether or not Q is efficiently computable. If Q is efficiently computable, then problems with a small solution set $\mathcal{W}$ are easy to solve by an exhaustive search.

4.1.3 Beyond Worst-case Analyses

In traditional complexity theory, one usually considers only algorithms that work for *all* instances, i.e., in the worst case. Such problems do not specify an instance distribution, i.e., they can be considered to be triples $(\mathcal{X}, \mathcal{W}, Q)$. In cryptography, the worst-case hardness of a problem is useless. Even if (the computational problem of) breaking a cryptosystem is hard in the worst case (for example NP-complete), it is still possible that the problem is easy with substantial probability or even with very high probability. What is required in cryptography are statements about *almost-everywhere* hardness. This forces us to reconsider the notion of success of an algorithm for solving a certain computational problem and the notion of a reduction.

4.1.4 Problems Relevant in Cryptography

We briefly repeat what was already discussed in Section 2.1: The three types of computational problems relevant in cryptography.

In cryptography, an important class of computational problems are so-called *games*. In a game (e.g. the MAC-forgery game; see Definition 3.7), the instance is not only a value, as in a search problem, but an instance is an interactive process (a discrete system, see the next chapter) with which an algorithm (trying to win the game) can interact in several steps. The game is characterized by a special *winning condition*. The algorithm succeeds if it provokes this winning condition to become true. Search problems are a special type of game which are non-interactive.

Another important class of computational problems in cryptography are so-called *distinction problems*. The task of an algorithm is to determine which of two options (e.g. values, or discrete systems) it is given access to. An example is the so-called Decisional Diffie-Hellman (DDH) problem (see Definition 3.10). More generally, many security definitions can be cast in terms of the distinction problem between a (so-called) real-world system and a (so-called) ideal-world system.

Closely related to distinction problem are *bit-guessing problem* where the task is, when given access to a system, to guess a bit hidden within the system. The two types of problems can be shown to be in a certain sense equivalent.

4.1.5 Outline of the Section

In Sections 4.2 we discuss a first example of an interesting reductions and discuss the limitations of the statement. In Sections 4.3 we discuss a more involved and more interesting example in detail, allowing us to discover some important reduction principles. In Sections 4.4 we discuss computational problems and solvers at an abstract level, and in Sections 4.5. In Sections 4.6 we revisit the basic types of computational problems in cryptography at an abstract level.

4.2 Example 1: Security of the RSA LSB

Recall the reduction concept mentioned in Section 2.2.2. We introduce a first example of a non-trivial cryptographically interesting reduction. We describe it in a conventional language (using terms like efficient algorithm); our more abstract and more general terminology and concepts will be introduced later.

The security of the RSA trapdoor one-way permutation relies on the computational hardness of computing e-th roots modulo n. However, even if this problem is hard, it is conceivable that finding certain parts of the e-th root (i.e., of the message in the naïve RSA system), for example the least significant bit (LSB) is nevertheless easy. In this section we prove the surprising result that finding the least significant bit of an RSA-plaintext from a given ciphertext is computationally equivalent to finding the entire plaintext.

Theorem 4.1. *If there exists an efficient algorithm A which for a given k-bit RSA-modulus n, exponent e, and ciphertext c, computes the LSB of the corresponding plaintext m (where $c = m^e$), then there exists an efficient algorithm which for a given c computes the entire corresponding m (by calling algorithm A k times as a subroutine).*

Proof. On input n, e and $c = m^e$ modulo n, algorithm A computes the LSB of m, i.e, $A(c) = \text{LSB}(m)$. Such an algorithm A can be exploited as follows.

By the multiplicative property of the RSA function we (i.e., the reduction algorithm) can easily compute

$$(2m)^e = 2^e m^e = 2^e c,$$

and similarly

$$(4m)^e = 4^e c,$$

etc., always to be understood modulo n. If $m < n/2$, then $2m < n$ and hence $2m$ is not reduced modulo n. If $m > n/2$, then $2m > n$ and hence $2m$ is reduced modulo n (i.e., n is subtracted). Because n is odd we have

$$A(2^e c) = 0 \iff m < n/2.$$

By a similar argument one can show that

$$A(4^e c) = 0 \iff \begin{cases} 0 < m < n/4 \\ \text{or} \\ n/2 < m < 3n/4 \end{cases}$$

Each value $A\big((2^i)^e c\big)$ allows us to exclude one half of the remaining possible region for m. For instance, if $A(2^e c) = 1$ we know that $m > n/2$. If in addition $A(4^e c) = 0$ we know that $n/2 < m < 3n/4$. Similarly, $A(8^e c) = 1$ tells us that $5n/8 < m < 3n/4$, etc. After k applications of algorithm A one ends up with a single value m. In other words, we can determine the binary expansion of m/n, bit-by-bit: $A(2^e c)$ is the LSB of $\lfloor 2\frac{m}{n} \rfloor$, $A(4^e c)$ is the LSB of $\lfloor 4\frac{m}{n} \rfloor$, etc. Hence, we have proved the following lemma, which concludes the proof of the theorem. □

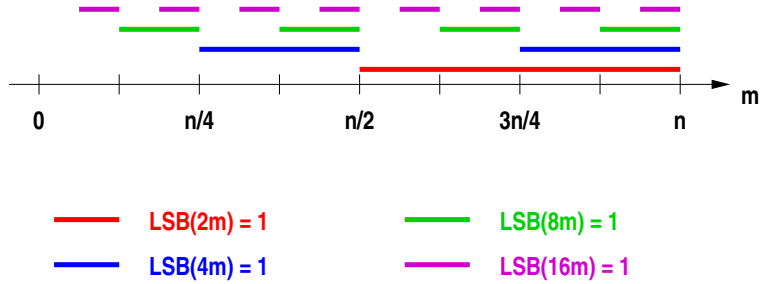

Figure 4.1: Illustration of the proof of Theorem 4.1. The regions within the interval $[0, n-1]$ are shown for which $A(2^e c) = 1$, $A(4^e c) = 1$, $A(8^e c) = 1$ and $A(16^e c) = 1$. The expression $\mathrm{LSB}(2m)$ is to be understood in the sense that $2m$ is reduced modulo n before the LSB is computed. (Similarly for the other expressions.)

Lemma 4.2. $A\big((2^i)^e c\big) = \mathrm{LSB}(\lfloor 2^i \frac{m}{n} \rfloor)$.

Let us discuss the significance (or insignificance) of Theorem 4.1. It seems to state that a single-bit message b can be securely encrypted by choosing a random m with LSB b. (Note that this is a randomized PKE scheme.) However, Theorem 4.1 is not strong enough to prove the IND-CPA security of this scheme. One would need that guessing the LSB correctly with probability significantly greater than 0.5 allows to invert the RSA function.[5] Indeed, such a stronger statement can be proved. Moreover, an even stronger result can be proved:[6] The $\log k$ least significant bits are simultaneously unapproximable if inverting the RSA function is hard. More precisely, an algorithm which distinguishes the $\log k$ least significant bits of a random plaintext from uniformly random bits with non-negligibly advantage can be used to compute the entire plaintext. This shows that one could use the naïve RSA encryption (with randomization) to encrypt a few bits.

We note that a reduction statement of this type can be read in two different directions, which are each valid:

- The least significant bits are secure if we (only) assume the hardness of inverting the RSA function.
- The RSA function can be inverted if one can slightly distinguish the least significant bits from random. This task appears much easier than inverting RSA, and this can be seen as casting doubt on the security of RSA.

We also note that in a reasonable cryptographic design, the RSA-function is usually only used as a TOWP and not as a (naïve) public-key encryption scheme. In this case the relevance of understanding the security of the LSB is unclear.

[5] This statement can be rephrased as follows: The LSB of the RSA function is a *hard-core predicate*.

[6] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr, RSA and Rabin functions: Certain parts are as hard as the whole, *SIAM Journal on Computing*, 17(2):194–209, 1988.

4.3 Example 2: Guessing the LSB of a Discrete Logarithm

4.3.1 Introduction

In this section we discuss another example of a cryptographic reduction. This example is more interesting than that of the previous section, for several reasons. First, it is related to the result of a famous paper.[7] Second, the reduction goes all the way from the hardness of computing a value (a discrete logarithm) to showing that a certain bit cannot even be guessed with a slight advantage over a random guess. This result can be used to obtain a pseudo-random generator based on the sole assumption that the DL problem in the given group is worst-case hard. Third, the reduction is obtained in a sequence of steps, which can each be understood as an individual reduction, where the overall statement is obtained as the composition of these individual reductions.

The overall treatment is quite involved, and this provides motivation for our abstract approach to defining and understanding reductions, which is to follow in the rest of the chapter.

Already in this section we use terminology introduced later in the chapter: We use the term *solver* (instead of algorithm) for the object that solves a computational problem. We also use the term *performance* to measure the amount of success of a solver:[8]

- For a game, the performance of a solver is the probability of winning the game (see Definition 2.1).
- For a bit-guessing problem like the LSB-problem, the performance of a solver is the advantage in guessing the bit correctly, i.e., $2p - 1$ if p is the probability of guessing the bit correctly (see Definition 2.5).

A *worst-case* problem means that one requires a solver to achieve a certain performance for *all* instances of the problem. In other words, the performance of a solver for a worst-case problem is defined as the minimum of the performance over all instances of the problem.

An *average-case* problem means that one requires a solver to achieve a certain performance on average over the instances of the problem chosen according to a specified distribution.

[7] see Manuel Blum and Silvio Micali, How to Generate Cryptographically Strong Sequences of Pseudorandom Bits, *SIAM Journal on Computing*. vol. 13, no. 4, 1984, pp. 850–864.

[8] The term performance does *not* refer to efficiency or speed.

4.3.2 The Problem Statement

We consider a group $\mathcal{H} = \langle h \rangle$ of order $|\mathcal{H}| = m$, generated by the generator h, and we consider two computational problems for $\mathcal{H}$:

- the DL-problem in $\mathcal{H}$, i.e., given $y = h^x$, compute x;
- the LSB-problem in $\mathcal{H}$, i.e., given $y = h^x$, compute the least significant bit (LSB) of x.

Since the LSB-problem is easy for even group order (see exercise below) we assume from now on that $|\mathcal{H}| = m$ is odd.

Exercise 4.2. Show that if the group order m is even, then guessing the LSB is easy even in the worst case, i.e., there exists an efficient solver for the worst-case LSB-problem with performance 1.

We use the following notation to denote different computational problems for the group $\mathcal{H}$. We use DL and LSB to refer to discrete logarithm and LSB problems, respectively. A set $\mathcal{I}$ as superscript for $\mathcal{I} \subseteq \mathcal{H}$ denotes the worst-case problem for the instance set $\mathcal{I}$. Often $\mathcal{I} = \mathcal{H}$. A $\mathcal{H}$-valued random variable Y as superscript denotes the average-case problem when the instance is (chosen according to the distribution of) Y. A uniform random variable over $\mathcal{H}$ is denoted as H. The computational problems that will arise in our context are:

- $\mathsf{DL}^{\mathcal{H}}$ denotes the worst-case DL-problem (for the full instance set $\mathcal{H}$).
- $\mathsf{DL}^{\mathcal{I}}$ denotes the worst-case DL-problem for an instance set $\mathcal{I} \subseteq \mathcal{H}$.
- DL^{H} denotes the average-case DL-problem (uniform over $\mathcal{H}$).
- $\mathsf{LSB}^{\mathcal{H}}$ denotes the worst-case LSB-problem (for the full instance set $\mathcal{H}$).
- $\mathsf{LSB}^{\mathcal{I}}$ denotes the worst-case LSB-problem for an instance set $\mathcal{I} \subseteq \mathcal{H}$.
- LSB^{H} denotes the average-case LSB-problem (uniform over $\mathcal{H}$).

Our goal is to prove that if the DL-problem is hard (the assumption) then the apparently much simpler problem of guessing (only) the least significant bit (LSB) of the discrete logarithm, is also hard (the conclusion). The goal is to prove such a statement

- for the *weakest* possible form of the assumption, and
- for the *strongest* possible form of the conclusion.

The (weak) assumption is that solving the worst-case DL-problem $\mathsf{DL}^{\mathcal{H}}$ with performance close to 1 is hard. This means that there is no efficient algorithm that can solve *every* instance of the DL-problem with probability at least $1-\delta$, for some small $\delta > 0$. Note that this assumption is weaker than the assumption that solving the worst-case DL-problem with performance ϵ for some $\epsilon > 0$ is hard,

and it is also weaker then the assumption that the average-case DL-problem DL^{H} with performance $1 - \delta$ is hard. In other words, our assumption does not exclude the existence of an efficient worst-case DL-algorithm that works only mildly well or one that works very well on average.

The (strong) conclusion is that the average-case LSB-guessing problem LSB^{H} is hard, even if only a small advantage (i.e., a guessing probability slightly above 0.5) is required. Note that this is a (much) stronger statement than that the worst-case LSB-problem $\mathsf{LSB}^{\mathcal{H}}$ is hard.

The described statement corresponds to the following theorem, which is a reduction statement. It is only informally stated; what "simple operations" is not made formal.

Theorem 4.3. *For any $\delta, \epsilon > 0$, and for any solver S for LSB^{H} with performance at least ϵ, there exists a solver for $\mathsf{DL}^{\mathcal{H}}$ with performance at least $1 - \delta$ which invokes S a polynomial (in $\log(1/\delta)$ and $1/\epsilon$ and $\log m$) number of times and otherwise performs only a few simple operations.*[9]

From this theorem one can conclude the following corollary, assuming that negligible is made precise in an adequate manner. This is a typical formulation of Theorem 4.3 appearing in the literature.

Corollary 4.4. *If for some $\delta < 1$ there exists no polynomial-time algorithm for solving $\mathsf{DL}^{\mathcal{H}}$ with performance at least $1 - \delta$, then there exists no polynomial-time algorithm for solving LSB^{H} with non-negligible performance.*

Theorem 4.3 is proved by the composition of a sequence of reductions. We develop a natural line of thinking in which these individual reductions are developed, making explicit the various interesting and clever ideas that one needs in order to succeed.

4.3.3 Idea 1: Using an LSB-oracle to Compute the DL

Recall that raising any element $y \in \mathcal{H}$ to the group order m yields the neutral element (denoted as 1), i.e., we have

$$y^m = 1.$$

We first describe two useful operations on group elements. Given $y = h^x$ with $x > 0$ one can reduce x by 1 (modulo m), i.e., one can compute $y' = h^{x-1}$) as follows:

$$y' = y \cdot h^{-1}.$$

[9] The theorem is not a totally precise mathematical statement since we have not defined "a few simple operations", but the proof will make explicit which operations are needed and one will see that they are simple. One could make a more formal statement involving the notion of polynomial-time.

Moreover, the (unique) square root z of $y = h^x$ can be computed as

$$z = y^{(m+1)/2}$$

because $z^2 = y^{m+1} = y$. If x is even, then $z = h^{x/2}$, i.e., the binary representation of x is shifted to the right by one bit.

If we have a solver S for $\mathsf{LSB}^{\mathcal{H}}$ that is always correct (i.e., with performance 1), then a solver T for $\mathsf{DL}^{\mathcal{H}}$ can be obtained as follows. Given an instance $y = h^x$, T determines the LSB of x by invoking S. Then, T computes y' for which the LSB of x is masked: If the LSB of x is 1 (i.e., x is odd), T computes $y' = y \cdot h^{-1}$, otherwise it sets $y' = y$. If we define x' by $y' = h^{x'}$, then x' is even. Thus T can shift (the binary representation of) x' to the right by one bit by computing

$$z = y'^{(m+1)/2} = h^{x'/2} = h^{\lfloor x/2 \rfloor}.$$

Then T sets $y = z$ and repeats the above steps until $z = h$, thus computing all the bits of x, and hence x. This requires at most $\lceil \log_2 x \rceil \leq \lceil \log_2 m \rceil$ invocations of S.

We next observe that the above argument also works if S guesses the LSB correctly only with probability at least $1 - \delta'/2$ for any instance, i.e., if it has performance $1 - \delta'$ for $\mathsf{LSB}^{\mathcal{H}}$. In this case, the constructed solver T for $\mathsf{DL}^{\mathcal{H}}$ has success probability (and hence performance) at least $1 - \delta' \lceil \log_2 x \rceil / 2$. This follows from the union bound of probability theory.[10] The algorithm is incorrect only if at least one of the invocations of S yields a wrong result. Since we will want the performance to be at least $1 - \delta$, one has to choose δ' accordingly.

4.3.4 Idea 2: From Worst-Case to Average Case

We actually have to work with a much weaker LSB-solver, namely one for which only the *average-case* performance is guaranteed and, moreover, where the performance is *very small*, namely only guaranteed to be at least ϵ.

A technique that sometimes works for solving the first problem is to randomize the instance so that it becomes an average-case instance. A technique that sometimes works for solving the second problem is to amplify the performance by repetition (see idea 4 below).

A given instance $y = g^x$ can be randomized by choosing a random $r \in \mathbf{Z}_m$ and computing the randomized instance

$$y' = y \cdot g^r = g^{x+r}$$

(where exponents are reduced modulo m). Unfortunately, this randomization does not work because if

$$x + r \geq m,$$

[10] The union bounds states the simple fact that the probability of the union of events is upper bounded by the sum of the probabilities.

i.e., if $x + r$ is reduced modulo m (which we call an overflow). In this case, the LSB changes because m is odd. Since x is unknown, one does not know whether the LSB was changed, and hence the LSB-solver's answer for the instance y' is useless.

We therefore need another idea and come back to idea 2 in Section 4.3.7.

4.3.5 Idea 3: Restriction to the Initial Segment

The above randomization technique would work well if x were known to be very small. So the following idea can be used. For a fixed integer parameter n, we divide the interval $[0, m-1]$ for x into n segments of length $\lceil m/n \rceil$ (where the last segment can be shorter) and assume we have a solver that works if x is in the first segment. Let

$$\mathcal{I} = \{h^0, h^1, \ldots, h^{\lceil m/n \rceil - 1}\} \subseteq \mathcal{H}$$

denote the "initial segment" of the instance set of the problems we consider. The worst-case DL- and LSB-problems for instances in $\mathcal{I}$ are denoted as $\mathsf{DL}^{\mathcal{I}}$ and $\mathsf{LSB}^{\mathcal{I}}$, respectively. Moreover, let

$$t = \lceil \log_2(\lfloor m/n \rfloor + 1) \rceil$$

be the number of bits needed to represent the discrete logarithm of an element of $\mathcal{I}$.

For this idea to work we need to show three things.

- First, a solver S for $\mathsf{DL}^{\mathcal{I}}$ can be used to construct a solver for $\mathsf{DL}^{\mathcal{H}}$ (this is idea 3). If we have a worst-case solver S for the initial segment, i.e., for $\mathsf{DL}^{\mathcal{I}}$, then we can obtain a worst-case solver T for $\mathsf{DL}^{\mathcal{H}}$, as follows. For $\ell = 0, \ldots, n-1$, T computes

  $$y' = y \cdot g^{-\ell \lceil m/n \rceil} = g^{x - \ell \lceil m/n \rceil}$$

  and invokes S for instance y'. One of these shifted instances is guaranteed to be in $\mathcal{I}$. If a solution x' is reported by S, then it is tested for correctness (which is possible efficiently), and the shift is adjusted to obtain the correct DL-value:

  $$x = x' + \ell \lceil m/n \rceil.$$

  T needs to invoke S at most n times.

- Second, we need to observe that for instances of $\mathsf{DL}^{\mathcal{I}}$, the above-described technique for using an $\mathsf{LSB}^{\mathcal{H}}$-solver to construct a $\mathsf{DL}^{\mathcal{H}}$-solver (idea 1 above) works also if we only have an $\mathsf{LSB}^{\mathcal{I}}$-solver. This is the case because all instances for which the LSB-solver S is invoked have exponents in the interval $\mathcal{I}$.

- Third, we need to investigate how the randomization and amplification technique works for instances known to be in $\mathcal{I}$ (see below).

4.3.6 Idea 4: Performance Amplification for Worst-Case Solvers

Given a solver S for $\mathsf{LSB}^{\mathcal{I}}$ with performance ϵ' we can construct a solver T for $\mathsf{LSB}^{\mathcal{I}}$ with performance $1-\delta'$ by statistical amplification: For an adequate parameter q, T invokes S q times and then takes a majority decision. By the law of large numbers, it suffices to choose q roughly on the order of $1/(\epsilon')^2$ and $\log(1/\delta')$.

Exercise 4.3. Work this out.

4.3.7 Making Idea 2 Work: Randomizing the Instance

A solver S for LSB^{H} (i.e., for the uniform distribution) with performance ϵ can be used to construct a solver T for $\mathsf{LSB}^{\mathcal{I}}$ with performance ϵ', for ϵ' to be discussed below, as follows.

Given $y = g^x$ with $y \in \mathcal{I}$, T chooses an offset r uniformly at random from $[0, k-1]$ for k discussed below, adds the offset r to x by computing

$$y' = y \cdot h^r = h^{x+r},$$

calls S (once) for the instance y', receives the answer b of S, and returns $b \oplus r_0$, where r_0 is the LSB of r.[11]

To avoid an overflow (i.e., $x + r \geq m$), we must have $0 \leq r \leq k-1$ for

$$k = m - \lfloor m/n \rfloor.$$

Note that $x + k - 1 < m$. An instance $x \in \mathcal{I}$ is hence randomized to a uniform instance in the interval

$$J(x) = \{h^x, h^{x+1}, \ldots, h^{x+k-1}\},$$

which depends on x. The size of this interval satisfies

$$|J(x)| \;=\; k \;=\; m - \lfloor m/n \rfloor \;\geq\; m(1 - 1/n).$$

4.3.8 Idea 5: Changing the Instance Distribution

The problem with this approach is that we consider the computational problem LSB^{Y_x} for Y_x uniform in $J(x)$ instead of the problem LSB^{H} (for H uniform in $\mathcal{H}$), for which we assumed that the performance of S is at least ϵ. This problem can be addressed by using the following fact (which the reader can prove as an exercise).

[11] The answer of S must be complemented if r is odd (by computing $b \oplus r_0$) because if b is the correct LSB of $x + r$, then $b \oplus r_0$ is the correct LSB of x (provided there is no overflow).

Exercise 4.4. Prove that for a bit-guessing problem and a solver S for it, if one changes the instance distribution by at most d in terms of statistical distance, then the performance of S changes by at most $2d$.[12] Also state this formally.

Exercise 4.5. Prove that in our example, the statistical distance between the instance distribution of LSB^{Y_x} (for any x) and LSB^{H} is at most $1/n$, i.e., for all $x \in \mathcal{I}$,

$$\delta(Y_x, H) \;\leq\; 1/n.$$

Using the results of these two exercises, we conclude that the performance of solver T for LSB^{Y_x} is at least

$$\epsilon' \;=\; \epsilon - 2/n.$$

4.3.9 Putting Things Together

For given δ and ϵ in Theorem 4.3, one can choose the parameters n and q involved in the above reductions so as to obtain an overall reduction with as small as possible complexity blow-up.

The parameter n must be chosen so that $\epsilon > 2/n$ since otherwise ϵ' is not positive. One can for example choose $n = 4/\epsilon$. The parameter q must be chosen sufficiently large so as to provide the necessary amplification.

4.3.10 Summary

Let us summarize the complete argument, which consists of five steps, each making use of one of the ideas and providing a reduction of a computational problem Q to another computational problem P (where $P = Q$ is possible). Each such reduction is characterized by the following two factors:

- The mapping of a solver S for P to a solver T for Q.
- The mapping translating the performance of S to the performance of T.

The five relevant computational problems are $\mathsf{DL}^{\mathcal{H}}$, $\mathsf{DL}^{\mathcal{I}}$, $\mathsf{LSB}^{\mathcal{I}}$, LSB^{Y_x}, and LSB^{H}. The five reduction steps, in the order as they are composed (not in the order in which they were discussed), are:

- $\mathsf{DL}^{\mathcal{H}}$ is reduced to $\mathsf{DL}^{\mathcal{I}}$ (idea 3): The given instance is shifted (at most) n times such that one of the shifted instances lies in the initial segment I. The solver for $\mathsf{DL}^{\mathcal{I}}$ must be called at most n times, and the performance remains the same.

[12] The same statement holds for games, without the factor 2.

- $\mathsf{DL}^{\mathcal{I}}$ is reduced to $\mathsf{LSB}^{\mathcal{I}}$ (idea 1): The solver for $\mathsf{LSB}^{\mathcal{I}}$ is called $t = \lceil \log_2(\lfloor m/n \rfloor + 1) \rceil$ times. If the performance for $\mathsf{LSB}^{\mathcal{I}}$ is $1 - \delta'$, then the performance for $\mathsf{DL}^{\mathcal{I}}$ is $1 - 2t\delta'$. Since this should be at least $1 - \delta$, we have to choose $\delta' = \delta/(2t)$.
- $\mathsf{LSB}^{\mathcal{I}}$ is reduced to itself (idea 4): The solver for $\mathsf{LSB}^{\mathcal{I}}$ is called q times, such that the performance is amplified from ϵ' to $1 - \delta'$.
- $\mathsf{LSB}^{\mathcal{I}}$ is reduced to LSB^{Y_x} (idea 2): The solver for LSB^{Y_x} is called once, and the performance remains identical (namely ϵ').
- LSB^{Y_x} is reduced to LSB^{H} (idea 5): The solver is the same but the performance is reduced by $2/n$ from ϵ to $\epsilon' = \epsilon - 2/n$.

4.3.11 Some Remarks

We point out that, formally, the reduction of $\mathsf{LSB}^{\mathcal{I}}$ to LSB^{Y_x} via randomization is actually a reduction for a specific x, i.e., for a problem with a single instance. The claimed reduction holds for every x, and the reduction of LSB^{Y_x} to LSB^{H} (idea 5) also works for every $x \in I$. Therefore their combination works for every $x \in \mathcal{I}$ and hence for the problem $\mathsf{LSB}^{\mathcal{I}}$.[13]

4.4 Abstract Computational Problems

4.4.1 Introduction

The goal of complexity theory is to investigate the computational complexity of computational problems. More precisely, one wants to characterize the best achievable performance in solving the problem as a function of the complexity of the solver. In other words, for a given complexity value c one maximizes the achievable performance over all solvers with complexity at most c.

To achieve this goal, one would need a concrete computational model in which complexity is defined. The choice of such a model would, at least to some extent, be arbitrary, and such a treatment would detract from the essence of what reductions are about. We therefore provide a more abstract treatment of reductions. However, one could always take a more concrete viewpoint by thinking of a solver as the set of all algorithms of a given complexity, i.e., one can think of a solver as being a complexity value if one insists on taking a complexity-based viewpoint.

[13] This illustrates (once more) that reductions make sense even for problems which are not hard, for example a problem with a single instance. Reductions are not only used to prove hardness implications.

4.4.2 Preliminaries

For the composition of functions f and g we write gf instead of $g \circ f$. Hence we write, for example, $hgf(a)$ instead of $h(g(f(a)))$. Sometimes we write simply $f\,a$ instead of $f(a)$.

For functions $f_i : \mathcal{A}_i \to \mathcal{B}_i$ (for $1 \leq i \leq k$), $(f_1, \ldots, f_k)$ denotes the direct product of $f_1, \ldots, f_k$, i.e., the function $\mathcal{A}_1 \times \cdots \times \mathcal{A}_k \to \mathcal{B}_1 \times \cdots \times \mathcal{B}_k$ defined by

$$(f_1, \ldots, f_k)(a_1, \ldots, a_k) \;=\; (f_1(a_1), \ldots, f_k(a_k)).$$

For functions $f_i : \mathcal{A} \to \mathcal{B}_i$ (for $1 \leq i \leq k$), $[f_1, \ldots, f_k]$ denotes the function $\mathcal{A} \to \mathcal{B}_1 \times \cdots \times \mathcal{B}_k$ defined by

$$[f_1, \ldots, f_k](a) \;=\; \big(f_1(a), \ldots, f_k(a)\big).$$

The function assigning to a list of real numbers their sum is denoted as sum. For example, $\mathsf{sum}(3, 4, 5) = 12$. Similarly, for boolean-valued lists we denote by $\wedge$ (or $\vee$) the logical AND (or logical OR) of the values. For functions $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{A} \to \mathcal{B}$, where a partial order $\leq$ is defined on B, we write $f \leq g$ to mean $\forall a \, \big(f(a) \leq g(a)\big)$.

4.4.3 Problems, Solvers, and Performance

In a theory of computation, at the most abstract level one considers two types of objects.

- *Problems* are objects that should be computed or solved.
- *Solvers* are objects that compute or solve a problem.

Definition 4.2. A problem p is (an object) equipped with a set Σ_p of solvers, a partially ordered set $(\Omega_p; \leq)$ of performance values[14], and a *performance function*

$$\overline{p} \;:\; \Sigma_p \to \Omega_p$$

assigning a performance value $\overline{p}\,(s)$ to every solver $s \in \Sigma_p$. A solver s for which $\overline{p}\,(s) \geq a$ for $a \in \Omega_p$ is called an *a-solver* for p.

Performance values are often real numbers (i.e., $\Omega_p \subseteq \mathbb{R}$), for example a success probability or a distinguishing advantage, but one can also consider more general performance sets, for example pairs of real numbers. We will consider different types of problems, for example games with performance set $[0, 1]$ and distinction problems with performance set $[-1, 1]$.

[14] Here $a \leq b$ means that the performance value b is (in some sense) at least as good as a.

4.4.4 Upper Bounds on Performance

An important goal in cryptography and complexity theory is to prove that a certain problem p is hard (e.g., the problem of breaking a cryptosystem). Ideally, one wants to prove an upper bound on the performance of any solver for p, i.e., a statement of the form $\forall s\ \ \overline{p}\,(s) \leq \epsilon$ for some small ϵ. This can also be written simply as

$$\overline{p}\ \leq\ \epsilon,$$

where ϵ is understood as the constant function $\Sigma_p \to \Omega_p$ mapping every solver to the value ϵ. (Note that usually ϵ is real-valued.) Such a statement is often called *information-theoretic* or *unconditional* since it holds for any solver. In particular, it does not depend on the solver's implementation complexity.

However, since for a usual computational problem p there always exists a (possibly inefficient) solver that solves the problem well (i.e., with high performance), one proves a weaker statement where the upper bound depends on the performance function $\overline{q}$ of another problem q. Such a statement is called a reduction. More generally, $\overline{p}$ can also be upper bounded in terms of the performance functions $\overline{q_i}$ $(i = 1, \ldots, k)$ of several problems.

4.5 Abstract Reductions

4.5.1 The Reduction Concept

An important concept in computer science is to use a solver s for a problem p with performance at least a (i.e., $\overline{p}\,(s) \geq a$) to construct a solver s' for a problem q with performance at least a' (i.e., $\overline{q}\,(s') \geq a'$), where $s' = \rho\,(s)$ for a function

$$\rho\ :\ \Sigma_p \to \Sigma_q$$

(called the *reduction function*) and where $a' = \tau\,(a)$ for a $\leq$-respecting function

$$\tau\ :\ \Omega_p \to \Omega_q$$

(called the *performance translation function*).[15] This can be summarized by the function inequality

$$\tau\,\overline{p}\ \leq\ \overline{q}\,\rho \tag{4.1}$$

which states that for any a-solver s for p, $\rho\,(s)$ is a $(\tau\,(a))$-solver for q:

$$\forall s \in \Sigma_p :\ \tau\,\overline{p}\,(s)\ \leq\ \overline{q}\,\rho\,(s).$$

In the following, performance translation functions will tacitly be assumed to be $\leq$-respecting.

[15] τ is $\leq$-respecting if

$$a \leq b\ \Rightarrow\ \tau\,(a) \leq \tau\,(b),$$

i.e., a better solver for s does not result in a worse solver for s'.

A special case of (4.1) is $\overline{p}\ \leq\ \overline{q}\,\rho$, where τ is the identity function, i.e., ρ transforms a solver for p into an equally good solver for q. Another special case of (4.1) is $\tau\,\overline{p}\ \leq \overline{q}$, where ρ is the identity function, i.e., every solver for p is also a solver for q with performance related via the function τ.

Definition 4.3. For functions ρ and τ satisfying (4.1), ρ is called a *τ-reduction* of problem q to problem p.[16]

4.5.2 Interpretations of Reductions

The usefulness and strength of a reduction is determined by two factors:

- by how much ρ blows up the complexity of a solver s (i.e., reduces the efficiency), and
- by how much τ reduces the performance. (In some cases, τ can also increase the performance.)

Inequality (4.1) can be interpreted in two different ways: as implying a lower bound on $\overline{q}$ (in terms of $\overline{p}$), or as implying an upper bound on $\overline{p}$ (in terms of $\overline{q}$). For the latter interpretation it is convenient to write the inequality in a different form, by considering a $\leq$-respecting function $\lambda :\ \Omega_q \to \Omega_p$ satisfying

$$\mathsf{id}\ \leq\ \lambda\,\tau \tag{4.2}$$

(i.e., $a \leq \lambda\tau\,a$ for all $a \in \Omega_q$). If Ω_p and Ω_q are intervals of $\mathbb{R}$ and if τ is strictly $\leq$-respecting, then one can choose $\lambda = \tau^{-1}$ (i.e., $\mathsf{id} = \lambda\,\tau$). We have

$$\begin{aligned} \tau\,\overline{p}\ \leq\ \overline{q}\,\rho\ \ &\Longrightarrow\ \ \lambda\,\tau\,\overline{p}\ \leq\ \lambda\,\overline{q}\,\rho \\ &\Longrightarrow\ \ \overline{p}\ \leq\ \lambda\,\overline{q}\,\rho\,, \end{aligned} \tag{4.3}$$

where in the first step we have used the $\leq$-respecting property of λ. If equality holds in (4.2), then (4.3) and (4.1) are (logically) equivalent. Inequality (4.3) is the form of a reduction statement used to state an upper bound on the performance of any solver for $\overline{p}$, in terms of the function $\overline{q}$.

4.5.3 Complexity-Theoretic Interpretation

This subsection is not needed for understanding the rest of the course.

Recall the discussion of Section 4.4.1 about complexity. Let us assume that all problems we consider have the same solver set Σ and that we have fixed a computational model with a complexity notion assigning to every solver $s \in \Sigma$ a complexity value $\gamma(s)$ in some domain Γ of complexity values, for example $\Gamma = \mathbb{N}$. Here

$$\gamma :\ \Sigma \to \Gamma$$

[16] The term "reduction" is motivated by the fact that, by virtue of the reduction, the problem of finding a solver for q is "reduced" to the problem of finding a solver for p.

is the *complexity function* and $\gamma(s)$ is the implementation complexity when s is optimally implemented in the computational model. We can then associate with a complexity value c the class Σ_c of all solvers with at most that complexity:

$$\Sigma_c \;=\; \big\{s \,|\, \gamma(s) \leq c\big\}.$$

We also consider a reduction $\rho : \Sigma \to \Sigma$.

Now, one particular interpretation of abstract reductions is that, formally, the (derived) solver set Σ' is the set Γ of complexity values, and the (derived) performance function

$$\overline{p}' :\; \Gamma \to \Omega$$

maps complexities to performance and assigns to a complexity value c the best achievable performance for any solver with complexity at most c:

$$\overline{p}'(c) \;=\; \sup\big\{\overline{p}\,(s) \,|\, s \in \Sigma_c\big\}.$$

Moreover, the (derived) reduction function ρ' is a function $\Gamma \to \Gamma$ mapping complexities to complexities. A complexity value c is mapped to the maximal possible complexity of a solver resulting from applying the reduction ρ to any solver with complexity at most c:

$$\rho'(c) \;=\; \sup\big\{\gamma(\rho\,(s)) \,|\, s \in \Sigma_c\big\}.$$

We do not make this more formal but point out that a computational model would have to come with an exact description of the function ρ'. This would mean, among other things, that the maximal complexity of an algorithm A with complexity c invoking an algorithm B with complexity c' would need to be defined as a function of c and c'. This is very tedious, and to some extent arbitrary, already for simple computational models. The use of asymptotics allows to eliminate the arbitrariness to some extent, but it comes at the (high) price of a loss of concreteness and significance.

4.5.4 Composition of Reductions

Reductions can be composed:

Lemma 4.5. $\tau\,\overline{p} \leq \overline{q}\,\rho \;\;\wedge\;\; \tau'\overline{q} \leq \overline{r}\,\rho' \;\;\Longrightarrow\;\; \tau'\tau\,\overline{p} \;\leq\; \overline{r}\,\rho'\rho\,.$

Proof. Composing both sides of $\tau\,\overline{p} \;\leq\; \overline{q}\,\rho$ with (the $\leq$-respecting function) τ' on the left side results in $\tau'\tau\,\overline{p} \;\leq\; \tau'\overline{q}\,\rho$ since τ' is $\leq$-respecting. Composing both sides of $\tau'\overline{q} \;\leq\; \overline{r}\,\rho'$ with ρ on the right side results in $\tau'\overline{q}\,\rho \;\leq\; \overline{r}\,\rho'\rho$.[17] Combining the two inequalities yields $\tau'\tau\,\overline{p} \;\leq\; \overline{r}\,\rho'\rho\,.$ □

Reduction statements of the form (4.3) also compose:

Lemma 4.6. $\overline{p} \leq \lambda\,\overline{q}\,\rho \;\;\wedge\;\; \overline{q} \leq \lambda'\,\overline{r}\,\rho' \;\;\Longrightarrow\;\; \overline{p} \;\leq\; \lambda\,\lambda'\,\overline{r}\,\rho'\rho\,.$

[17] Note that $\tau'\overline{q} \leq \overline{r}\,\rho'$ means that $\tau'\overline{q}\,(s) \leq \overline{r}\,\rho'(s)$ for all $s \in \Sigma_p$, and hence it also holds for all s in the image of ρ, which is a subset of Σ_q. Therefore $\tau'\overline{q}\,\rho \leq \overline{r}\,\rho'\rho\,.$

4.5.5 Generalized Reductions

A reduction statement $\tau\,\overline{p} \;\leq\; \overline{q}\,\rho$ can be generalized by letting the construction make use of a list $\boldsymbol{s} = (s_1, \ldots, s_n)$ of solvers, one for each problem in a list $\boldsymbol{p} = (p_1, \ldots, p_n)$ of problems. Letting, accordingly, τ be a $\leq$-respecting[18] function $\tau : \Omega_{p_1} \times \cdots \times \Omega_{p_n} \to \Omega_q$ and letting ρ be a function $\rho : \Sigma_{p_1} \times \cdots \times \Sigma_{p_n} \to \Sigma_q$, inequality (4.1) can be generalized to

$$\tau\,\big(\overline{p_1}\,(s_1), \ldots, \overline{p_n}\,(s_n)\big) \;\leq\; \overline{q}\,\rho\,(s_1, \ldots, s_n)$$

for all $(s_1, \ldots, s_n)$ or, equivalently, the function inequality

$$\tau\,\overline{\boldsymbol{p}} \;\leq\; \overline{q}\,\rho\,. \tag{4.4}$$

In other words, ρ yields a $(\tau\,(a_1, \ldots, a_n))$-solver $\rho\,(s_1, \ldots, s_n)$ for q from any list of a_i-solvers s_i for p_i.

A reduction statement of the form $\overline{p} \;\leq\; \lambda\,\overline{q}\,\rho$ can also be generalized by replacing q by a list $\boldsymbol{q} = (q_1, \ldots, q_k)$ of problems with the same solver set Σ, ρ by a list $\boldsymbol{\rho} = [\rho_1, \ldots, \rho_k]$ of functions with $\rho_i : \Sigma \to \Sigma_{q_i}$, and λ by a $\leq$-respecting function $\lambda : \Omega_{q_1} \times \cdots \times \Omega_{q_k} \to \Omega_p$. Let $\overline{\boldsymbol{q}} = (\overline{q_1}, \ldots, \overline{q_k})$. We can now consider the statement that

$$\overline{p}\,(s) \leq \lambda\big(\overline{q_1}\,\rho_1(s), \ldots, \overline{q_k}\,\rho_k(s)\big)$$

for all $s \in \Sigma$, which can equivalently be written as

$$\overline{p} \;\leq\; \lambda\,\overline{\boldsymbol{q}}\,\boldsymbol{\rho}\,, \tag{4.5}$$

where

$$\lambda\,\overline{\boldsymbol{q}}\,\boldsymbol{\rho} \;=\; \lambda\,\big[\overline{q_1}\,\rho_1, \ldots, \overline{q_k}\,\rho_k\big] \;=\; \lambda\,\big(\overline{q_1}, \ldots, \overline{q_k}\big)\big[\rho_1, \ldots, \rho_k\big].$$

This inequality states that the performance of a solver s for p is upper bounded by a function λ of the performances of the solvers $\rho_1(s), \ldots, \rho_k(s)$ on the problems $q_1, \ldots, q_k$, respectively. A typical choice of function λ is the sum (of real numbers).

Composition statements for generalized reductions follow similarly to Lemma 4.6.

4.5.6 Worst-Case Problems

For a set $\mathcal{P}$ of problems one can define the corresponding worst-case problem, which is formally again a problem and denotes as $\mathcal{P}$.

Definition 4.4. For a set $\mathcal{P}$ of problems with the same solver set Σ and the same performance set Ω, where $\Omega \subseteq \mathbb{R}$, the *worst-case problem*, denoted also as $\mathcal{P}$, is

[18] This means that $\Big(\forall i\;(a_i \leq b_i)\Big) \;\Longrightarrow\; \tau\,(a_1, \ldots, a_n) \leq \tau\,(b_1, \ldots, b_n)$.

the problem whose performance function $\overline{\mathcal{P}} : \Sigma \to \mathbb{R}$ is defined as follows: the performance of a solver s is the worst performance for any problem in $\mathcal{P}$:

$$\overline{\mathcal{P}}(s) \;=\; \inf_{p \in \mathcal{P}} \overline{p}(s).$$

Example 4.6. A computational problem, like the discrete logarithm problem, has typically many instances, each of which can be understood as a problem (with a single instance). Then the worst-case version of the problem (e.g. the worst-case DL-problem) is obtained by Definition 4.4.

The proof of the following lemma is obvious and left as an exercise. It states that if every problem in a set $\mathcal{Q}$ can be reduced to some problem in $\mathcal{P}$ for the same reduction function ρ and performance translation function τ, then the corresponding worst-case problems can also similarly be reduced. The lemma was implicitly used in Idea 5 in Section 4.3.8.

Lemma 4.7. *Let $\mathcal{P}$ and $\mathcal{Q}$ be sets of problems, let ρ be a reduction function and let τ be a performance translation functions. Then*

$$\forall q \in \mathcal{Q}\ \exists p \in \mathcal{P} :\ \tau\,\overline{p} \le \overline{q}\ \rho \implies \tau\,\overline{\mathcal{P}} \le \overline{\mathcal{Q}}\ \rho.$$

4.6 Basic Types of Computational Problems

In this section we discuss the three basic types of computational problems important in cryptography: games, distinction problems, and bit-guessing problems, as instantiations of the abstract problem/solver concepts of the previous section. We recall Section 2.1 where these problem types were already discussed.

4.6.1 Games and Multi-games

To instantiate the concept of a game (using the abstract concepts of Section 4.4; see also Section 2.1.1), we consider a setting with a set $\mathcal{G}$ of (deterministic) games and a set $\mathcal{W}$ of (deterministic) winners, and a function

$$\omega\colon \mathcal{W} \times \mathcal{G} \to \{0,1\},$$

where 0 and 1 denote the Boolean truth values, and $\omega(w,g) = 1$ is interpreted as w winning game g.[19]

Definition 4.5. A game G is a $\mathcal{G}$-valued random variable, a winner W is a $\mathcal{W}$-valued random variable,[20] the performance set is $\Omega = [0,1]$, and the performance of W for game G is the winning probability, i.e.,

$$\overline{G}(W) \;:=\; \mathsf{Pr}^{WG}\big(\omega(W,G) = 1\big).$$

[19] Note that $\mathcal{G}$ and $\mathcal{W}$ are understood as arbitrary (abstract) sets, not necessarily discrete systems.
[20] That is, the solver set Σ is the set of $\mathcal{W}$-valued random variables.

We have already seen several examples of games, for example the function inversion problem or the DL problem.

We often consider settings in which multiple games are defined, each of which can either be won or not by a given winner w.

Definition 4.6. A *multi-game* is a list $(g_1, \ldots, g_k)$ of games, and the g_i are called *sub-games* or *component games*. We denote by $(g_1, \ldots, g_k)^\vee$ (and by $(g_1, \ldots, g_k)^\wedge$) the game corresponding to the logical OR (the logical AND) of the k subgames, i.e., the game of winning at least one (respectively all) of the sub-games:

$$\omega(w, (g_1, \ldots, g_k)^\vee) \;=\; \bigvee_{i=1}^{k} \omega(w, g_i).$$

We state the following two simple fcts, where the second one follows from the union bound of probability theory.

Lemma 4.8. *If $\omega(w,g_1) \;\to\; \omega(w,g_2)$ for all w, then* $\overline{G_1} \;\le\; \overline{G_2}$.

Lemma 4.9. $\overline{(G_1, \ldots, G_k)^\vee} \;\le\; \mathsf{sum}\big(\overline{G_1}, \ldots, \overline{G_k}\big)$.

Example 4.7. We recall the hash-then-sign example discussed in Chapter 2, which can be described in terms of three games:

1. the hash collision event (game g_1),
2. the forgery event for the short-message signature scheme (game g_2), and
3. the forgery event for the overall (long-message) signature scheme (game g_3).

We showed in Chapter 2 that if g_3 is won, then either g_1 or g_2 is won, i.e.,

$$\omega(w,g_3) \;\to\; \omega(w,g_1) \vee \omega(w,g_2),$$

and hence we have

$$\overline{G_3} \;\le\; \overline{G_1} + \overline{G_2}$$

(also written as $\overline{G_3} \;\le\; \mathsf{sum}(\overline{G_1}, \overline{G_2})$).

4.6.2 Distinction Problems

To instantiate the concept of a distinction problem (using the abstract concepts of Section 4.4; see also Section 2.1.2), we consider a setting with a set $\mathcal{O}$ of objects, a set $\mathcal{D}$ of (deterministic) distinguishers, and a function

$$\kappa\colon \mathcal{D} \times \mathcal{O} \to \{0,1\},$$

where 0 and 1 denote the Boolean truth values.

Definition 4.7. A *distinction problem* is a pair (S_0, S_1) of $\mathcal{O}$-valued random variables (or, more precisely, distributions), and will be denoted as $\langle S_0 \mid S_1 \rangle$. A distinguisher D is a $\mathcal{D}$-valued random variable, and the performance of D for distinction problem $\langle S_0 \mid S_1 \rangle$ is

$$\overline{\langle S_0 \mid S_1 \rangle}(D) \;:=\; \Delta^D(S_0, S_1) \;=\; \mathsf{Pr}^{DS_1}\big(\kappa(D, S_1) = 1\big) - \mathsf{Pr}^{DS_0}\big(\kappa(D, S_0) = 1\big).$$

We recall Definition 2.3 that for a class $\mathcal{D}' \subseteq \mathcal{D}$ of distinguishers, the advantage is

$$\Delta^{\mathcal{D}'}(S, T) \;:=\; \sup_{D \in \mathcal{D}'} \Delta^D(S, T).$$

If one can complement the bit $\kappa(d, s)$ "output" by a distinguisher $d \in \mathcal{D}'$, i.e., if for every $d \in \mathcal{D}'$ there is a $d' \in \mathcal{D}'$ such that $\kappa(d', s) = \kappa(d, s) \oplus 1$,[21] then we have $\Delta^{d'}(S_0, S_1) = -\Delta^d(S_0, S_1)$ and therefore

$$\Delta^{\mathcal{D}}(S_0, S_1) = \sup_{D \in \mathcal{D}} \big|\Delta^D(S_0, S_1)\big|.$$

Indeed, we have:

Lemma 4.10. *If $\mathcal{D}'$ is closed under complementing the output bit (see above), then $\Delta^{\mathcal{D}'}$ is a pseudo-metric.*[22]

Proof. Left as an exercise. □

Lemma 2.2 can be restated as

$$\overline{\langle S_0 \mid S_k \rangle} \;=\; \mathsf{sum}\left(\overline{\langle S_0 \mid S_1 \rangle}, \ldots, \overline{\langle S_{k-1} \mid S_k \rangle}\right). \tag{4.6}$$

4.6.3 Bit-Guessing Problems

To instantiate the concept of a bit-guessing problem (using the abstract concepts of Section 4.4; see also Section 2.1.3), we again, as for distinction problems, consider a setting with a set $\mathcal{O}$ of objects, a set $\mathcal{D}$ of (deterministic) distinguishers, and a function κ: $\mathcal{D} \times \mathcal{O} \to \{0, 1\}$.

Definition 4.8. A *bit-guessing problem* is a pair a $\mathcal{O} \times \{0, 1\}$-valued random variable (S, B), and is denoted as $[\![S; B]\!]$. A distinguisher (or bit-guesser) D is a $\mathcal{D}$-valued random variable, and the performance of D for bit-guessing problem $[\![S; B]\!]$ is

$$\overline{[\![S; B]\!]}(D) \;:=\; \Lambda^D\big([\![S; B]\!]\big) \;=\; 2 \cdot \left(\mathsf{Pr}^{D(S,B)}\big(\kappa(D, S) = B\big) - \tfrac{1}{2}\right).$$

[21] This is the case for any reasonable set $\mathcal{D}'$.

[22] Recall that a pseudo-metric on a set $\mathcal{X}$ is a function $f : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ satisfying $f(x, x) = 0$, $f(x, y) = f(y, x)$, and $f(x, z) \leq f(x, y) + f(y, z)$ for all $x, y, z \in \mathcal{X}$ (triangle inequality).

The performance $\overline{[\![S; B]\!]}(D)$ is between -1 and 1, where performance 1 means that D's guess is correct with probability 1, and performance -1 means that D's guess is correct probability 0.

We recall Lemma 2.3 which states that

$$\overline{[\![S_U; U]\!]} \;=\; \overline{\langle S_0 \mid S_1 \rangle}, \tag{4.7}$$

where U is an unbiased random bit, independent of anything else. We also recall Lemma 2.4 which can be restated as

$$\overline{[\![S; B]\!]} \circ \rho \;=\; 2 \cdot \overline{\langle [S, B] \mid [S, U] \rangle}, \tag{4.8}$$

where ρ is the simple reduction function mapping D to D' in Lemma 2.4.