Chapter 6

Equivalences of Basic Cryptographic Functions

In real world applications the basic cryptographic functions

1. Symmetric ciphers:
   (a) bitblock ciphers
   (b) bitstream ciphers
2. Asymmetric ciphers
3. Keyless ciphers:
   (a) one-way functions
   (b) hash functions
4. Random generators:
   (a) physical random generators
   (b) (algorithmic) (pseudo-) random generators
5. Steganographic procedures

are used for the construction of cryptographic protocols. We'll see that the existence of most of them—1a, 1b, 3a, 3b, 4b in suitable variants—is equivalent with the basic problem of theoretic computer science $\mathbf{P} \stackrel{?}{\neq} \mathbf{NP}$, and thus lacks a proof.

Warning: Most parts of this section are *mathematically inexact*. Statements on complexity are formulated in the naive way and justified by heuristic arguments. Then we sketch the approach to formalizing them by TURING machines. However this model turns out as insufficient for cryptology. The mathematically sound versions of complexity results for cryptologic procedures are given in Appendix B.

6.1 One-Way Functions

We continue to use the informal definition from 4.1. An exact approach is given in Appendix B.5.

Application A natural application of one-way functions is one-way encryption. This means:

- *Everyone* can encrypt.
- *No one* can decrypt.

What is it good for if no one can decrypt? There are several meaningful applications for one-way functions, in particular for the special case of hash functions, see 6.2:

- Password management, for instance in Unix or MS Windows. No one must be able to read the password. But the operating system must be able to compare an entered password with the one it has in its data base in encrypted form. **("cryptographic matching")**
- A similar application is **pseudonymization**: Data of a person should be combined with data from the same person stored elsewhere or at other times without revealing the identity of this person.
- Another application is making **digital signatures** faster, see 6.2.
- The crucial property of asymmetric encryption is that nobody can derive the private key from the public one. However the direct naive application of one-way functions doesn't work, as we saw already for the ElGamal cipher in 4.5.

Examples of conjectured one-way functions:

1. The discrete exponential function, see 4.1
2. Consider a bitblock cipher

$$F\colon M \times K \longrightarrow C$$

that resists an attack with known plaintext. A standard trick to get a one-way function $f\colon K \longrightarrow C$ from it works as follows:

$$f(x) := F(m_0, x).$$

In words: We take a fixed plaintext m_0—maybe the all-zero block—and encrypt it with a key that is exactly the block x to be one-way encrypted. Inverting this function amounts to an attack with known plaintext m_0 on the cipher F.

3. Let $n \in \mathbb{N}$ be a composite module. From 5.2 we know that—at least in the case where n is the product of two large prime numbers—computing square roots $\bmod n$ is probably hard. Hence the squaring map $x \mapsto x^2 \bmod n$ is a probable one-way function of the residue class ring $\mathbb{Z}/n\mathbb{Z}$. Note that calculating the inverse map is possible with additional information in form of the prime factors of n. Such an additional information is called a "trapdoor". The function is then called a "trapdoor one-way function". This is the crucial security feature of the RABIN cipher.
4. The same conclusion holds for the RSA function $x \mapsto x^e \bmod n$ with an exponent e that is coprime with $\lambda(n)$ (oder $\varphi(n)$).

6.2 Hash Functions

Hash functions are the most important special cases of one-way functions. They are also known as "message digests" or "cryptographic check sums".

Definition 1 Let Σ be an alphabet and $n \in \mathbb{N}$ be a fixed integer ≥ 1. A one-way function

$$h\colon \Sigma^* \longrightarrow \Sigma^n$$

is called **weak hash function** over Σ.

It maps character strings of *arbitrary* lengths to character strings of a given *fixed* length. (Since Σ^* is infinite we interpret the one-way property as: the restriction of h to Σ^r is one-way for all sufficiently large r.

Definition 2 A one-way function $f\colon M \longrightarrow N$ is called **collision free** if there is no efficient way to find $x_1, x_2 \in M$ with $x_1 \neq x_2$, but $f(x_1) = f(x_2)$.

This is a kind of "virtual injectivity". Needless to say that true injective one-way functions are collision free. If $\#M > \#N$, then f cannot be injective, but nevertheless could be collision free.

Definition 3 A **(strong) hash function** is a collision free weak hash function.

For practical applications (mostly with $\Sigma = \mathbb{F}_2$) the length n of the hash values should be as small as possible. On the other hand to exclude efficient invertibility, and thus to get cryptographic security, n must be sufficiently large. We want a weak hash function to deliver uniformly distributed values that look statistically random, and to be safe from an exhaustion attack as illustrated in Figure 6.1. Inserting m blanks at will we generate 2^m different—but optically indistinguishable—versions of a text document. If m is large enough, with high probability one of these versions will have the given hash value.

row 1	(add blank)	$\rightarrow$ 2 different versions
$\vdots$	$\vdots$	
row i	(add blank)	$\rightarrow$ 2 different versions
	$\vdots$	$\vdots$
row m	(add blank)	$\rightarrow$ 2 different versions

Figure 6.1: An exhaustion attack: How to fake a document to have a given hash value by generating 2^m different versions

As a consequence $n = 80$ is just too weak as a lower bound, we'd better use 128-bit hashes. This is the hash length of the well-known but outdated functions MD2, MD4, MD5.

But virtually all applications even need collision free hashes. Remember the birthday paradox, see I.2.6: To exclude collisions with sufficient certainty we need about twice the bitlength than for the one-way property. So hash values of 160 bits are just below the limit. The former standard hash functions SHA-1 and RIPEMD use exactly this length. Their use is strongly discouraged. In the context of AES the hash function SHA-2 with at least 256-bit values was specified, conveniently also denoted as SHA-256 etc. [see `http://csrc.nist.gov/publications/`]. The new standard SHA-3 is valid since 2015.

In fact for the MDx functions there is a systematic way to find collisions [Dobbertin 1996ff.], also SHA-1 collisions are known (2005).

document a		document b	
row 1	(add blank)	row 1	(add blank)
$\vdots$	$\vdots$	$\vdots$	$\vdots$
row m	(add blank)	row m	(add blank)

Figure 6.2: A collision attack: How to fake a document to have the same hash value as another document

Applications

(Strong) hash functions are in use for

- digital signatures: To sign a long message with the private key would take much time due to the slowness of asymmetric ciphers. The standard procedure is to sign a hash of the message.

  For security we need a collision free hash function. Otherwise the attacker could get a valid signature for an arbitrary document a without stealing Alice's private key: He produces an innocently looking document b that Alice is glad to sign. Then he fabricates $q = 2^m$ variants $a_1, \ldots, a_q$ and $b_1, \ldots, b_q$ of both documents, for example by inserting spaces at m different positions. If he finds a collision $h(a_i) = h(b_j)$, he lets Alice sign b_j, getting a valid signature for a_i too.

- transforming a long, but memorizable passphrase ("Never change a working % password 24 because you'll ? forget the nEW one+") into an n-bit key (`BA8C0C8C1C65364F` in hexadecimal notation) for a symmetric cipher.

6.3 Conversion Tricks

We give heuristic reasons that the following statements (A) to (D) are equivalent, and that each of them implies (E)—for a formal mathematical proof we don't have yet the exact definitions.

These implications also have practical relevance for constructing a basic function given another one. A coarse summary—for the discussion on regulations of cryptography that pop up from time to time—consists of the statements

- Who wants to prohibit encryption also must prohibit hash functions and pseudo-random generators.
- Who wants to make cryptography impossible must prove that $\mathbf{P} = \mathbf{NP}$.

(A) There is a one-way function $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$.

($\tilde{\mathbf{A}}$) There is a one-way function $\tilde{f}\colon \mathbb{F}_2^{2n} \longrightarrow \mathbb{F}_2^n$.

(B) There is a weak hash function $h\colon \mathbb{F}_2^* \longrightarrow \mathbb{F}_2^n$.

(C) There is a strong symmetric cipher $F\colon \mathbb{F}_2^n \times \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$ (where "strong" means secure under a known-plaintext attack).

(D) There is a perfect pseudo-random generator $\sigma\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^{p(n)}$.

(E) $\mathbf{P} \neq \mathbf{NP}$.

Remark 1 Making the statements precise in terms of complexity theory we have to state (A) – (D) for families of functions that are parametrized by n.

Remark 2 A pseudo-random generator is perfect if for unknown $x \in \mathbb{F}_2^n$, given some bits of the output $\sigma(x)$, there is no efficient way to predict some more bits of the output, or to compute x. In the specification p is a polynomial with integer coefficients—from a "seed" of length n the generator produces $p(n)$ bits.

We omit reasoning about the implication "(D) $\Longrightarrow$ (E)".

"(C) $\Longrightarrow$ (D)": Set $\sigma(x) = (s_1, \ldots, s_{p(n)/n})$ with $s_0 := x$ and $s_i := F(s_{i-1}, z)$ for $i \geq 1$, where the key z is a secret constant parameter. Note the similarity with the OFB mode for bitblock ciphers. For no block s_i of the sequence the attacker is able to determine the previous block s_{i-1}—otherwise the cipher wouldn't be secure. It is not obvious that this property suffices to show perfectness, we'll show this in Chapter IV.

"(D) $\Longrightarrow$ (C)": Consider the bitstream cipher that uses $\sigma(x)$ as bitstream and x as key.

"(A) $\Longrightarrow$ (C)": There is a simple approach by E. BACKUS: Set $F(a,k) = a + f(k)$. Under a known-plaintext attack a and $c = F(a,k)$ are known. Hence also $f(k) = c - a$ is known. So the attack reduces to inverting f.

[Other approaches: MDC (= Message Digest Cryptography) by P. GUTMANN, or the FEISTEL scheme.]

"(C) $\Longrightarrow$ (A)": See the example in Section 6.1.

"(A) $\Longrightarrow$ ($\tilde{\text{A}}$)": Define $\tilde{f}$ by $\tilde{f}(x,y) := f(x+y)$. Assume we can compute a pre-image (x,y) of c for $\tilde{f}$. Then this gives also the pre-image $x+y$ of c for f.

"($\tilde{\text{A}}$) $\Longrightarrow$ (B)": Pad $x \in \mathbb{F}_2^*$ with (at most $n-1$) zeroes, giving $(x_1, \ldots, x_r) \in (\mathbb{F}_2^n)^r$. Then set

$$\begin{aligned} c_0 &:= 0, \\ c_i &:= \tilde{f}(c_{i-1}, x_i) \quad \text{for } 1 \le i \le r, \\ h(x) &:= c_r. \end{aligned}$$

This defines $h\colon \mathbb{F}_2^* \longrightarrow \mathbb{F}_2^n$.

Let $y \in \mathbb{F}_2^n$ be given. Assume the attacker finds a pre-image $x \in (\mathbb{F}_2^n)^r$ with $h(x) = y$. Then she also finds a $z \in (\mathbb{F}_2^n)^2$ with $\tilde{f}(z) = y$, namely $z = (c_{r-1}, x_r)$ (where $y = c_r$ in the construction of h).

"(B) $\Longrightarrow$ (A)": Restricting h to $\mathbb{F}_2^n$ also gives a one-way function.

6.4 Physical Complexity

The obvious approach to assessing the complexity of an algorithm is counting the primitive operations that a customary processor executes, or, more exactly, counting the clock cycles. This would lead to concrete results like: "Computing ... costs at least (say) 10^{80} of the following steps: ...". For example we could count elementary arithmetical operations (additions, multiplications, ...), taking into account the word size of the processor (e. g. 32 bits) and the number of clock cycles for the considered operations. [Note that this number might not be uniquely defined on a modern CPU with pipeline architecture.]

For many concrete algorithms statements of this kind are possible, and often lead to interesting mathematical problems as abundantly demonstrated by D. KNUTH in his books.

Unfortunately no flavour of complexity theory yields results on the *minimum* number of steps that *each* algorithm for solving a certain problem must execute, except for extremely simple problems like evaluating a polynomial for a certain argument. If we knew results of this kind, we could mathematically prove the security of cryptographic procedures without recurring to unproven conjectures or heuristic arguments.

This kind of reasoning could take into account physical bounds that limit the resources computers in this universe can dispose of. A known estimate of this kind was proposed by Louis K. SCHEFFER in `sci.crypt`:

- Our universe contains at most 10^{90} elementary particles. This is certainly an upper bound for the number of available CPUs.
- Passing an elementary particle with the speed of light takes at least 10^{-35} seconds. This is certainly a lower bound for the time required by a single operation.
- Our universe has a life span of at most 10^{18} seconds ($\approx 30 \times 10^9$ years). This is certainly an upper bound for the available time.

Multiplying these bounds together we conclude that at most $10^{143} \approx 2^{475}$ operations can be executed in our universe. In particular 500-bit keys are secure from exhaustion ...

> ... until such time as computers are built from something other than matter, and occupy something other than space. (Paul CISZEK)

Note that this security bound holds for the one algorithm "exhaustion". It has no relevance for the security of even a single cryptographic procedure! (As long as there is no proof that no attack is faster than exhaustion.)

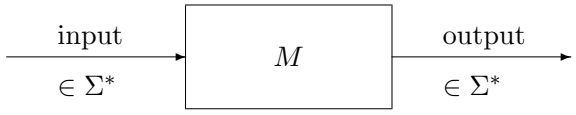
Needless to say that a realistic upper bound is smaller by several orders of magnitude.

For comparision we list some cryptologically relevant quantities:

seconds/year	3×10^7
CPU cycles/year (1 GHz CPU)	3.2×10^{16}
age of our universe (years)	10^{10}
CPU cycles since then (1 GHz)	3.2×10^{26}
atoms of the earth	10^{51}
electrons in our universe	8.37×10^{77}
ASCII strings of length 8 (95^8)	6.6×10^{15}
binary strings of length 56 (2^{56})	7.2×10^{16}
binary strings of length 80	1.2×10^{24}
binary strings of length 128	3.4×10^{38}
binary strings of length 256	1.2×10^{77}
primes with 75 decimal places (about 250 bits)	5.2×10^{72}

6.5 TURING Machines

The mathematical results of complexity theory consist almost exclusively of asymptotic cost estimates, and in almost all cases these estimates are upper bounds. Complexity theory in its various flavours relies on diverse models of computation. In this section we shortly sketch the common formalism by TURING machines.



Here Σ (as usual) denotes a finite alphabet. The input is a finite string on a tape that is infinite in both directions. The TURING machine M can assume states from a finite set that also contains a state "halt". Depending on the state the machine executes certain operations, for instance reads one character from the tape, changes its state, writes one character to the tape, moves the reading head by one position to the left or to the right. If M reaches the state "halt", then the current string on the tape is the output.

Let $L \subseteq \Sigma^*$ be a language. If M reaches the "halt" state after a finite number of steps for all inputs $x \in L$, then we say that M **accepts the language** L. If $f\colon L \longrightarrow \Sigma^*$ is a function, and M reaches "halt" after finitely many steps for each $x \in L$ with output $f(x)$, then we say that M **computes** f.

With some effort, and not too overwhelming elegance, we can describe all algorithms by TURING machines. Then by counting the steps we may express their complexities in the form: for input x the machine M takes τ_x steps until reaching "halt".

Usually we consider "worst case" complexity. Let $L_n := L \cap \Sigma^n$. Then the function

$$t_M\colon \mathbb{N} \longrightarrow \mathbb{N}, \quad t_M(n) := \max\{\tau_x \mid x \in L_n\},$$

is called **(time) complexity** of the TURING machine M (for L).

The subset **P** ("polynomial time") of the set of all functions from L to Σ^* consists of the functions $f\colon L \longrightarrow \Sigma^*$ for which there exists a TURING machine M and an integer $k \in \mathbb{N}$ such that

(i) M computes f,

(ii) $t_M(n) \leq n^k$ for almost all $n \in \mathbb{N}$.

Remark Equivalent with (ii) is the statemant: There is a polynomial $p \in \mathbb{N}[X]$ with $t_M(n) \leq p(n)$ for all $n \in \mathbb{N}$.

For if there is such a polynomial $p = a_r X^r + \cdots + a_0$ (with $a_r \neq 0$), then

$$\begin{aligned} a_r n^r &\geq a_{r-1} n^{r-1} + \cdots + a_0 \quad \text{for } n \geq n_0, \\ p(n) &\leq 2a_r n^r \quad \text{for } n \geq n_0, \\ p(n) &\leq n^{r+1} \quad \text{for } n \geq n_1 = \max\{2a_r, n_0\}. \end{aligned}$$

Conversely if $t_M(n) \leq n^k$ for $n \geq n_0$, then we choose $c \in \mathbb{N}$ with $t_M(n) \leq c$ for the finitely many $n = 0, \ldots, n_0 - 1$. Then $t_M(n) \leq p(n)$ for all $n \in \mathbb{N}$ with $p = X^k + c$.

Analogously we define the set **EXPTIME** ("exponential time"): f is in **EXPTIME** if there exist a TURING machine M, an integer $k \in \mathbb{N}$, and real numbers $a, b \in \mathbb{R}$ with

(i) M computes f,

(ii) $t_M(n) \leq a \cdot 2^{bn^k}$ for almost all $n \in \mathbb{N}$.

Obviously $\mathbf{P} \subseteq \mathbf{EXPTIME}$.

Examples with $\Sigma = \mathbb{F}_2$.

1. Assume

   $$L := \{(p, z) \in \mathbb{N}^2 \mid p \text{ prime } \equiv 3 \pmod 4,\ z \in \mathbb{M}_p^2\}$$

   is coded as a subset of Σ^* by a suitable binary representation. Let $f(p, z) =$ the square root of z mod p, likewise coded as an element of Σ^*. Then $f \in \mathbf{P}$ by 5.3

2. Let $L = \mathbb{N}_2$ be the set of integers ≥ 2 (binary coded). Let $f(x) =$ be the smallest prime factor of x. Then $f \in \mathbf{EXPTIME}$ since we can try all the integers $\leq \sqrt{x} \leq 2^{n/2}$.

   Presumably $f \notin \mathbf{P}$.

3. The **knapsack problem**. Here

   $$L = \{(m, a_1, \ldots, a_m, N) \mid m, a_1, \ldots, a_m, N \in \mathbb{N}\}$$

   with suitable binary encoding,

   $$f(m, a_1, \ldots, a_m, N) = \begin{cases} 1, & \text{if there is } S \subseteq \{1, \ldots, m\} \\ & \text{with } \sum_{i \in S} a_i = N, \\ 0 & \text{otherwise.} \end{cases}$$

   Then $f \in \mathbf{EXPTIME}$ since we can try all of the 2^m subsets $S \subseteq \{1, \ldots, m\}$.

   Presumably $f \notin \mathbf{P}$.

6.6 The Class NP

We say that the TURING machine M computes $f\colon L \longrightarrow \Sigma^*$ **nondeterministically** if for each $x \in L$ there is a $y \in \Sigma^*$ such that M, given the concatenation xy of x and y as input, reaches "halt" after finitely many steps with output $f(x)$.

Example Let $\Sigma = \mathbb{F}_2$ and $L = \{(n, a, x) \in \mathbb{N}^3 \mid n \geq 2, a, x \in \mathbb{M}_n\}$. Let $f = \log_a \bmod n$ be the discrete logarithm.

For a given x let y be the logarithm of x—it doesn't matter in the definition from where we get the logarithm, in any case it exists. All the TURING machine M has to do is to check whether $a^y = x$. Then it writes y to the tape and halts.

General idea A candidate y for the solution is provided, M only does a check.

Alternative idea An unbounded number of *parallel* TURING machines each checks a different $y \in \Sigma^*$.

The set **NP** ("nondeteministic polynomial time") is defined as the set of all functions for which there exists a TURING machine M and an integer $k \in \mathbb{N}$ with:

(i) M computes f nondeterministically,

(ii) $t_M(n) \leq n^k$ for almost all $n \in \mathbb{N}$.

We have the inclusions

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXPTIME}.$$

The first of these is trivial, the second is a theorem that we don't prove here.

The most important unsolved problem of theoretical computer science is the conjecture

$$\mathbf{P} \overset{?}{\neq} \mathbf{NP}.$$

Likewise unproven is the conjecture

$$\mathbf{NP} \overset{?}{\neq} \mathbf{EXPTIME}.$$

On the other hand the statement

$$\mathbf{P} \neq \mathbf{EXPTIME},$$

is proven, if only by constructing "artificial" problems. There is no known "natural" problem proven to be in the difference set.

By the way we cannot make cryptanalysis of a cipher more difficult than **NP**: Exhaustion—that is trying all keys with a known plaintext—is always possible, and the encryption function must be efficient, hence in **P**.

Examples

1. If f is the discrete logarithm as above, then $f \in \mathbf{NP}$.
2. Likewise factoring integers is in **NP**.
3. Also the knapsack problem is in **NP**.

We call the function f **NP-complete** if for each TURING machine M that computes f (deterministically!) and each function $g \in \mathbf{NP}$ there exists a TURING machine N that computes g and an integer $k \in \mathbb{N}$ such that

$$t_N(n) \leq t_M(n)^k \quad \text{for almost all } n \in \mathbb{N}.$$

In other words the complexity of N is at most polynomial in the complexity of M.

Interpretation **NP**-complete problems are the maximally complex ones among those in **NP**.

It is known that **NP**-*complete problems exist.* We refrain from proving this theorem here.

For instance the knapsack problem is **NP**-complete, as is the determination of zeroes of (polynomial) functions $p\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$. Factoring integers is presumably not **NP**-complete.

Should $\mathbf{P} = \mathbf{NP}$ hold—nobody believes it—, then all functions in **NP** would be **NP**-complete. If not, the following drawing illustrates the relative situation of the complexity classes: