A Quick and Dirty Introduction to Cryptology

Ivan Damgård

March 30, 1999

1 Introduction

This note is a brief introduction to some basic concepts in cryptography. We make no attempt whatsoever to be formally correct in defining the concepts, we are only after an inutitive understanding of which concepts we can play around with and what natural categories they fall into.

2 What is Data Security?

Some people think data security was born when we invented computers. This is of course false. The need for secure communication over insecure channels is as old as mankind itself. Data Security is about basic user needs, most importantly

Confidentiality - being able to send data from A to B such no third person can learn anything about what is being sent.

Authenticity - being able to send a message from A to B such that the recipient B can verify that the message really was sent from A, and - in the strongest case - anyone else can also verify this.

Using computers and digital communication rather than paper does not really change these basic needs, but does of course change the technical solutions we use to meet the demands.

3 Crypto- and other Systems

At the most basic level, we have two types of objects:

Cryptosystems Which are basically algorithms for transforming messages into a form that is completely unreadable to anyone, except hopefully the intended receiver.

Authentication Systems Which are algorithms for adding "check values" to messages, such they can be verified by the recipient and any tampering can hopefully be detected.

A cryptosystem basically consists of a set of *plaintexts* $\mathcal{P}$, a set of *ciphertexts* $\mathcal{C}$, and algorithms E, D for encryption resp. decryption. Finally, there is an algorithm G that can efficiently generate pairs of keys k_e, k_d, which are matching pairs of keys for en- and decryption. More precisely, for any message $m \in \mathcal{M}$, the value $E(k_e, m) \in \mathcal{C}$, and it must hold that $m = D(k_d, E(k_e, m))$. Note that in general, the algorithms E, D are allowed to be probabilistic.

In some systems, we always have $k_e = k_d$, they are said to be *symmetric* or *conventional*. In that case, G usually just chooses k_e uniformly from some finite set. In other systems $k_e \neq k_d$, in fact computing k_d from k_e is a computationally difficult problem. Therefore, one can make k_e public while keeping k_d secret and known to only one user. Such systems are called *asymmetric* or *public-key* systems. In this case, G usually gets a security parameter k as input, and goes through a rather complicated process to output k_e, k_d. In this case k measures the size of the computational problem one must solve in order to compute k_d from k_e.

An authentication system consists of a set of *plaintexts* $\mathcal{P}$, a set of *check-values* $\mathcal{A}$, and algorithms A, V for authentication, resp. verification. Finally, there is an algorithm G that can efficiently generate pairs of keys k_a, k_v, which are matching pairs of keys for authentication and verification. More precisely, for any message $m \in \mathcal{M}$, the value $A(k_a, m) \in \mathcal{A}$, and it must hold that $V(k_d, m, A(k_a, m)) = accept$ (whereas if m is replaced by a different m' supposedly the outcome would be $reject$, at least most of the time).

Just as for cryptosystems, authentication systems can be divided into symmetric and asymmetric/public-key systems. A public-key authentication system is also called a *digital signature* system.

4 Security?

What does it mean that a system is secure? Definitions of this fall into two categories:

Unconditional Security Where no amount of computing power will allow you to break the system.

Computational Security Where only attackers with bounded (polynomial) computing power cannot break the system.

Unconditional security is defined using *information theory*, which is basically a tool allowing us to measure (in an absolute way) the amount of information contained in a given message. Given this, security is straightforward to talk about, for instance for cryptosystems, we basically say that the encypted data must contain no or almost no information about the plaintext message.

Computational security often leads to more complicated definitions. Here, we define a game that is played by an *adversary*, which we think of as a polynomial time machine that gets some initial input and then has access to an oracle. The input models the data that any adversary in a real situation must know, such as the public encryption key for a public key system. The oracle models the additional information that we think an attacker in the real world may gain access to, e.g. by eavesdropping or actively attacking honest users. By defining which questions the adversary may ask the oracle, we can define different types of games, also often known as types of *attacks*. Standard examples for cryptosystems are:

Ciphertext only attack The adversary knows a priori a probability distribution with which plaintext messages are chosen, and can ask for a ciphertext c, encrypted with some fixed key (of course, the adversary can ask as many times as he wants, and will not necessarily get the same answer next time).

Known plaintext attack Again some plaintext distribution is given, and the adversary can ask for a pair (m, c), where c is m encrypted with some fixed key.

Chosen plaintext attack The enemy can give a plaintext message m to the oracle and is then given the encryption c under some fixed key.

Chosen ciphertext attack The enemy ciphertext c and is then given the corresponding plaintexts m.

Finally, we must define what it means for the adversary to *win* the game. Typically, we say that adversary wins, if he produces a particular output. For instance, we could say that his goal is to find the secret key used for encryption. But the goal could also be much less ambitious, for example guessing whether a given ciphertext encrypts a given plaintext or not.

A definition of security specifices such a game, and demands that any "efficient" adversary wins the game with only "negligible" probability.

For a formal definition, we must of course make precise what we mean here. One approach is to use concrete bounds and for example let "efficient" mean "doable in less than 2^{80} bit operations" and "negligible" mean "less than 2^{-64}".

While this approach leads to very directly applicable results, it can be very cumbersome to deal with in some cases, in particular in connection with public key systems that lend themselves more easily to a complexity theory inspired treatment. Here, we have a security parameter k which is used as input when setting up the system and measures the difficulty in breaking it. We then say that "efficient" means "probabilistic polynomial time in k" and "negligible" means "as a function of k, converges to 0 faster than any inverse polynomial". We will make this more precise and discuss it more later in the course.

In any case, the goal is of course to build crypto- and authentication systems where, even using the most friendly of oracles, the adversary cannot even reach the least ambitious goal. Put another way, even under sthe strongest attack, the system cannot be broken in the weakest sense.

5 Primitives

How do we build secure systems? This question can be answered in an absolute sense for unconditional security, as we shall see later. For computational security, this is a hard question because a proof of security for, e.g., a public key system would imply that $P \neq NP$, and probably much more than that. The best we can do is to base our systems on assumptions that some given problem(s) is hard to solve.

This means of course that the specific problems we use need to be analysed as good as we know how. However, given state of the art, it is of interest not to rely just on a single problem - which may next year turn out to be easy after all. We would rather base ourselves on more generic, general assumptions, also sometimes known as "primitives".

The most basic such primitive is a *one-way function.* This is simply a function $f : \{0,1\}^k \rightarrow \{0,1\}^k$ such that it is *easy* to compute $f(x)$ from x, but given $y = f(x)$ where x is random, it is *hard* to find any element in $f^{-1}(y)$.

As implied by the above discussion, we don't know if one-way functions exist. However, the importance of this primitive comes from the fact that the existence of one-way functions is a necessary assumption for existence of nearly any non-trivial cryptographic construction with computational security. Hence the ultimate goal in theoretical cryptography is always to be able to build some type of system based on any one-way function, since this is the weakest possible assumption. For instance, we know that digital signature systems with the strongest possible security exist if and only if one-way functions exist.

However, it is not always possible to make do with just one-way functions. For example, public key cryptosystems are not known to follow from any one-way function, and most researchers believe that this is in fact not the case. We therefore deal with several stronger versions: *one-way permutations* (which should be self-explanatory) and *one-way trapdoor permutations*, which are permutations that are normally hard to invert, but given some extra information - the trapdoor - inversion is easy. For example, secure public key encryption follows from any one-way trapdoor permutation.

This type of general result often leads to rather inefficient constructions, because we are not allowed to use any special properties of the functions involved, other than one-wayness, or whatever the definition itself guarantees. Another research direction therefore is to provide practical systems based on specific problems. Even such systems can (and should) be provably secure in the sense that we can argue that breaking our system is "as hard" as solving the problem we base it on. The standard way of doing this is to provide an efficient reduction that transforms an adversary breaking the system into an algorithm solving the problem we assume is hard.

6 Protocols

Even given secure crypto- and authentication systems, we do not automatically have real implementations of solutions ensuring basic data security needs.

A "secure cryptosystem" is an abstract mathematical object, and even given such an animal, this does not tell me how to write a program for sending encrypted files securely from A to B.

First of all, we must ensure that all players in the game have the right key(s) available when they need to encrypt or decrypt something - of course without exposing secret keys to an adversary. Secondly, my cryptosystem may well be defined as a method for en- and decrypting blocks of a fixed length. Now, how do I use this encrypt an arbitrary length file without introducing new weaknessnes into the system?

Solutions to this sort of problem fall into the very large and very fuzzy area called cryptographic protocols. Generally speaking, we talk about protocols as soon as we have two or more players interacting, rather than for instance just an adversary trying to attack a cryptosystem.

The game we play here is similar to the one in the previous section: we start from a set of black boxes implementing certain functionalites securely, and we want to construct protocols with provable security. Again, provable should be understood in a relative sense: what we can prove is that if an adversary could cheat our protocol, then he could break one of the basic black-boxes we started from. And again the standard approach is to provide constructively a reduction showing that the existence of a successful adversary against our protocol implies existence of an adversary breaking the black-box, i.e. the public-key cryptosystem, or whatever.