University of Maryland
CMSC456 — Introduction to Cryptography
Professor Jonathan Katz

Lecture 4

1 More on Definitions of Security

At the end of the last class, we gave the following definition of security:

Definition 1 *An encryption scheme over message space $\mathcal{M}$ is* secure *if, for all $m_1, m_2 \in \mathcal{M}$ and all efficient algorithms A, the following holds:*

$$|\Pr[A(C) = m_1 | C = \mathcal{E}_k(m_1)] - \Pr[A(C) = m_1 | C = \mathcal{E}_k(m_2)]| < \epsilon.$$

This leaves us with two questions: what do we mean by efficient? And, how small should ϵ be?

1.1 Efficient Algorithms

Our notion of efficient will be equated with *probabilistic polynomial time* (PPT). A polynomial time algorithm A is one for which there exists a polynomial $p(\cdot)$ such that the running time of A on input $x \in \{0,1\}^*$ is at most $p(|x|)$. A *probabilistic* algorithm also has the ability to "flip" random coins (or, equivalently, bits) and use the result of these coin tosses in its computation. Of course, if the algorithm is polynomial time, it can flip at most a polynomial number of bits.

The question was raised as to whether it is necessary to consider probabilistic algorithms. In fact, it is a major open question in complexity theory as to whether randomness helps; i.e., whether a probabilistic poly-time algorithm is any more powerful than a poly-time algorithm. Certainly, randomness seems to help us *design* efficient algorithms, and randomness turns out to be essential for secure cryptography. In any event, since we want to protect against the broadest class of adversary possible, and since it seems that allowing the adversary to flip random coins seems reasonable (we do it ourselves...) , we allow PPT algorithms in our definition of security.

Now, another question that needs to be answered for a polynomial-time algorithm is: polynomial in *what*? Consider, for example, the one-time pad encryption scheme. Here, the ciphertext is always the same length (for a fixed message space) so it does not make much sense to talk about asymptotic running times of an algorithm. We can fix this by introducing a *security parameter* which will eventually be a measure of how secure we want the scheme to be. (In the case of the one-time pad, which is perfectly secure, this is irrelevant. But we will see later examples of when it becomes relevant.) We will represent this security parameter by 1^k — a string of k ones. We then require that the adversary run in time polynomial in k. If it helps, for now one can think of the security parameter as being akin to key length — as the key length increases (informally), the scheme gets more secure.

1.2 Negligible Functions

The next question we turn to is how small ϵ should be? Since we don't want to set any *a priori* (constant) bound on the success of an adversary, we prefer instead to talk about asymptotic security and to let the adversary's advantage depend on the security parameter. On the other hand, we want to adversary's advantage to be (asymptotically) very small, so how can we ensure this? We do this by forcing ϵ to be a function of k; furthermore, we will require that $\epsilon(k)$ grow smaller than any inverse polynomial. A typical example of such a function is the inverse exponential function: $\epsilon(k) = 2^{-k}$. Note that $2^{-k} = O(1/k^c)$ for any constant c. Since functions of this sort crop up a lot in cryptography, we will formally define them:

Definition 2 *A function $\epsilon(\cdot)$ is* negligible *if for any $c > 0$ there exists an $N_c > 0$ such that, for all $N > N_c$ we have: $\epsilon(N) < 1/N^c$.*

We can now rephrase Defintion 1 as follows:

Definition 3 *An encryption scheme over message space $\mathcal{M}$ is* secure *if for all $m_1, m_2 \in \mathcal{M}$ and all PPT algorithms A, there exists a negligible function $\epsilon(\cdot)$ such that:*

$$\left|\Pr[A(1^k, C) = m_1 | C = \mathcal{E}_k(m_1)] - \Pr[A(1^k, C) = m_1 | C = \mathcal{E}_k(m_2)]\right| < \epsilon(k).$$

Note that for us to ever be able to satisfy this definition, the key generation algorithm must depend on the security parameter k (why?). In fact, this makes sense: before we can generate a key, we need to know how much security we want to achieve.

2 Computational Hardness

Recall the reason we gave the modified definition was to design more efficient schemes (i.e., scheme in which we do not need to share n bits in order to encrypt an n-bit message). In other words, any scheme we construct will be breakable by an "all-powerful" algorithm (since only the one-time pad is secure against *all* algorithms); we can only hope that the scheme will not be breakable by a PPT algorithm. This suggests some notion of "computational hardness"; we attempt to formalize this notion here.

We choose to formalize the notion via the concept of a *one-way function*. Informally, this is a function which is easy to compute yet hard to invert. A formal definition follows:

Definition 4 *A function $f : \{0,1\}^* \to \{0,1\}^*$ is* one-way *if the following hold:*

- *The function is easy to compute. Namely, there exists a PPT algorithm A which, on input x returns $f(x)$ in time polynomial in $|x|$.*
- *The function is hard to invert. Namely, for all PPT algorithms A there exists a negligible function $\epsilon(\cdot)$ such that:*

$$\Pr[x \leftarrow \{0,1\}^k; y = f(x); x' = A(1^k, y) : f(x') = y] \leq \epsilon(k). \tag{1}$$

A few words are in order about the definition. First, let's look at the notation I used to define hardness of inversion. Everything before the colon (":") represents an experiment; everything after the colon represents the event whose probability we are interested in. So, in words, (1) means the following: we pick a random k-bit string x. We then compute $y = f(x)$. Next, we give y as input to algorithm A which outputs x'. Now, the event we are interested in is whether $f(x') = y$ or not. And we require that the probability of this event be "small".

Note that the probability on the left-hand-side of (1) is a function of k — the length of x. So it makes sense to compare this to the function $\epsilon(k)$.

Note also that we give A the auxiliary input 1^k. This can be viewed as simply telling A what the length of the original input is. But the reason for doing so is more fundamental. Imagine a function that maps every input string to the 1-bit string 0. Note that allowing A to run in polynomial time here doesn't help if we only run $A(y)$ — since y is always going to be 0, A will be forced to always run in a constant number of steps! We avoid such unpleasant scenarios by giving 1^k to A and allowing A to run in time polynomial in $|1^k| = k$.

Finally, note that A is *not* required to output the original value x. As long as A can find any inverse of y we will count that in A's favor. It might very well be the case that f is not one-to-one and hence many different inputs map to the same output.

It is worth giving a simplified definition for the case when f is a permutation (what is meant here is that, when restricting the input of f to $\{0,1\}^n$, then f is in fact a permutation over $\{0,1\}^n$).

Definition 5 *We say f is a* one-way permutation *if the following hold:*

- *f is easy to compute (as above).*
- *Let $x \in \{0,1\}^n$ for arbitrary n. Then $f(x) \in \{0,1\}^n$ and furthermore f (when restricted to $\{0,1\}^n$) is a permutation over $\{0,1\}^n$.*
- *f is hard to invert. Namely, for all PPT algorithms A there exists a negligible function $\epsilon(\cdot)$ such that:*
$$\Pr[x \leftarrow \{0,1\}^k; y = f(x) : A(y) = x] \leq \epsilon(k).$$

Notice that the definition of being "hard to invert" is equivalent to what was given previously; we just simplified it for the case of f being a permutation rather than a function. (Make sure you understand why we can make these simplifications.)

2.1 An Example

Let's look at a simple example before seeing some more complicated examples from number theory. Consider the function $f(x, y) = x \cdot y$ where the domain of f is pairs of integers both strictly greater than 1. Is this a one-way function? Well, note that inverting the function requires factoring, something we all know is "hard". But let's examine whether this qualifies to be one-way. We need to consider the following probability:

$$\Pr[x, y \leftarrow \{0,1\}^k; z = x \cdot y; (x', y') = A(z) : x' \cdot y' = z].$$

Is this negligible for any polynomial-time algorithm A? (Think about it a minute.)

In fact, a very simple algorithm shows that it is *not* one-way. Consider A that works as follows: A checks if its input z is even; if so, it outputs $(2, z/2)$. Otherwise, A gives up. What is the probability that A succeeds in inverting z in the experiment above? (3/4)

However, we can make the following plausible conjecture about factoring: for any PPT algorithm A there exists some polynomial $p(\cdot)$ such that A *fails* to factor z (in the above experiment) at least $1/p(k)$ of the time. It turns out that we can define a notion of a *weak* one-way function and that f given above satisfies the appropriate definition. Furthermore, we state the following beautiful theorem:

Theorem 1 *One-way functions exist iff weak one-way functions exist.*

The theorem is in fact constructive. Thus, if factoring is moderately hard (as captured by the notion of being weak one-way), then we can construct a (strong) one-way function as well.

Next class we will see some simple examples of one-way functions and permutations based on number theory.