University of Maryland
CMSC456 — Introduction to Cryptography
Professor Jonathan Katz

Lecture 15

1 Pseudorandom Functions and Permutations

A PRG was nice, because it allowed us to share random keys of small size and generate shared pseudorandom keys of large size. But even nicer than having shared access to a pseudorandom *string* would be shared access to a pseudorandom *function*. Let's look at what this might mean.

For simplicity, we consider functions with fixed input and output length. Consider a function $F : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^n$ in which the first input to F is called the *key* and the second input to F is simply called the *input*. k is the *key length*, m is the *input length* and n is the *output length* of F. F is called a *keyed function* for the following reason: fixing a particular key $s \in \{0,1\}^k$ defines a function $F_s : \{0,1\}^m \to \{0,1\}^n$ as follows: $F_s(x) = F(s,x)$. So you can view choosing a key at random as choosing a particular function at random from some large set of functions.

Now, informally speaking, our PRGs had the property that their outputs "looked random" when evaluated on a random seed. The analogous property we would like from F is that it should "look like a random function" when a random key is chosen. But what exactly is a random function?

A random function can be viewed in the following way: if you give the function an input $x \in \{0,1\}^m$ that it has not seen before, the function picks a random $y \in \{0,1\}^n$, stores (x,y) for future reference, and outputs y. If you later ask the function an input x which it *has* seen before, it finds the appropriate pair (x,y) and returns y (so the function always reports the same answer if you ask it the same query).

We can also view all these choices as being made in advance, instead of upon receiving a query. Thus, for every possible input $x_i \in \{0,1\}^m$ the function chooses a random $y_i \in \{0,1\}^n$ and stores (x_i, y_i) in some huge table. In fact, if we order the xs lexicographically, it is enough to just store the ordered list $y_1, \ldots, y_n$.

How much space would this actually take if we were to implement this? Well, each y_i requires n bits and there are 2^m of them to store, for a total of $n \cdot 2^m$ bits of storage just to represent a random function. Clearly, it is infeasible to store a truly random function for moderate values of m, n. But we can certainly imagine such a thing existing as an *oracle*. (In fact, you could imagine implementing a random function if the answers were chosen "on-the-fly" instead of in advance. But there would be some bound on the number of different queries you could ask it, depending on the amount of storage that is available. If you want to *share* a random function, however, the answers must be decided upon in advance. And since you don't know what the queries are going to be, this requires sharing the full $n \cdot 2^m$ bits.) We denote the set of all functions from $\{0,1\}^m$ to $\{0,1\}^n$ by $\mathsf{Rand}^{m \to n}$.

We will now define a formal notion of a pseudorandom function (PRF). Recall our informal definition that a PRF "looks like" a random function; more formally, a PPT adversary

cannot tell them apart with too high a probability. In what way? Well, say we give an adversary a black box and tell it that the box implements either a random function or a pseudorandom function[1] and ask it to determine which one. Then the probability that the adversary guesses "pseudorandom" should be the same whether the function is actually pseudorandom or not. More formally, a function $F : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^n$ is a (t, ϵ)-PRF if for all algorithms A running in time at most t the following holds:

$$\left|\Pr[s \leftarrow \{0,1\}^k : A^{F_s(\cdot)} = 1] - \Pr[F \leftarrow \mathsf{Rand}^{m \to n} : A^{F(\cdot)} = 1]\right| \leq \epsilon.$$

(Note that here we have k a fixed value, so the above is not a function of a parameter k as was the case in previous lectures. Similarly, instead of making A run in polynomial time we have instead fixed an upper bound t for the adversary's running time. The intention was to simplify things a little bit!) In words, the left expression consider the experiment in which a random key s is chosen, and A is given oracle access to the function F_s. In the right experiment, a completely random function F from m bits to n bits is chosen, and A is given oracle access to F. We claim that A cannot distinguish between these two cases with probability better than ϵ.

We always implicitly assume that it is "easy" to compute $F_s(x)$, given both s and x. But the above definition says, in particular, that it is "hard" to predict the value of $F_s(x)$ — even when x is known — if s is unknown and randomly chosen.

Although we did not give a formal, complexity-theoretic definition of one-way functions, we hope the reader will see how such a definition would proceed.[2] And we therefore state the following theorem with little justification and no proof.

Theorem 1 *Pseudorandom functions exist if and only if one-way functions exist.*

1.1 PRPs

We can define an analogous notion of a pseudorandom permutation (PRP). Here, we consider functions $P : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^m$ such that, for all $s \in \{0,1\}^k$, we have that P_s is a permutation over $\{0,1\}^m$ (and call functions with this property *keyed permutations*). Also, we define Perm^m as the set of all permutations on $\{0,1\}^m$. (As an exercice: how many bits does it take to represent a completely random permutation? I.e., how many different permutations over $\{0,1\}^m$ are there?) In a manner completely analogous to the above, we say that P is a (t, ϵ)-PRP if it is a keyed permutation such that for all algorithms A running in time at most t we have:

$$\left|\Pr[s \leftarrow \{0,1\}^k : A^{P_s(\cdot)} = 1] - \Pr[P \leftarrow \mathsf{Perm}^m : A^{P(\cdot)} = 1]\right| \leq \epsilon.$$

Finally, we mention that once we have a keyed permutation it is natural to talk about taking the inverse of P_s, for a particular s. Again, in this case we would hope that it be

[1] Technically speaking, the original F is a pseudorandom function but any particular F_s (i.e., once key s is fixed) is not. If you know the value of s, the function no longer "looks random". So instead of saying that the box "implements a pseudorandom function" I should technically say the box "impelments F_s for randomly-chosen s". But I won't always be this careful.

[2] For the very interested reader, we point out that the complexity-theoretic definition can no longer consider functions F with input length; instead, the input length must depend on the key length.

"easy" to compute $P_s^{-1}(y)$, given s and y (note that this is not necessarily the case — as we have seen already in this class, there are plenty of permutations for which the forward direction can be computed easily but the inverse cannot be). But of course, we might hope that it be "difficult" to predict $P_s^{-1}(y)$ when s is unknown. In fact, there are secure PRPs for which it is "easy" to predict the inverse of $P_s^{-1}(y)$, for a particular choice of y and even for a randomly chosen s.

So this requires another definition. We say that P is a *strong* (t, ϵ)-PRP if it is a keyed permutation such that for all algorithms A running in time at most t we have:

$$\left|\Pr[s \leftarrow \{0,1\}^k : A^{P_s(\cdot), P_s^{-1}(\cdot)} = 1] - \Pr[P \leftarrow \mathsf{Perm}^m : A^{P(\cdot), P^{-1}(\cdot)} = 1]\right| \leq \epsilon.$$

Note that (in each experiment) we now give A access to *two* oracles: one representing the forward evaluation of the permutation and one representing the inverse.

As mentioned earlier, not every PRP is automatically a strong PRP. We state this here as a theorem:

Theorem 2 *Assuming the existence of one-way functions, there exists a keyed permutation P which is a PRP but not a strong PRP.*

It is a useful exercise to try to construct an example of such a P.
Finally, we state the following theorem for reference:

Theorem 3 *Strong PRPs exist if and only if one-way functions exist.*

1.2 Block Ciphers

What is done in practice? In practice, people use *block ciphers*. These block ciphers are keyed functions, just as described above. Famous examples of block ciphers include DES, IDEA, triple-DES, and AES. As an example, DES is just a keyed permutation $\mathsf{DES} : \{0,1\}^{56} \times \{0,1\}^{64} \to \{0,1\}^{64}$. That is, the key length of DES is 56 bits, and once a key s is fixed, DES_s is a permutation over $\{0,1\}^{64}$. Furthermore, it is efficient to compute and to invert DES if you know the key.

When block ciphers are used in an application, it is important to ask what properties of the cipher are assumed. For example, does security rely on the fact that the cipher is a PRP or does it require the stronger assumption that the cipher is a strong PRP? On the one hand, it is reasonable to assume that DES or AES is a strong PRP (ciphers are designed with this in mind). On the other hand, this is a stronger assumption, so basing the security of your scheme on a weaker assumption is preferable. Finally, some "off-the-shelf" ciphers may be PRPs but not strong PRPs.

Finally, we mention that the security of block ciphers (as opposed to what we have been aiming for in this class) is largely heuristic. In other words, we have no particular reason to believe that AES is a good block cipher, other than the fact that it was designed by experts and studied intensively by cryptographers for the past 2 years. On the other hand, we could (by Theorem 3) construct a strong PRP based on the hardness of factoring. The hardness of factoring is backed up by 300 years of intense scrutiny by many, many people who were experts in a variety of fields. The **huge** disadvantage of this approach is that the resulting function would be orders of magnitude slower than DES or AES, and therefore not practical for use.