ECS 227: Modern Cryptography
Lecture 6

The Asymptotic / Complexity-Theoretic Approach

Phillip Rogaway

30 January 1996

Reference for this lecture: Sections 1.2 and 1.3 of Oded Goldreich's notes, available from the Web (start at our course home page).

6 The Asymptotic Approach to Cryptography

6.1 Complexity Classes

Let us recall the familiar complexity classes **P**:

> **P** = $\{L$: there exists a polynomial time Turing Machine M such that
>
> $x \in L \Rightarrow M(x)$ accepts, and
>
> $x \notin L \Rightarrow M(x)$ rejects.
>
> $\}$.

When we say that TM M is polynomial time (PT) we mean that there exists some polynomial poly such $\mathrm{TIME}_M(x) \leq \mathrm{poly}(|x|)$, where $\mathrm{TIME}_M(x)$ denotes the number of computational steps that machine M takes on input x.

Often in computer science **P** is taken as the formalization of what is "efficiently computable." We shall be somewhat more generous and give our Turing machines coins, allowing them to error on occasion. We define the complexity class **BPP** as follows:

> **BPP** = $\{L$: there exists a probabilistic polynomial time (PPT) TM M such that:
>
> $x \in L \Rightarrow \Pr[M(x) \text{ accepts}] \geq 2/3$, and
>
> $x \notin L \Rightarrow \Pr[M(x) \text{ accepts }] \leq 1/3$.
>
> $\}$.

When we say that TM M is PPT we mean that there exists some polynomial poly such that $\mathrm{TIME}_M(x, r) \leq \mathrm{poly}(|x|)$, where $\mathrm{TIME}_M(x, r)$ denotes the number of computational steps that M takes on input x and infinite string r. Observe how we think of a probabilistic Turing machine as a deterministic one in which there's an additional input r. Running time measures continue to be worse-case (they don't depend on r) but now there is an induced probability space (for any given x). This probability space is determined by taking every bit of r to be the result of a fair coin toss. When we make probabilistic statements, as above, this this the probablity space which we mean.

BPP stands for bounded (away from 1/2) probabilistic polynomial time. It is the "error probablity' (2/3, 1/3) which is bounded away from 1/2.

We shall take **BPP** as our notion of what languages are efficiently computable. In cryptography, those random coins are often relevant.

There is a large literature concerning the extent to which probabilism helps in computation. We shall not explore this topic very much in this course, but note that coins are not only the basis for secrecy, but many algorithms of cryptographic relevance (e.g., deciding if a number is prime) are only known to be efficiently solvable only if one permits probabilism. So **BPP** is a somewhat better model of what is practically-solvable than **P**.

As usual, the above are *languages* and often we are interested in polynomial-time computable functions, probabilistic probabilistic polynomial-time adversaries with access to an oracle, and so forth, instead. In each case the formal notion of what we are talking about can easily be recovered.

6.2 Negligible Functions

Polynomially-bounded functions grow at a rate which is slow enough that we consider their growth "reasonable." Negligible functions, instead, vanish so quickly that we regard them as insignificant. The usual definition is that they vanish faster than the inverse of any polynomial:

Definition 6.2.1 $\epsilon : \mathsf{N}^+ \to \mathsf{R}^+$ *is* negligible *if* $(\forall c > 0)\ (\exists N_0)\ (\forall n > N_0)$

$$\epsilon(n) < \frac{1}{n^c}.$$

A common ideom is that $\epsilon(n) \leq n^{-c}$ *for all sufficiently large* n. It is understood that the value of "sufficient" may depend on all variables (eg., c) already mentioned.

Examples of negligible functions are 2^{-n} and $n^{-\sqrt{\lg n}}$.

If someone tells you that $\epsilon(n)$ is negligible you know that it *eventually* becomes small. Eventually it's less than .001. Eventually it's less than 2^{-100}. Eventually $\epsilon(n) < 2^{-100} n^{-10}$. But you don't know when "eventually" might be.

One of the reasons that cryptographers like to work with negligible functions is the same "detail-less arithmetic" that makes working with anonymous polynomials such a pleasure. For example, it's easy to see that:

negligible + negligible = negligible, or

negligible · polynomial = negligible.

6.3 Complexity-Theoretic Notion of a Secure PRG

Now that we have our basic language set up, let's recast in the complexity-theoretic framework a definition which we've already described in the "exact" manner. For the moment, let's continue to separate the syntax from the semantics.

Definition 6.3.1 *A* pseudorandom generator *(PRG) is a polynomial-time computable function $g : \{0,1\}^* \to \{0,1\}^*$ for which $|g(s)| = \ell(|s|)$, where $\ell : \mathsf{N} \to \mathsf{N}$, $\ell(n) > n$, is the "stretching factor" of the generator.*

Contrast this to our earlier (syntactic) notion of a PRG. There the function g acted on strings of some particular length. Here g acts on strings of *every* length. You can think of a (complexity-theoretically defined) PRG as a *family* of generators, $\{g_n\}$, the generator g_n being the one that stretches strings of length n. Additionally, we require that this entire family be efficiently computable.

Our notion of security is now the following:

Definition 6.3.2 *PRG g, with stretching factor $\ell(\cdot)$, is said to be* secure *(in the uniform sense) if for all PPT adversaries A,*

$$\begin{aligned}\mathrm{Adv}_A^{g,\$}(n) \;=\; & \Big|\Pr\left[s \leftarrow \{0,1\}^n; y \leftarrow g(s) : A(y) = 1\right] - \\ & \Pr\left[y \leftarrow \{0,1\}^{\ell(n)} : A(y) = 1\right]\Big|\end{aligned}$$

is negligible.

The parenthetical *in the uniform sense* will be explained later in today's lecture.

Notice the universal quantifier over all polynomial-time adversaries. We require that every one of these achieves negligible advantage.

Think about the above definitions and try to understand how it is the same, and how it is different, from the definitions we have given earlier for PRGs.

From now on, when speaking in terms of asymptotic security, we refer to g as a PRG when we mean that g is a *secure* PRG.

6.4 A Second Example — One-Way Functions

Let's do a second one of these complexity-theoretic definitions.

In the first lecture we mentioned one-way functions— maps which are "easy" to compute in the forward direction but hard to compute in the reverse direction. Here is a formal definition.

Definition 6.4.1 *A (strong) (uniform)* one-way function *is a polynomial-time computable* $f : \{0,1\}^* \to \{0,1\}^*$ *for which, for all PPT adversaries A,*

$$\mathrm{Adv}^f_A(n) \quad = \quad \Pr\left[x \leftarrow \{0,1\}^n;\ y \leftarrow f(x);\ x' \leftarrow A(1^k, y) :\ f(x') = y\right] .$$

is negligible.

Adversary A is trying to find an inverse to y. Because f is not necessarily one-to-one, y could have many inverse besides x. We regard A as successful if she outputs *any* x' for which $f(x') = y$.

Observe that the distribution the adversary sees on y-values is what is induced by sampling randomly from the domain and applying f. In general, it makes no sense to speak of the difficulty of inverting a function if one does not describe the distribution on the inputs to the inverting algorithm.

The first argument to A, 1^k, ensures that A really gets polynomial time in k to invert y. This could matter if f is severely length-reducing. In that case being polynomial time in $|y|$ wouldn't be the same as being polynomial time in k.

Once again, take time; stare at the definition until it not only makes sense, but seems completely completely natural to you.

The parenthetical "strongly" in the definition refers to the fact that we are asking the adversary to be unable to invert f on all but a vanishing fraction of the inputs. There is a corresponding notion of f being *weakly* one-way. In that definition we require only that, for every polynomial-time adversary A, there exists a polynomial poly such that, for all sufficiently large n, $\mathrm{Adv}^f_A(n) < 1 - 1/\mathrm{poly}(n)$. In other words, A may invert f often —even most of the time— but there is some significant (inverse polynomial) fraction of the inputs for which the adversary does *not* manage to invert f.

Strongly one-way functions exists if and only if weakly one-way functions exist. Look and Mike Luby's book for a proof.

6.5 Comparing Asymptotic and Exact Security

Let us compare our earlier, "exact" way of talking about security with the above, "asymptotic" way discussing security.

1. Syntactically, the basic objects of interest in the asymptotic approach are always families of functions. Even when we define a primitive as a single function (as we did for PRGs and one-way functions) it is a function with domain $\{0,1\}^*$ which we can therefore think of it as a collection of functions, one for each input length. In contrast, the exact approach need not talk about families of functions (though it sometimes does); here it makes sense to speak of a single function with finite domain and range. Because of this, the exact approach more directly models those primitives which are finite functions (e.g. DES) which have no natural extensions to infinite families.

2. The asymptotic approach has a certain "definiteness'— for example, a PRG g is or is not secure, the notion of "what is secure" being formally defined. This linguistic and conceptual concreteness can be useful. In contrast, the exact approach never actually defines what it means for an object to be "secure" —we only define what it means to have an attack of a certain efficacy.

3. The asymptotic approach is well-suited for understanding the basic theoretical relationships among cryptographic primitives. For example, it is known (by complex reductions of [Hastad, Impagliazzo, Levin and Luby]) that a PRG exists if a one-way function exists; while it is an open question as to whether public key cryptography is possible based on the assumption of a one-way function. So coarse-grained assertions about the existence of a reduction are well-captured in the asymptotic-approach. But the *quality* of reduction is made more apparent in the exact approach. (Note this is, to some extent, an historical accident. You will find some complexity-theoretic treatments which are quite explicit about the efficiency of their underlying reductions. In any case, the efficiency of the reduction can always be culled out of the proof.)

6.6 Non-Uniformity

There is an interesting criticism that one might raise about our (asymptotic) notions of security. The criticism is that we have not given enough power to our adversaries, in the following sense. Suppose we are going to use a PRG g. When we use it, typically we stretch strings of some particular length n. That is, any real usage of an asymptotically-defined cryptographic primitive begins with a choice of the security parameter. What if there is *no* efficient adversary that breaks the scheme across all security parameters— but there *is* an efficient adversary that breaks the scheme for any particular security parameter? Nothing in our definition excludes this possibility, and, if it happened, we would be rather dissatisfied with our scheme.

Here is another way to describe the issue. What if an adversary, who is going to be attacking a scheme with security parameter n, can, by computing a *long* time, compute some magical "hint" (depending on n) which will let the adversary efficiently break all subsequent *instances* of length n. That is, an expensive preprocessing step, based only on the security parameter, leads the adversary to efficiently attack the scheme for that choice of security parameter.

To be concrete: imagine that the NSA, by spending 2^{66} time, just once, can find some efficient program that will efficiently breaks DES. This would be a defect of DES; you shouldn't be able to break DES by spending lots of time; you should be able to break DES_a, for specific keys, by spending lots of time.

Here is a slick way to handle this issue: we'll say that a non-uniform adversary, A, is an adversary with an (infinite) string of "advice," a. We'll sometimes indicate this advice as a subscript, A_a. It makes perfect sense to speak of A_a being polynomial time, PPT, etc.. We'll demand security against all non-uniform PT A's.

Let's consider how the advice string captures the idea of non-uniformity. Let A be a non-uniform PT adversary. Adversary A would like to know hint a_n when the input is of length n. So think of A's advice a as consisting of $a = (a_0, a_1, a_2, \ldots)$. Since A is polynomial-time each a_i need only have some polynomial length (in n), and so A can read a_n in time polynomial in n.

There are several ways to think of non-uniform adversaries. You might prefer to think of A as a family of algorithms, $\{A_n\}$, where the size and running time of A_n is bounded by some particular polynomial poly(n). Or, A can be thought of as a family of polynomial-size Boolean circuits. It is easy to see that many such notions are equivalent.

Almost always, each uniform notion of security has a non-uniform counterpart. Usually it is not too hard to move from one notion to the other, though sometimes the most "elegant" uniform definition might look quite different from the most elegant non-uniform one.

When we're doing reductions, you either start from a uniform complexity assumption and get a uniform conclusion, or you can start from a non-uniform complexity assumption and get a non-uniform conclusion. Starting from a uniform assumption is never enough to give you a non-uniform conclusion.

Proofs from non-uniform assumptions to non-uniform conclusions tend to be simpler than their uniform counterparts. That's because, in the non-uniform case, the the non-uniformity lets you get away with certain non-constructive (or not efficiently-constructive) arguments. We may see examples of this.

6.7 Non-uniformity obviates randomness

Above you may have noticed that we said "non-uniform PT adversaries" and not non-uniform *probabilistic* PT time adversaries. That's because, when you have non-uniformity, it seems that you never need coins. This is a "meta-statement" that has no exact meaning. But we will give a concrete example of this statement below.

Let's define a new complexity class, **P/poly**, to capture those languages which are recognizable by non-uniform polynomial time algorithms.

$$\begin{aligned}
&\mathbf{P/poly} = \{L : \text{there exists a non-uniform PT TM } M_a \text{ such that} \\
&\quad x \in L \Rightarrow M_a(x) \text{ accepts, and} \\
&\quad x \notin L \Rightarrow M_a(x) \text{ rejects.} \\
&\}.
\end{aligned}$$

Note that M_a is deterministic, but it has that infinite advice string a. We will show that $\mathbf{BPP} \subseteq \mathbf{P/poly}$. The name **P/poly** stands for **P** with polynomial-length advice.

First we need a generally useful fact about "error reduction." When we defined **BPP** we had acceptance probabilities of $(2/3, 1/3)$, depending on whether the input w was or was not in the language. Let us now show that these numbers are immaterial, in the sense that, had we wanted a smaller error probablity, such as $(1-2^{-m}, 2^{-m})$, where $m = |w|$, we could have gotten such an error probablity quite easily.

Let M be a **BPP** machine achieving error bounds $(2/3, 1/3)$ for some language L. We want to construct a new **BPP** machine, M', which achieves error bounds of $(1-p(m), p(m))$ for L, where $m = |w|$ is the length of the input. The algorithm for M' is simple:

> Run M $n = n(|w|)$ times on w, using independent random coins on each run. On some of these executions M accepts; on others, M rejects. Let x be the fraction of the time that M accepts. Let machine M' accept if $x \geq 1/2$, let it reject otherwise.

In other words, M' just runs M a bunch of times and takes the majority vote. The value of n shall be determined from the proof which we now give.

We need a standard result from probablity.

Theorem 6.7.1 [**Hoefding Inequality**] *Let* $\mathsf{X}_1, \ldots, \mathsf{X}_n$ *be i.i.d. RVs (independent, identically-distributed random variables) on the interval* $[0,1]$. *Let* $\mu = \mathrm{E}[\mathsf{X}_i]$ *be their (common) mean, and let* $\mathsf{X} = (\mathsf{X}_1 + \cdots + \mathsf{X}_n)/n$ *be the RV which is their mean. Then, for any* $\delta \geq 0$,

$$\Pr\left[|\mathsf{X} - \mu| \geq \delta\right] \leq 2e^{-2\delta^2 n}$$

Bounds of this form, when the variables are Bernoulli trials, are called Chernoff bounds.

To apply the above bound for **BPP** error reduction we let X_i be the RV which corresponds to the outcome of trial number i in the running of M'. The X_i's are i.i.d. because M' used independent coins on each run. RV X corresponds to our algorithm's x. Thus

$$\begin{aligned} \Pr\left[M' \text{ errs} \mid w \in L\right] &\leq \Pr\left[|\mathsf{X} - 2/3| > 1/6\right] \\ &\leq 2e^{-n/18} \end{aligned}$$

The same bound holds for the probablity of M' erring when $w \notin L$. So to ensure that M' errs with probability at most $p(m)$ we want to select n such that $2e^{-n/18} < p(m)$, so select $n = \lceil 18 \ln p(m) + 1 \rceil$, say. Thus within **BPP** we can get an error probablity of 2^{-n^c} for any constant $c > 0$ we desire, as achieving such an error requires just $\Theta(n^c)$ executions of a polynomial-time algorithm.

Now let us prove the theorem we claimed earlier:

Theorem 6.7.2 **BPP** $\subseteq$ **P/poly**.

Proof. The following proof uses the *probabilistic method*— showing that a combinatorial object with a certain property exists by showing that a random object has the property with nonzero probablity. (In this case our use of the probabilistic method amounts to no more than just a way of counting, but it is nice to things of things this way.)

Let $L \in \mathbf{BPP}$. Using the error-reduction technique just given, we may assume that L is accepted by a $\mathbf{BPP}$-machine which errs on strings of length n with probablity at most 2^{-n-1}. We are trying to exhibit a $\mathbf{P/poly}$-machine that accepts L.

Think of M as a deterministic TM on two inputs: the (true) input, x, and a random string, r, of some poly(n) number of coins, r. Fix $n = |x|$. Then define string r as being *x-good* if $M(x,r)$ does the right thing: $M(x,r)$ accepts if $x \in L$, and $M(x,r)$ rejects if $x \notin L$. Define string r as *super-good* if r is x-good for every $x \in \{0,1\}^n$.

Suppose, for every n, you "know" —somehow— a super-good string r_n. Then you can expunge the coins of M: just run M not on random coins, but, when the input is of length n, on r_n, instead. By definition, M does the right thing.

We show that, for every n, there does indeed exist a super-good string r_n.

Choose a *random* string $r \in \{0,1\}^{\mathrm{poly}(n)}$. What is the probablity that r is super-good?

$$\begin{aligned}
\Pr[r \text{ is super-good}] &= 1 - \Pr[r \text{ is not } x\text{-good, for some } x \in \{0,1\}^n] \\
&\geq 1 - \sum_{x \in \{0,1\}^n} \Pr[r \text{ is not } x\text{-good}] \\
&\geq 1 - 2^n 2^{-n-1} \\
&\geq 1/2.
\end{aligned}$$

The second line is just that $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$. Now we have shown that the probablity that r is super-good is > 0, so clearly there must exist some super-good string. $\diamondsuit$

Note that the above proof is non-constructive; we don't know how to find a super-good string efficiently, nor how to test if a given string is super-good.