Lecture 26: Probability in Cryptography **Math 20 Fall 2014, Dartmouth College**

Contents

1 Public Key Cryptography

How do you establish a cryptographic protocol on the internet? It's usually impossible to meet in person and agree on an encryption system, and you'd like to set it up so that anyone can communicate with you in a secure way. Wouldn't it be great if you could publish a bucket of padlocks that anyone can take and lock their message with, and only you have a key to open them? A procedure like this is called *public key cryptography*. There is a public way to encrypt a message so that only you can decrypt them. The way it's done, is that there are functions out there called *trapdoor functions*. It means they only go one way, like a trapdoor - they are easy for a computer to do but would take a very long time to reverse. One such function is multiplication - it takes a long time to factor integers, compared to how long it takes to multiply them.

Amazingly, some trapdoor functions make a full circle, come back to the value you started from if you keep applying the function. But you need to know exactly how many times to apply it to get the initial value back. So you tell people how far they should go, and that's your *public key*, the padlock you put on your website that anyone can use to encrypt a message for you. But only you know how far more it takes to get the message back. Here's an example of how this might work.

Theorem 1 (Fermat's Little Theorem) *If p is a prime number, then for any integer a:*

$$a^p \equiv a \pmod{p}.$$

In words, this means that the remainder of a^p when dividing by p is a. It's enough to test it for $a < p$, if you think about the properties of remainders, you can convince yourself that the general case will follow. Try if out for $p = 7$ and $a = 0, 1, 2, 3, 4, 5, 6$.

The statement can be modified to work for a product of two primes pq, as follows:

$$a^{(p-1)(q-1)+1} \equiv a \pmod{pq}.$$

Try it out for $p = 5, q = 7$. Then $pq = 35$, $(p-1)(q-1)+1 = 25$. So we're saying that for any integer a,

$$a^{25} \equiv a \pmod{35}.$$

To use this in public key cryptography, you might announce your public key 5 and base 35. Your *private key* is 5, so that if you raise something that was raised to power 5 again to power 5, it's equivalent to raising it to power 25 in the first place. Then whoever sends you a message, will raise it to power 5 and take the remainder modulo 35. Then you will use your private key, to recover the message by raising what you received to power 5 and taking the remainder mod 35 again. Exponentiating and taking remainders is an easy task for a computer, it can be done in little computational time. But going the other way, finding out what integer a was used to get the encrypted message is difficult. Of course, in real life the primes used in this procedure are very, very big and generated by the computer.

This procedure based on Fermat's Little Theorem is called RSA. It ruled the cypherspace for many years, but by now, it is being gradually abandoned in favor of another type of trapdoor function-based protocol, *elliptic curve cryptography*.

Using strong cryptography for civilian purposes was banned for a long time, it was considered a weapon. But from 1995 to 98 a Berkley math graduate student, DJ Bernstein, fought a court case to be able to publish a paper that included a strong encryption algorithm. The court eventually ruled that software source code should be protected by the First Amendment and the government restrictions on the use of cryptography are unconstitutional. Now, without the use of strong cryptography we wouldn't have the necessary tools for secure online banking or keeping our communications private.

2 Checksums and the Birthday Problem

If you transmit a bunch of encrypted data, how do you check if it didn't get corrupted on the way? You could do a simple calculation, and transmit the result along with the data. Then whoever receives it, can do the calculation on the data again and if they get the same result, they will know that the data is not corrupted. Or will they?

The result of that calculation is called a *checksum* and is often used to verify that data was transmitted correctly. Depending on the nature of the calculation, it may be very hard for an attacker to construct another stream of encrypted data that results in the same checksum. However, there is a chance that in a collection of different encrypted texts there will be two distinct ones that have the same checksum. Suppose that the checksum is 8-bit long and roughly uniformly distributed over the space of 8-bit strings. Then there are $2^8 = 256$ possible checksums and according to the Birthday Problem, there is a better than 50% chance that in a set of 19 different data streams two will have the same checksum.

3 Pseudo Random Number Generators

Computers can't really generate random numbers. They instead try to generate sequences of numbers with properties that approximate a random sequence, but are not really random. If they approximate it well, a sequence of such numbers is called pseudo-random. Think about digits of *pi*. For all we know, it might be true that any digit appears about one tenth of the time, any two-digit number appears about one in a hundred times and so on. It's an open problem to decide if that's true. If it is, can we use digits of π in place of random numbers? Well, it all depends on whether we expect that the numbers can be predicted. If the test for randomness knows to check whether they are digits of π, it will learn to predict them for sure. If not, it probably won't. So in a way, randomness is in the eye of the beholder.

Sometimes, a PRNG uses a random *seed*, a smaller amount of bits that are assumed to be random, to extrapolate a much larger pseudo-random number. Have you ever tried to generate a large collection of random bits on your machine? It may have asked you to move your mouse on make some keystrokes on the keyboard. Then it takes that human-generated input and turns it into a number. A generator may use this seed and perform a calculation whose result may land anywhere in some larger space. There are services that sell random bits, they tend to use radioactive decay or atmospheric noise - that are believed to be truly random.

The *random* package in Python uses a pseudo random number generator called Mersenne Twister.

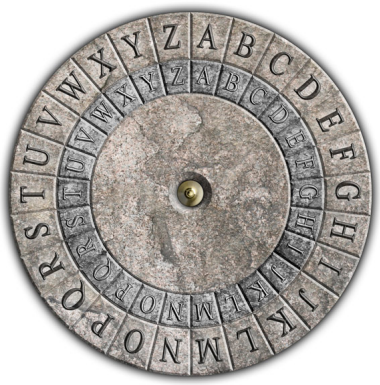
PRNGs made the headlines last year, when it was revealed that the NSA installed a backdoor in a generator used in much commercial software. Long story short, instead of using the random seed to reliably generate a wide spectrum of numbers, the calculation it performed on the seed was carefully tailored to make encryption easy to break. Under normal circumstances, if the parameters were not carefully engineered, finding the right relations between parameters to break the encryption would be next to impossible. Here's how cryptographer Matthew Green relayed the story on his blog:

> For those not following the story, Dual_EC is a pseudorandom number generator proposed by NIST for international use back in 2006. Just a few months later, Shumow and Ferguson made cryptographic history by pointing out that there might be an NSA backdoor in the algorithm. This possibility - fairly remarkable for an algorithm of this type - looked bad and smelled worse. If true, it spelled almost certain doom for anyone relying on Dual_EC to keep their system safe from spying eyes. Now I should point out that much of this is ancient history. What is news today is the recent leak of classified documents that points a very emphatic finger towards Dual_EC, or rather, to an unnamed '2006 NIST standard'. The evidence that Dual_EC is this standard has now become so hard to ignore that NIST recently took the unprecedented step of warning implementers to avoid it altogether.[1]

Dual_EC stands for "dual elliptic curve," it's one of many standards based on elliptic curves.

4 Code breaking by Metropolis-Hastings

The wheel pictured on the right was an invention attributed to Julius Caesar, and assists in encoding and decoding the *Ceasar cipher*. It simply replaces each letter with a letter k spots further in the alphabet, for some k of your choice, and by turning the inside wheel you get a helpful tool that lets you check which letter corresponds to which. This is the simplest of *substitution ciphers*, a class of ciphers that replace a letter by another (for example A by N, B by Q, C by E etc.) But not all of them preserve the order, as the Caesar cipher does. A



[1] http://blog.cryptographyengineering.com/2013/09/the-many-flaws-of-dualecdrbg.html

modern, much more sophisticated descendant of the Caesar wheel was the Enigma machine, and while the Caesar wheel had only 26 settings, Enigma had 158,962,555,217,826,360,000 (158 quintillion) of them. And that's still less than one in two million of all the possible permutations of the letters of the alphabet.

what is the best way to break a substitution cypher? If the text is long, statistical analysis. By Law of Large Numbers, if a letter appears a given percentage of the time in a given language, it with high probability approach that proportion in any text that is sufficiently long. What happens if the text is short? Checking one permutation per nanosecond would take 12 billion years. But if the text is short, there are also relatively few possibilities as to what the original, unencrypted text could have been. An in that case, any substitution cipher, from Caesar to Enigma, can be broken using our recent acquaintance, the Metropolis-Hastings algorithm.

Let g be a permutation of the alphabet, and s_g be the stationary distribution probability associated to g. Consider a Markov chain on the space of permutations, defined by: pick two entries in permutation g and swap them. What are the transition probabilities in this chain? What is the stationary distribution?

If two permutations cannot be achieved from one another by swapping two entries, then the transition probabilities are 0. Otherwise, they're each $1/\binom{26}{2}$. Since the transition matrix is symmetric, the stationary distribution is uniform. Since the transition matrix of the original chain is symmetric, the probabilities a_{ij} in the Metropolis-Hastings procedure are:

$$a_{ij} = \min(1, \frac{s_j P_{ji}}{s_i P_{ij}}) = \min(1, \frac{s_j}{s_i})$$