Math 189H Joy of Numbers Activity Log

Tuesday, November 22, 2011

A. S. Besicovitch: "A mathematician's reputation rests on the number of bad proofs he/she has given - pioneering work is ugly."

Alfred North Whitehead: "Everything of importance has been said before by somebody who did not discover it."

The Mersenne prime $M_{216091} = 2^{216091} - 1$, with 65050 digits, was discovered in 1985 by Chevron Geosciences.

We started by watching some of these computations for encrypting and decrypting a la RSA take place in real time, using Maple. Most descriptions of cryptosystems talk about their 'key' sizes - the chosen parameters built into the system - in bits, since most data is stored in binary. [Additions, multiplications, and exponentiations are usually carried out in binary, too, which is partly why fast exponentiation is implemented using squarings.] You can have 2^n n-bit numbers, which using our yardstick of $2^{10} \approx 10^3$ gives us an estimate of the decimal size of the numbers involved. The usual parameters one sees these days are RSA-512 and RSA-1024 (less often, RSA-2048), meaning, for RSA-1024 for example, that 1024-bit moduli [that is, 300-digit decimal numbers] are used. A 300-digit encryption/decription exponent is also usually used. There are some applications, I have heard, where the specific encryption exponent $e = 2^{16} + 1$ is used; this requires 5 multiplications using the squaring method for encryption, putting all of the work of the system on the decryption side (presumably on a central and more powerful computer system). [Although perhaps it is $e = 2^{17} + 1$ that is used?]

Even using 300-digit or even 600-digit moduli and exponents (meaning we chose a pair of primes with 150 or 300 digits each), the fast exponentiation process we learned is very fast; encryption and decryption each took under a second on the instructor's not really very fast machine. The security of RSA, we learned, lies on how difficult it is to factor our modulus $n = pq$ into the two primes we chose (and then very quickly forgot!, once we had computed our decryption exponent d). The best procedures that we currently know, using the 'General Number Field Seive' (or GNFS), require computations of the order of $\exp(1.9[\log(\mathrm{n})]^{1/3}[\log(\log(\mathrm{n}))]^{2/3})$ operations to factor the number n. Conservative thumbnail estimates, for a computer capable of one billion instructions per second (GIPS), are that a 512-bit number (i.e., 150 digits) would require about 30 years; a 1024-bit number would require 300,000,000 years. The process is highly distributable, however, so it can be spread across many (thousands? millions?) computers. In 2009, a 232-digit 'challenge' number, named RSA-768 [not to be confused with the cryptosystem of the same name!], was factored by a team of researchers using hundreds of machines over a period of 2 years.

One innovation that the implementation we used had was something that is called 'salting' (or a 'nonce'?), designed to thwart a standard 'attack' on many cryptosystems. The basic idea is that, in the end, the number of different messages one might send is really not that large. And if we assume that Eve the eavesdropper knows your encryption method, and simply makes a guess that your message is 'Meet me at 6:00 under the clock tower.', for example, she can encrypt that message a' the same as you could and simply compare her encrypted message $(a')^e \underset{n}{\equiv} b'$ to yours; if they match, then she has discovered your message, since $a^e \underset{n}{\equiv} b = b'$ means that $a \underset{n}{\equiv} b^d \underset{n}{\equiv} (b')^d \underset{n}{\equiv} (a')^{ed} \underset{n}{\equiv} a'$, so she has verified that her guess was correct. All without ever knowing d (!). This line of reasoning leads to what is usually known as a *dictionary attack*; take all of the messages $a_1, \dots, a_r$ that you reasonably expect to see used, and encrypt all of them; $b_i \underset{n}{\equiv} a_i^e$. [This, it is assumed, may take a very long time, and require a very large amount of storage!, but it is work you are doing long before you try to break a message.] Now, when you see an encrypted message b sent, you compare it to all of your encrypted messages b_i (which you have probably stored in a order you can more quickly search); if you find a match $b = b_i$, then you know that the message sent was a_i (!). The standard method to defeat this attack is to simply make the number of messages 'reasonably' sent unreasonably large, by adding 'padding' at the end of each message that is just a completely random collection of symbols. Even if Eve is quite certain what the message is, without the correct random padding a dictionary attack approach will never work.

The RSA cryptosystem as we have described it is designed to send a message that nobody but the recipient can read. To do so the sender is using the receiver's public modulus and encryption exponent. But in many cases, like an order to transfer money from your bank account to some other destination, we not only want the message to be sent secretly, but we want the bank to be sure that the order comes from you! Modular exponentiation can do this, provide that the sender also has their own set of moduli and exponents. Let's say Alice is writing to Bob, and Alice has public numbers n_A, e_A and private exponent d_A, and Bob has the corresponding numbers n_B, e_B and d_B. Alice wants to encrypt the message a to send to Bob, so needs to use $b = a^{e_B} \bmod n_B$ for the message. But anybody could do that, since Alice is using publicly available information. But only Alice knows what d_A is. If Alice takes the intended message, b, and computes $c = b^{d_A} \bmod n_A$, this is something only she could do. If Alice now sends Bob the encrypted message c, instead of b, then Bob, if he and Alice have agreed that this is what Alice would do, can recover the intended message a by, first, computing $b = c^{e_A} \bmod n_A$ using Alice's public information, and then computing $a = b^{d_B} \bmod n_B$ using his own secret information. Since only Alice could have carried out the (to her) second step, using her private exponent d_A, the fact that an intelligible message a results from this process that Bob has carried out essentially guarantees that the message was sent from Alice! To recap, Alice sends a message to Bob by using Bob's public exponent and modulus (so that only Bob can read the message) followed by her private key and public modulus, so that only she could have created it. Bob then strips off Alice's private exponent using her public one, and then strips off his own public exponent using his private one. [The two operations, Bob's public and Alice's private, could have been done in the opposite order (we in fact described it that way instead, in class); can you see any advantages or disadvantages to doing it one way or the other?]

Bob now is certain that the message he received could only have come from Alice. Or at least, someone that knows the credentials that Alice has reported to be in possession of (i.e., the secret exponent d_A that goes with her public information n_A, e_A). The fact is, though, that in order to carry out this transaction Alice and Bob must trade public moduli and exponents. This makes their communication, and most any 'public key' encryption scheme like this, susceptible to what is known as a *Man in the Middle* attack. If someone (usually called Malcolm, for malicious attacker) controls the line of communication between Alice and Bob, so that he can intercept messages and insert his own, then he can capture the public keys that each are passing to one another and insert his own. He can pass his own 'fake' public keys n'_A, e'_A (having computed d'_A) to Bob, pretending to be Alice, and the public keys n'_B, e'_B (having computed d'_B) to Alice, pretending to be Bob. Then when Alice and Bob pass 'encrypted' messages (through Malcolm!), Malcolm can read every message, and then re-encrypt it using the keys he has provided. He uses Alice's public e and Bob's 'fake' d to read the message from Alice, then uses Alice's 'fake' d and Bob's real e to encode the message and send it to Bob. (He can do the same sort of thing for messages from Bob.) Alice and Bob will never notice a thing.

The solution to this problem is for Alice and Bob not to get the public keys from one another. Instead they get them from Trent (the usual name for a third-party *Trusted Authority*), whose identity has already been verified by Alice and Bob. In the case of your web browser, Trent's credentials have essentially been pre-programmed into the browser, using 'certificates' which contain these credentials. Names like Thawte and Verisign might be familiar to you in this regard. [In practice, when you do business with a company, you verify that you are talking to who you think you are through a trusted authority, and the company knows they are talking to you because you have provided (using their public key) your password, which surely only you know... The kind of mechanism described above can then be used to provide continuity to an ongoing conversation, without the need to repeatedly provide your password.] Or, as one of you pointed out, you can pass the keys to one another in person! The key (no pun intended...), in the end, is to have some way to know that the person you are talking to is really the person you think he/she is. In the anonyomous world of the internet, the trusted authority is usually the only viable mechanism to carry this out on the scale that communications and commerce require!