Institut for Matematik og Datalogi
Syddansk Universitet

October 11, 2010
JFB

Introduction to Computer Science
E10 – Lecture 12

Lecture, October 11, 8:15–10, U37

We covered the subsection on "Software Verification" in section 5.6. Then, we began on chapter 12 in the textbook, covering up the the definition of the language Bare Bones, ending on page 583, but not doing any examples.

Lecture, November 8, 8:15–10, U37

We will continue with chapter 12 in the textbook, giving an example using Bare Bones and then concentrating on security (also from sections 3.5 and 4.5) and cryptography.

Lecture, November 11, 14:15–16, U71

We will finish chapter 12.

Supplementary notes on RSA

The textbook leaves out many important details regarding the implementation of RSA. (Most of this was, however, covered in your Mathematical Tools for Computer Science course.) For example, the textbook gives the incorrect impression that in computing $m^e \pmod{n}$ that one would first compute m^e and then reduce modulo n. This is not what occurs in practice since it is infeasible for the large numbers used. The intermediate result would have about $e \log(m)$ bits, which would usually be more than 2^{500} bits (either m^e or c^d would be extremely long)! Thus, one computes this using intermediate computations and reducing modulo n after each step. This works because of the following:

Lemma. For all nonnegative integers a, b and any integer $n > 1$,
$a \cdot b \pmod{n} = (a \pmod{n})(b \pmod{n}) \pmod{n}$.

Note that this can be proven using the fact that $a = x \pmod{n}$ if and only if $0 \le a < n$ and there is an integer k such that $a = x + k \cdot n$.

The powers can be computed efficiently using the following algorithm:

```
function power(a,exp,n)

# Compute a^exp (mod n) for nonnegative exp

   if exp = 0 then return(1)
   else if (exp is odd) then
           return((a*power(a,exp-1,n)) mod n)
        else
           c <- (power(a,exp/2,n))
           return((c * c) mod n)
```

The values e and d are *multiplicative inverses* of each other modulo $(p-1)(q-1)$ (i.e. $e \cdot d \pmod{(p-1)(q-1)} = 1$). They can be computed by using the Extended Euclidean Algorithm, which computes greatest common divisors.

Def. $\gcd(a, b)$ = greatest common divisor of a and b = largest $d \in \mathbb{Z}$ (the integers) such that $d|a$ and $d|b$

If $\gcd(a, b) = 1$, then a and b are *relatively prime*.

Thm. $a, b \in \mathbb{N}$ (nonnegative integers). There exist $s, t \in \mathbb{Z}$ such that $sa + tb = \gcd(a, b)$.

Claim: The integers $d = gcd(a, b)$, s and t can be found efficiently, using the Extended Euclidean Algorithm.

For RSA, the value e is chosen so that $\gcd(e, (p-1)(q-1)) = 1$. To find d, we also need a value k such that $e \cdot d = 1 + k(p-1)(q-1)$. Thus, we can compute d by solving for s in the equation $se + t(p-1)(q-1) = 1$. This can be done using the Extended Euclidean algorithm since $gcd(e, (p-1)(q-1)) = 1$.

Note that Jacob Allerelli's notes on modular arithmetic are available through the course home page.

Discussion section: November 15, 14:15–16, Terminal Room

Discuss the following problems in groups of two or three.

First, you will be using the program `gpg` to try encryption. Usage information can be obtained by typing `gpg -h | more` (hitting the space bar will get the rest of it; the vertical line says pipe the output through the next program, and `more` shows a page at a time).

1. Create a public and private key using `gpg --gen-key`. You should choose DSA and El Gamal, and size 2048. Go to the directory `.gnupg` using `cd .gnupg`. List what is in the directory using `ls -al`. Try the commands `gpg --list-keys` and `gpg --fingerprint` to list the keys you have, with the fingerprints, which make it easier for you to check that you have the correct key from someone. How would you use fingerprints?

2. You can save your public key in a file in a form that can be seen on a screen using `gpg --export -a Your Name >filename`. You are "exporting" your key and specifying where the output should go. Then look at it using `more filename`; the `-a` made it possible to see it reasonably on your screen, since it changes it to ASCII.

3. Mail this file to someone else. (Either in another group or within your own group.)

4. Try to figure out how to use `gpg` to "import" the public key you got from someone else. Check the fingerprint.

5. Create a little file and encrypt it. You can use `gpg -sea filename`. What does this do?

6. Mail your file to whoever has your public key. Read their file using the command `gpg -d inputfile >outputfile`. Then look at the output file you created.

7. You can also encrypt a file for your own use using a symmetric key system protected by a pass phrase. Try using `gpg --force-mdc -ca filename`. Then try decrypting as with the file you decrypted previously. Why might you want to do this?

8. Problems 47 and 50 on page 167.
9. Question 4 on page 168.
10. Problem 48 on page 217.
11. Question 11 on page 219.