Thomas J. Watson School of Engineering & Applied Science
Department of Electrical & Computer Engineering

Cryptography & Information Security (EECE560)

Assignment-IV

Solutions

Submitted By

Alem Haddush Fitwi
afitwi1@binghamton.edu
Graduate Student

Submitted To

Dr. Scott Craver
Associate Professor
scraver@binghamton.edu

Submission Date

16 December 2017

September 16, 2017
Binghamton, New York

1) Explanation of the three lines the ***millerRabin*** procedure provided in the handout:

```
#--------------------------------------------------------------------------
set MAXTEST 20
proc millerRabin n {
        regexp {^(.*1)(0*)$} [tobinary ($n-1)] -> first last #Line-1
        set d [todecimal $first]                             #Line-2
        set k [string length $last]                          #Line-3
        for {set i 0} {$i<$::MAXTEST} {incr i} {
                if {[millerRabinTrial $n $d $k]==0} {return 0}
                }
        return 1;
}
#--------------------------------------------------------------------------
```

<table>
<tr><td colspan="2">One of the steps in Miller Rabin Primality Test Algorithm is to rewrite the number to be tested, n, as
$$S = n - 1 = 2^k d$$
$$d = decimal\ form\ of\ the\ bits\ in\ variable\ \text{first}, and$$
$$k = length(last)$$
Brief explanation and screen shot of the sample test for n=101, and n=100 are hereunder provided</td></tr>
<tr><td>Line-1</td><td>➢ By calling the proc “tobinary”, it first converts the decimal value of (n-1) to binary corresponding values. Then, this line of code makes matching as ensues
➢ If the least significant bit value of the number is ‘1’, the whole string of bits will be placed onto the ‘first’ variable; otherwise, if the significant bit is ‘0’, it will be removed from the number and stored on the “last” variable. It will keep on moving the “0” bit values to variable “last” until it encounters a bit value of ‘1’. Only “0” bit values where there is no bit value of “1” on their right side are moved!
➢ This is a smart way of doing division by 2 if the number is even. Even numbers end with “0” bit value, whereas odd numbers with a bit value of “1”. Removing a value “0” from the most significant side of an even number represented in bits is the same as division by two!
➢ In other words, this line of code indirectly divides the number (n-1) by 2 until the biggest odd number (d), which is not divisible by two is obtained. Then, the value stored on variable “first” is the odd (or d) part, and the value stored on variable “last” is the zeros, the number of times 2 should be raised.
$$first\ stores\ odd\ part\ of\ n-1\ in\ binary\ values, and$$
$$last\ stores\ zeroes\ removed\ from\ the\ least\ significant\ side\ of\ (n-1) if\ any\ exists$$</td></tr>
<tr><td>Line-2</td><td>➢ Line-2 simply converts the binary value (whose least significant bit is “1”) stored on variable first by code line-1, which is the odd part of (n-1), into decimal value using the “todecimal” proc and stores it on variable “d”. It is an odd number!</td></tr>
<tr><td>Line-3</td><td>➢ Line-3 computes the length or number of zeros stored in variable “last”, and copies it to variable k. The number of zeroes tells us how many times n-1 can be divided by 2! If n is even, then n-1 will be odd, and it will be entirely placed in variable first. There will be nothing in variable last. That is, $n - 1 = 2^0 d$</td></tr>
</table>

```
#------------------------------------------------------------------------------
set MAXTEST 20
proc millerRabin n {
        regexp {^(.*1)(0*)$} [tobinary ($n-1)] -> first last
        set d [todecimal $first]
        puts "The value in var first is {$first}"
        puts "The value in var d is {$d}"
#------------------------------------------------------------------------------
        set k [string length $last]
        puts "The value in var last is {$last}"
        puts "The value in var k is {$k}"
#------------------------------------------------------------------------------
        for {set i 0} {$i<$::MAXTEST} {incr i} {
                if {[millerRabinTrial $n $d $k]==0} {return 0}
                }
        return 1;
}
#------------------------------------------------------------------------------
% %
% set n 101
puts "The miller rabin primalty test result for {$n} is {[millerRabin $n]}"
%
The value in var first is {11001}
The value in var d is {25}
The value in var last is {00}
The value in var k is {2}
The miller rabin primalty test result for {101} is {1}
% #----------------------------------------------------------------------------
#Set the value of n to 100
set n 100
puts "The miller rabin primalty test result for {$n} is {[millerRabin $n]}"
%
The value in var first is {1100011}
The value in var d is {99}
The value in var last is {}   Because n-1 is odd, nothing is stored on var last
The value in var k is {0}
The miller rabin primalty test result for {100} is {0}
%
```

2) Performance graph of the RSA Key generation. The average times (average of ten trials for a key) for the nine RSA keys of sizes ranging from 400 bits to 2000 bits with a step-size of 200 were collected using the time command.

Table 2.1: Times collected using the time command while generating RSA keys

Key Size in Bits	Time cmd output in µs	Rounded off Time in Sec
400	6534323.7	6.53
600	13687333.5	13.69
800	20075627.6	20.08
1000	53466029.7	53.47
1200	109275816.5	109.28
1400	175388673.9	175.39
1600	272744956.4	272.74
1800	450104677.8	450.1
2000	457109351.3	457.12

Hereunder is provided the specifications of my laptop, using which I generated the RSA keys, as generated by the "lshw –short" linux command.

```
alem@alem-Satellite-S40-A:~$ sudo -i
[sudo] password for alem:
root@alem-Satellite-S40-A:~# lshw -short
H/W path         Device      Class          Description
======================================================
                             system         Satellite S40-A (PSKHGQ)
/0                           bus            VFKTA
/0/0                         memory         64KiB BIOS
/0/23                        memory         512KiB L2 cache
/0/24                        memory         128KiB L1 cache
/0/25                        memory         3MiB L3 cache
/0/26                        memory         4GiB System Memory
/0/26/0                      memory         DIMM [empty]
/0/26/1                      memory         DIMM [empty]
/0/26/2                      memory         4GiB SODIMM DDR3 Synchronous 1600 MHz (0.6 ns)
/0/26/3                      memory         DIMM [empty]
/0/27                        processor      Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz
/0/100                       bridge         3rd Gen Core processor DRAM Controller
/0/100/1                     bridge         Xeon E3-1200 v2/3rd Gen Core processor PCI Express Root Port
/0/100/1/0                   display        GK208M [GeForce GT 740M]
/0/100/2                     display        3rd Gen Core processor Graphics Controller
/0/100/14                    bus            7 Series/C210 Series Chipset Family USB xHCI Host Controller
/0/100/16                    communication  7 Series/C210 Series Chipset Family MEI Controller #1
/0/100/1a                    bus            7 Series/C210 Series Chipset Family USB Enhanced Host Controller #2
/0/100/1b                    multimedia     7 Series/C210 Series Chipset Family High Definition Audio Controller
/0/100/1c                    bridge         7 Series/C210 Series Chipset Family PCI Express Root Port 1
/0/100/1c/0      eth0        network        RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller
/0/100/1c.1                  bridge         7 Series/C210 Series Chipset Family PCI Express Root Port 2
/0/100/1c.1/0    wlan0       network        QCA9565 / AR9565 Wireless Network Adapter
/0/100/1d                    bus            7 Series/C210 Series Chipset Family USB Enhanced Host Controller #1
/0/100/1f                    bridge         HM76 Express Chipset LPC Controller
/0/100/1f.2                  storage        7 Series Chipset Family 6-port SATA Controller [AHCI mode]
/0/100/1f.3                  bus            7 Series/C210 Series Chipset Family SMBus Controller
/0/1             scsi0       storage
/0/1/0.0.0       /dev/sda    disk           500GB HGST HTS545050A7
/0/1/0.0.0/1     /dev/sda1   volume         100MiB Windows NTFS volume
/0/1/0.0.0/2     /dev/sda2   volume         75GiB Windows NTFS volume
/0/1/0.0.0/3     /dev/sda3   volume         120GiB Extended partition
/0/1/0.0.0/3/5   /dev/sda5   volume         92GiB HPFS/NTFS partition
/0/1/0.0.0/3/6   /dev/sda6   volume         24GiB Linux filesystem partition
/0/1/0.0.0/3/7   /dev/sda7   volume         3984MiB Linux swap / Solaris partition
/0/1/0.0.0/4     /dev/sda4   volume         270GiB Windows NTFS volume
/0/2             scsi2       storage
/0/2/0.0.0       /dev/cdrom  disk           CDDVDW SU-208BB
/1                           power          To Be Filled By O.E.M.
root@alem-Satellite-S40-A:~#
```

To generate the performance graph of the RSA key generation, I have used a piece of Matlab program provided below:

```
%@uthor: Alem Haddush Fitwi, BU-SUNY
% EECE 560 Assignment-IV
%Performance Analysis of RSA Key Generation
%---------------------------------------------------------------------------
% RSA Key Sizes in bits
RSA_Key_Sizes=[400 600 800 1000 1200 1400 1600 1800 2000];
% RSA Key Generation Times in seconds
RSAKeyGenTimes=[6.53 13.69 20.08 53.47 109.28 175.39 272.74 450.10 457.12];
%---------------------------------------------------------------------------
% Plotting the performance
plot(RSA_Key_Sizes,RSAKeyGenTimes)
title('RSA Key Generation Performance as size grows from 400 to 2000bits')
xlabel('RSA Key Sizes in Bits')
ylabel('RSA Key Generation Times in seconds')
%---------------------------------------------------------------------------
%----------------------End of Program--------------------------------------
```

Hereunder is the performance graph of the RSA key generation. My laptop worked fine while generation RSA keys whose sizes ranging from 400 bits to 1600bits. Above 1600 bits, my computer started to behave abnormally. It stacked for two days. Then, again I tried to record the times for 1800 and 200bits, and eventually succeeded! But it was terrible! These two key sizes messed up my laptop!

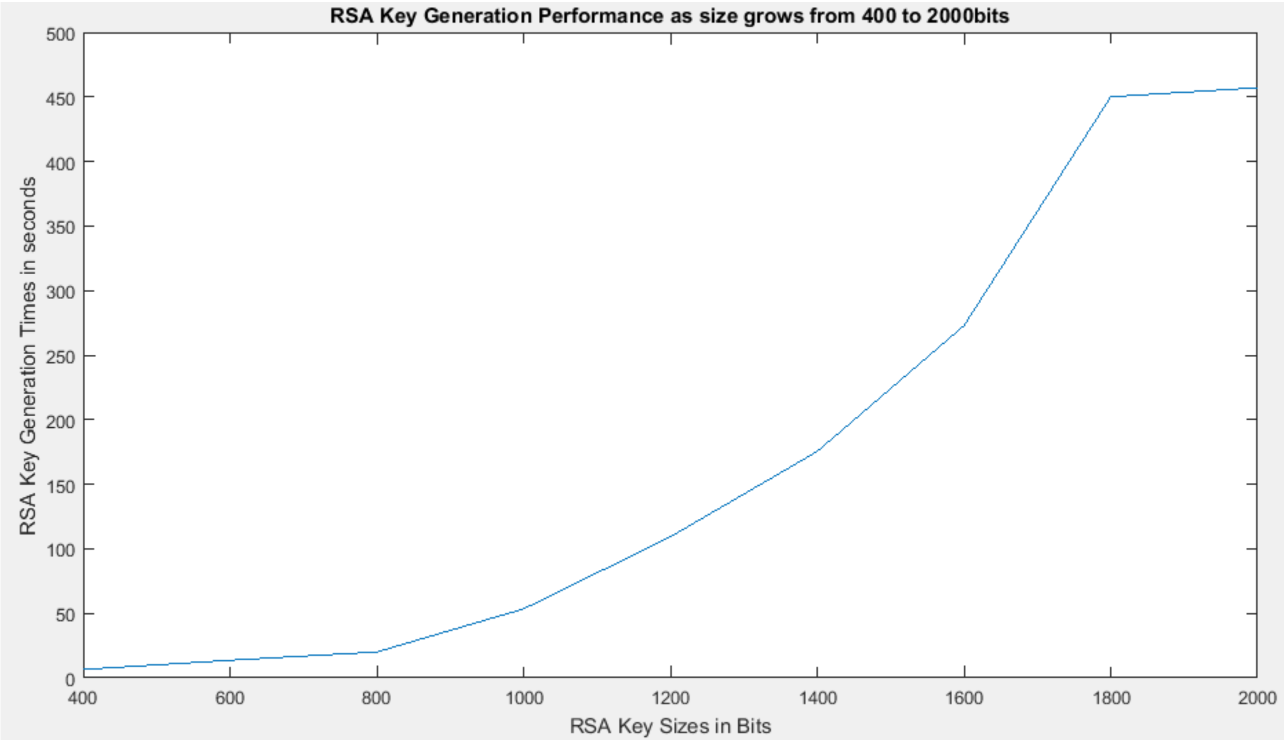

Figure 2.1: RSA Key Generation Performance

As you can see from the performance graph portrayed figure 2.1, the increase in generation time grows faster as the key sizes increases. But so weirdly, the gap between generation time of 1800-bit key and 2000-bit key narrowed to about 7 seconds.

3) A Tcl procedure for safe prime number generation, named "makesafeprime"

```
#----------------------------------------------------------------------------
#@uthor: Alem H. Fitwi
# A Tcl procedure for the generation of a safe prime number, P=2*q+1, where
# q is also a prime number. This procedure repeatedly calls the makeprime
#procedure provided in Dr. Scott Craver's note to generate P until the primality
#test of q=(p-1)/2, which is done by calling the millerRabing procedure, passes.
#----------------------------------------------------------------------------
proc makesafeprime bits {
      while {1} {
          set P [makeprime $bits]
          set q [expr ($P-1)/2]
          if {[millerRabin $q]==1} {
              break;
          }
      }
     set P
}
```

By running along will all the required procedures in the handout, I tested it for four cases where the sizes of the safe prime are 3 bits, 4 bits, 8 bits, and 100 bits. Screen shot of Test results are portrayed in the next page!

```
% #----------------------------------------------------------------------------
proc makesafeprime bits {
      while {1} {
         set P [makeprime $bits]
         set q [expr ($P-1)/2]
         if {[millerRabin $q]==1} {
             break;
         }
      }
     set P
}

#----------------------------------------------------------------------------
#Test is done for 3bits,4bits, 8bits, and 100bits long numbers
#----------------------------------------------------------------------------
set bits 3
puts "The generated {$bits} bits long safe prime number is:{[makesafeprime $bits]}"% % % %
%
The generated {3} bits long safe prime number is:{7}
% #----------------------------------------------------------------------------
set bits 4
puts "The generated {$bits} bits long safe prime number is:{[makesafeprime $bits]}"% 4
%
The generated {4} bits long safe prime number is:{11}
% #----------------------------------------------------------------------------
set bits 8
puts "The generated {$bits} bits long safe prime number is:{[makesafeprime $bits]}"% 8
%
The generated {8} bits long safe prime number is:{179}
% #----------------------------------------------------------------------------
set bits 100
puts "The generated {$bits} bits long safe prime number is:{[makesafeprime $bits]}"% 100
%
The generated {100} bits long safe prime number is:{938245978211499043391924137187}
%
```

4) Implementation of Diffie-Hellmann Key Exchange protocol

(a) Tcl code for “isgenerator {element p}” procedure that checks if an element is a generator

As we had it all in class, for any safe prime 2 or -2 is one of the possible generators of the multiplicative group modulo safe prime! Hence, the tcl code I wrote exploits this property of 2 and the fact that a safe prime has only four possible orders (1, 2, q, and 2q)

```
#-----------------------------------------------------------------
#@uthor: Alem H. Fitwi
# A Tcl procedure that finds a generator modulo safe prime number
# The modexp procedure is taken from Dr. Scott Craver's note
#-----------------------------------------------------------------
# 4. (a) proc for a generator

proc isgenerator {element p} {
        if {$element==2} {
                if { [modexp $element ($p-1)/2 $p]!=1} {
                        set g 2
                } else {set g [expr $p-2]}
        } elseif {$element!=2} {
                if {[modexp $element 1 $p]!=1 &&
                    [modexp $element 2 $p]!=1 &&
                    [modexp $element ($p-1)/2 $p]!=1} {
                        set g $element
                 }
         }
        set g
}
```

Hereunder is the test I carried out to make sure that the procedure works correctly. I used known safe primes some of whose generators are known to verify it, and then random ones.

```
% #-------------------------------------------------------------------------------
proc isgenerator {element p} {
        if {$element==2} {
                if { [modexp $element ($p-1)/2 $p]!=1} {
                        set g 2
                } else {set g [expr $p-2]}
        } elseif {$element!=2} {
                if {[modexp $element 1 $p]!=1 &&
                    [modexp $element 2 $p]!=1 &&
                    [modexp $element ($p-1)/2 $p]!=1} {
                        set g $element
                 }
         }
        set g
}
#-----------------------------------------------------------------------------
# Test-case-1: Testing using known values of P and element
# let p=23, and element=2 ==> generated g must be -2 or 21#
set element 2
set p 23
puts "The known safe prime number is {$p}"
puts "The generator is {[isgenerator $element $p]}"% % % % % 2
% 23
% The known safe prime number is {23}
%
The generator is {21}
% #-------------------------------------------------------------------------------
# Test-case-2:Testing using a random safe prime number from makesafeprime proc
# let p be set to [makesafeprime 100], and element=2.
set element 2
set p [makesafeprime 100]
puts "The generated 100bits long safe prime number is {$p}"
puts "The generator is {[isgenerator $element $p]}"% % % 2
%
1206834574731366970403674162247
% The generated 100bits long safe prime number is {1206834574731366970403674162247}
% The generator is {1206834574731366970403674162245}
%
```

100bit long safe prime is used here

(b) Generate a 100-bit long safe prime, compute g, and then c=g^2 mod p

Here again a new random safe prime number is generated, g is found, and c=g^2 mod p is computed as ensues:

```
#4. (b) generate a safe prime, find a generator g, and compute c=g^2 mod p
set p [makesafeprime 100]
puts "The generated 100bits long safe prime number is {$p}"
set element 2
puts "The element is {$element}"
set g [isgenerator $element $p]
puts "The generator is {$g}"
set c [modexp $g 2 $p]
puts "The computed value of c is {$c}"
```

Then, the following results were collected:

```
% #-----------------------------------------------------------------------
#4. (b) generate a safe prime, find a generator g, and compute c=g^2 mod p
set p [makesafeprime 100]
puts "The generated 100bits long safe prime number is {$p}"
set element 2
puts "The element is {$element}"
set g [isgenerator $element $p]
puts "The generator is {$g}"
set c [modexp $g 2 $p]
puts "The computed value of c is {$c}"% %
1126595504753551790974346575907
% The generated 100bits long safe prime number is {1126595504753551790974346575907}
% 2
% The element is {2}
% 2
% The generator is {2}
% 4
% The computed value of c is {4}
%
```

(c) *Generate secret values a and b for Alice and Bob using the randomlessthan procedure*

```
#-----------------------------------------------------------------------
# 4. (c) generate secret values a and b for Alice and Bod using randomless
set a [randomlessthan 20]
puts "Alice's secret exponent, a is {$a}"
set b [randomlessthan 20]
puts "Bob's secret exponent, b is {$b}"
#-----------------------------------------------------------------------
```

```
% #-----------------------------------------------------------------------
# 4. (c) generate secret values a and b for Alice and Bod using randomless
set a [randomlessthan 20]
puts "Alice's secret exponent, a is {$a}"
set b [randomlessthan 20]
puts "Bob's secret exponent, b is {$b}"% % 10
% Alice's secret exponent, a is {10}
% 15
%
Bob's secret exponent, b is {15}
%
```

(d) Compute the public values A and B

```
#-----------------------------------------------------------------------
# 4. (d) compute the public values, A and B for Alice and Bob using c as base
set A [modexp $c $a $p]
puts "c raised to a mod p, A={$A}"
set B [modexp $c $b $p]
puts "c raised to b mod p, B={$B}"
#-----------------------------------------------------------------------
```

```
% #-----------------------------------------------------------------------
# 4. (d) compute the public values, A and B for Alice and Bob using c as base
set A [modexp $c $a $p]
puts "c raised to a mod p, A={$A}"
set B [modexp $c $b $p]
puts "c raised to b mod p, B={$B}"% % 1048576
% c raised to a mod p, A={1048576}
% 1073741824
%
c raised to b mod p, B={1073741824}
%
```

(e) *Compute A^b and B^a*

```
#---------------------------------------------------------------------------
# 4. (e) compute A raised to b and B raised to a
set Ab [modexp $A $b $p]
puts "The value of the public exponent A={$A} raised to {$b} mod p is {$Ab}"
set Ba [modexp $B $a $p]
puts "The value of the public exponent B={$B} raised to {$a} mod p is {$Ba}"
#---------------------------------------------------------------------------
```

Generally A^b, B^a, and g^{ba} are expected to be modulo p congruent. However, A and B are computed using c=g^2modp=4 as a base not the generator g. Hence, the modulo congruency might exist in this case for c=4 is not a generator because c^(q) mod p is 1. Below is the result, anyways.

```
% #-------------------------------------------------------------------
#Check whether A^b = B^a = g^(ab) mod p
set Aa [modexp [modexp $g $a $p] $b $p]
puts "A raised to a is {$Aa}"
set Bb [modexp [modexp $g $b $p] $a $p]
puts "B raised to b mod p is {$Bb}"
set gab [modexp $g ($b*$a) $p]
puts "g raised to ab mod p is {$gab}"% % 378465885356608094582523568884
% A raised to a is {378465885356608094582523568884}
% 378465885356608094582523568884
% B raised to b mod p is {378465885356608094582523568884}
% 378465885356608094582523568884                    Ab=Ba=gab
%
g raised to ab mod p is {378465885356608094582523568884}
%
```