# Lab 9: Future Crypto

## 1 Details

Aim: To provide a foundation in some of the up-and-coming methods in cryptography.

## Light-weight crypto

**L2.1** In many operations within public key methods we use the exponential operation:

gx (mod p)

If we compute the value of gx and then perform a (mod p) it is a very costly operation in terms of CPU as the value of gx will be large. A more efficient method it use Montgomery reduction and use pow(g,x,p).

import random

g=3

x= random.randint(2, 100)

n=997

res1 = g\*\*x % n

res2= pow(g,x, n)

print res1

print res2

**Now add some code to determine the time taken to perform the two operations, and compare them:**

**Now increase the range for x (so that it is relatively large) and make n as a large prime number. What do you observe from the performance:**

**L2.1** Normally light-weight crypto has to be fast and efficient. The XTEA method is one of the fastest around. Some standard open source code in Node.js is (use **npm install xtea**):

var xtea = require('xtea');

var plaintext = new Buffer('ABCDEFGH', 'utf8');

var key = new Buffer('0123456789ABCDEF0123456789ABCDEF', 'hex');

var ciphertext = xtea.encrypt( plaintext, key );

console.log('Cipher:\t'+ ciphertext.toString('hex') );

console.log('Decipher:\t'+ xtea.decrypt( ciphertext, key ).toString() );

**A sample run is:**

Cipher: 52deb267335dd52a49837931c233cea8

Decipher: ABCDEFGH

**What is the block and key size of XTEA?**

**Can you add some code to measure the time taken for 1,000 encryptions?**

**Can you estimate the number for encryption keys that could be tried per second for your system?**

**L2.2** RC4 is a stream cipher created by Ron Rivest and has a variable key length. Run the following Python code and test it:

def KSA(key):

keylength = len(key)

S = range(256)

j = 0

for i in range(256):

j = (j + S[i] + key[i % keylength]) % 256

S[i], S[j] = S[j], S[i] # swap

return S

def PRGA(S):

i = 0

j = 0

while True:

i = (i + 1) % 256

j = (j + S[i]) % 256

S[i], S[j] = S[j], S[i] # swap

K = S[(S[i] + S[j]) % 256]

yield K

def RC4(key):

S = KSA(key)

return PRGA(S)

def asctohex(string\_in):

a=""

for x in string\_in:

a = a + ("0"+((hex(ord(x)))[2:]))[-2:]

return(a)

def convert\_key(s):

return [ord(c) for c in s]

key="0102030405"

plaintext = 'Hello'

if (len(sys.argv)>1):

plaintext=str(sys.argv[1])

if (len(sys.argv)>2):

key=str(sys.argv[2])

key = key.decode('hex')

key = convert\_key(key)

keystream = RC4(key)

print "Keystream: ",

for i in range (0,15):

print hex(keystream.next()),

print

print "Cipher: ",

keystream = RC4(key)

for c in plaintext:

sys.stdout.write("%02X" % (ord(c) ^ keystream.next()))

**Now go to** [**https://tools.ietf.org/html/rfc6229**](https://tools.ietf.org/html/rfc6229) **and test a few key generation values and see if you get the same key stream.**

**Tests:**

**Key:** 0102030405  **Key stream (first six bytes):**

**Key: Key stream (first six bytes):**

**Key: Key stream (first six bytes):**

**Key: Key stream (first six bytes):**

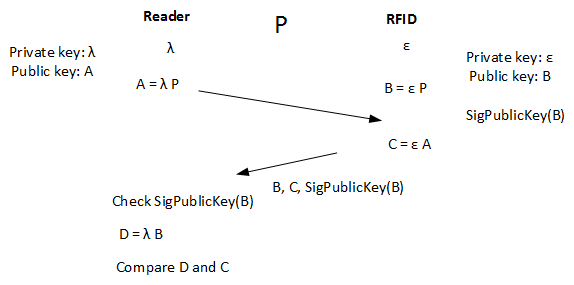
**How does the Python code produce a key stream length which matches the input data stream:**

**Can you test the code by decrypting the cipher stream (note: you just use the same code, and do the same operation again)?**

**RC4 uses an s-Box. Can you find a way to print out the S-box values for a key of** “0102030405”?

**What are the main advantages of having a variable key size and having a stream cipher in light-weight cryptography?**

**L1.3** The ELLI method can be used to identify an RFID tag.



Can you run the following code and determine that it works (C and D should be the same)? Can you also explain how it works?

from os import urandom

from eccsnacks.curve25519 import scalarmult, scalarmult\_base

import binascii

lamb = urandom(32)

a = scalarmult\_base(lamb)

eps = urandom(32)

b = scalarmult\_base(eps)

c = scalarmult(eps, a)

d = scalarmult(lamb, b)

print "RFID private key: ",binascii.hexlify(eps)

print "Reader private key: ",binascii.hexlify(lamb)

print

print "A value: ",binascii.hexlify(a)

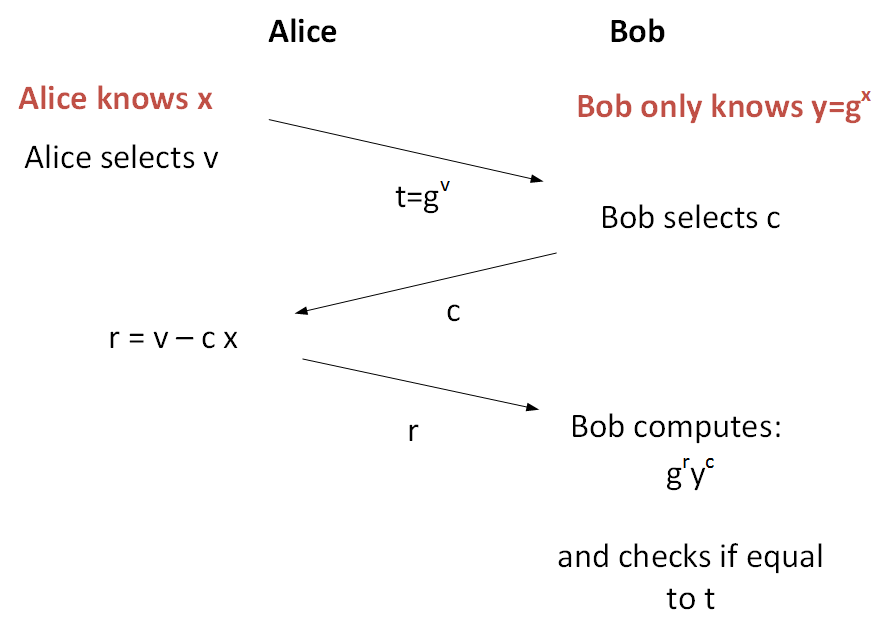
print "B value: ",binascii.hexlify(b)

print "C value: ",binascii.hexlify(c)

print "D value: ",binascii.hexlify(d)

## 3 Zero-knowledge proof (ZKP)

**L1.3** With ZKP, Alice can prove that he still knows something to Bob, without revealing her secret. At the basis of many methods is the Fiat-Shamir method:



Ref: https://asecuritysite.com/encryption/fiat

The following code implements some basic code for Fiat-Shamir, can you prove that for a number of values of x, that Alice will always be able to prove that she knows x.

x: Proved: Y/N

x: Proved: Y/N

x: Proved: Y/N

x: Proved: Y/N

The value of **n** is a prime number. Now increase the value of n, and determine the effect that this has on the time taken to compute the proof:

import sys

import random

n=97

g= 5

x = random.randint(1,5)

v = random.randint(n//2,n)

c = random.randint(1,5)

y= pow(g,x, n)

t = pow(g,v,n)

r = (v - c \* x)

print r

if (r<0): r=-r

Result = ( pow(g,r,n)) \* (pow(y,c,n)) % n

print 'x=',x

print 'c=',c

print 'v=',v

print 'P=',n

print 'G=',g

print '======'

print 't=',t

print 'r=',Result

if (t==Result):

print 'Alice has proven she knows x'

else:

print 'Alice has not proven she knows x'