Math 480 -- some crypto

2. Crypto in Sage

Enigma

(Somebody) and Hellmand and Diffie

Pycrypto: Symmetric Cypher Library

Sage includes the PyCrypto library, which is intended to ``provide a reliable and stable base for writing Python programs that require cryptographic functions. [..] Some modules are implemented in C for performance; others are written in Python for ease of modification."

```
# by A.M. Kuchling
```

```
import Crypto
help(Crypto)
```

```
Help on package Crypto:

NAME
    Crypto - Python Cryptography Toolkit

FILE
    /Users/was/build/sage-3.0.alpha1/local/lib/python2.5/site-packages/Crypto/__init_

DESCRIPTION
    A collection of cryptographic modules implementing various algorithms
    and protocols.

    Subpackages:
    Crypto.Cipher             Secret-key encryption algorithms (AES, DES, ARC4)
    Crypto.Hash               Hashing algorithms (MD5, SHA, HMAC)
    Crypto.Protocol           Cryptographic protocols (Chaffing, all-or-nothing
                              transform).   This package does not contain any
                              network protocols.
    Crypto.PublicKey          Public-key encryption and signature algorithms
                              (RSA, DSA)
    Crypto.Util               Various useful modules and functions (long-to-string
                              conversion, random number generation, number
                              theoretic functions)

PACKAGE CONTENTS
    Cipher (package)
    Hash (package)
    Protocol (package)
    PublicKey (package)
    Util (package)
    test

DATA
    __all__ = ['Cipher', 'Hash', 'Protocol', 'PublicKey', 'Util']
    __revision__ = '$Id: __init__.py,v 1.12 2005/06/14 01:20:22 akuchling ...
    __version__ = '2.0.1'

VERSION
    2.0.1
```

```
from Crypto.Hash import MD5
m = MD5.new('abc')
m.digest()
```

```
'\x90\x01P\x98<\xd2O\xb0\xd6\x96?}(\xe1\x7fr'
```

```
m.hexdigest()
```

```
'900150983cd24fb0d6963f7d28e17f72'
```

```
@interact
def _(msg = "abc"):
   print "The msg:\n\n'%s'\n\nhas MD5 hash:\n\n"%msg
   print MD5.new(msg).hexdigest()
```

msg abc

```
The msg:

'abc'

has MD5 hash:

900150983cd24fb0d6963f7d28e17f72
```

```
import Crypto.Hash
```

```
help(Crypto.Hash)
```

```
Help on package Crypto.Hash in Crypto:

NAME
    Crypto.Hash - Hashing algorithms

FILE
    /Users/was/build/sage-3.0.alpha1/local/lib/python2.5/site-packages/Crypto/Hash/_

DESCRIPTION
    Hash functions take arbitrary strings as input, and produce an output
    of fixed size that is dependent on the input; it should never be
    possible to derive the input data given only the hash function's
    output.  Hash functions can be used simply as a checksum, or, in
    association with a public-key algorithm, can be used to implement
    digital signatures.

    The hashing modules here all support the interface described in PEP
    247, "API for Cryptographic Hash Functions".

    Submodules:
    Crypto.Hash.HMAC          RFC 2104: Keyed-Hashing for Message Authentication
    Crypto.Hash.MD2
    Crypto.Hash.MD4
    Crypto.Hash.MD5
    Crypto.Hash.RIPEMD
    Crypto.Hash.SHA

PACKAGE CONTENTS
    HMAC
    MD2
    MD4
    MD5
    RIPEMD
    SHA
    SHA256

DATA
    __all__ = ['HMAC', 'MD2', 'MD4', 'MD5', 'RIPEMD', 'SHA', 'SHA256']
    __revision__ = '$Id: __init__.py,v 1.6 2003/12/19 14:24:25 akuchling E...
```

```
from Crypto.Hash import SHA
m = SHA.new('abc')
```

```
m.hexdigest()
```

```
'a9993e364706816aba3e25717850c26c9cd0d89d'
```

```
# The SHA hash in Sage is very fast:

s = 'lksj skljdf'*100
print len(s)
timeit("SHA.new(s).hexdigest")
```

```
1100
625 loops, best of 3: 7.91 Âµs per loop
```

```
import Crypto.Cipher
help(Crypto.Cipher)
```

```
Help on package Crypto.Cipher in Crypto:

NAME
    Crypto.Cipher - Secret-key encryption algorithms.

FILE
    /Users/was/build/sage-3.0.alpha1/local/lib/python2.5/site-packages/Crypto/Cipher

DESCRIPTION
    Secret-key encryption algorithms transform plaintext in some way that
    is dependent on a key, producing ciphertext. This transformation can
    easily be reversed, if (and, hopefully, only if) one knows the key.

    The encryption modules here all support the interface described in PEP
    272, "API for Block Encryption Algorithms".

    If you don't know which algorithm to choose, use AES because it's
    standard and has undergone a fair bit of examination.

    Crypto.Cipher.AES          Advanced Encryption Standard
    Crypto.Cipher.ARC2         Alleged RC2
    Crypto.Cipher.ARC4         Alleged RC4
    Crypto.Cipher.Blowfish
    Crypto.Cipher.CAST
    Crypto.Cipher.DES          The Data Encryption Standard.  Very commonly used
                               in the past, but today its 56-bit keys are too small.
    Crypto.Cipher.DES3         Triple DES.
    Crypto.Cipher.IDEA
    Crypto.Cipher.RC5
    Crypto.Cipher.XOR          The simple XOR cipher.

PACKAGE CONTENTS
    AES
    ARC2
    ARC4
    Blowfish
    CAST
    DES
    DES3
    IDEA
    RC5
    XOR

DATA
    __all__ = ['AES', 'ARC2', 'ARC4', 'Blowfish', 'CAST', 'DES', 'DES3', '...
    __revision__ = '$Id: __init__.py,v 1.7 2003/02/28 15:28:35 akuchling E...
```

```
from Crypto.Cipher import DES
obj=DES.new('abcdefgh', DES.MODE_ECB)   # The MODE_ is different than in the docs on
the web page
plain="The Sage Math Software is a space monster.  But a good kind of space monster."
len(plain)
```

```
77
```

```
obj.encrypt(plain)
```

```
Traceback (click to the left for traceback)
...
ValueError: Input strings must be a multiple of 8 in length
```

```
ciph=obj.encrypt(plain + ' '*(8-len(plain)%8))
ciph
```

```
'_\x08zb\x10~\xb0\x84\n\xdfH\xd1\x8b@\xc0\xda\ru:\xb4\xd7\xe5\x93\x1\
8\xd7j\xd8\xc3\xf7\xb2\xa1@z\x19\xd31\xbe\x05\x15\xf8\x1d\xc1\xd3\x8\
18:\xca\xce\x8e\xbf\xda\xac\xe5\x81\x13\x00F\x917L\x18\x18Z\x08\xce-\
q\x8d\xab\x0f\x8a\x8f8v\xb3\t\xd6\xbf\xe5&'
```

```
obj.decrypt(ciph)
```

```
'The Sage Math Software is a space monster.  But a good kind of
space monster.   '
```

```
@interact
def _(key="abcdefgh", plain="A message."):
  from Crypto.Cipher import DES
  obj = DES.new(key, DES.MODE_ECB)
  print repr(obj.encrypt(plain + ' '*(8-len(plain)%8)))
```

key abcdefgh

plain A message.

```
'\xc2\xf1\xea\xff\x81B\xc8\x12tL+NY/\x9e%'
```

```
import Crypto.PublicKey
help(Crypto.PublicKey)
```

```
Help on package Crypto.PublicKey in Crypto:

NAME
    Crypto.PublicKey - Public-key encryption and signature algorithms.

FILE
    /Users/was/build/sage-3.0.alpha1/local/lib/python2.5/site-packages/Crypto/Public]

DESCRIPTION
    Public-key encryption uses two different keys, one for encryption and
    one for decryption.  The encryption key can be made public, and the
    decryption key is kept private.  Many public-key algorithms can also
    be used to sign messages, and some can *only* be used for signatures.
```

```
        Crypto.PublicKey.DSA      Digital Signature Algorithm. (Signature only)
        Crypto.PublicKey.ElGamal  (Signing and encryption)
        Crypto.PublicKey.RSA      (Signing, encryption, and blinding)
        Crypto.PublicKey.qNEW     (Signature only)

    PACKAGE CONTENTS
        DSA
        ElGamal
        RSA
        pubkey
        qNEW

    DATA
        __all__ = ['RSA', 'DSA', 'ElGamal', 'qNEW']
        __revision__ = '$Id: __init__.py,v 1.4 2003/04/03 20:27:13 akuchling E...
```

```
from Crypto.Hash import MD5
from Crypto.PublicKey import RSA
import random
def randfunc(n):
   return ''.join(str(random.random())[4] for _ in xrange(n))

time RSAkey = RSA.generate(int(1024), randfunc)
```

```
    Time: CPU 1.81 s, Wall: 1.89 s
```

```
hash = MD5.new('This is a Sage').digest()
signature = RSAkey.sign(hash, "")
signature   # Print what an RSA sig looks like.
```

```
    (86033408128284843566677565227948550514266651328029959949039613360O4\
    41411176045404686382488434682271333770060475259075634928505596465429\
    296235914230272064682000328711848406924330510754392788944723971035 37\
    848199412247387092189802333527679968252026956051024167600711018657 11\
    047935764894167809661307056287600361L,)
```

```
RSAkey.verify(hash, signature)     # This sig will check out
```

```
    1
```

```
RSAkey.verify(hash[:-1], signature) # This sig will fail
```

```
    0
```

David Kohel's book and code

This website

http://echidna.maths.usyd.edu.au/~kohel/tch/Crypto/

contains a **very nice book** on many aspects of cryptography, and it uses Sage throughout for examples. It covers, elementary cryptanalysis, information theory, block and stream ciphers, public-key cryptosystems, and digital signatures.

```
# We illustrate the classical substitution cypher, which is easy to crack for large
messages
# using a frequency analysis...
```

```
# Create the "monoid" of all strings on the symbols A-Z.
S = AlphabeticStrings()
S
```

    Free alphabetic string monoid on A-Z

```
# Encode a string in this monoid.
msg = S('SAGEMATH')
msg
```

    SAGEMATH

```
# Create an object that allows was to make specific substitution cyphers.
E = SubstitutionCryptosystem(S)
E
```

    Substitution cryptosystem on Free alphabetic string monoid on A-Z

```
# Generate a random substitution cypher key (permutation of the alphabet)
K = E.random_key()
K
```

    DGNFPRHXVOTWEUBJKYALISMCQZ

```
# Make object that encrypts using the above substitution
encrypt = E(K)
encrypt
```

    DGNFPRHXVOTWEUBJKYALISMCQZ

```
# Object that decrypts using the inverse of the above substitution
decrypt = E(E.inverse_key(K))
```

```
# Encode a message in terms of the alphabet
m = E.encoding('WORLDDOMINATION')
m
```

    WORLDDOMINATION

```
# Actual encrypt the encoding
c = encrypt(m); c
```

    MBYWFFBEVUDLVBU

```
# Decrypt the encrypted version
decrypter(c)
```

```
WORLDDOMINATION
```

Implement public key systems from scratch for research, etc.

It is fairly straightforward to implement a wide range of standard public-key crypto-systems directly in Sage. NOTE: You would probably *not* want to use such an implementation for actually deploying a cryptosystem -- use this for educational and testing purposes only. The deployed cryptosystems, it is best to use existing well-tested crypto libraries and tools, for example, PyCrypto (see above), GNUtls, GPG, etc. Part of the reason for this is because of side channel attacks, and also because slower more robust code is vastly better than slightly faster less robust code in the context of cryptosystems that will actually be deployed.

Diffie-Hellman

```
@interact
```

```
def diffie_hellman(button=selector(["New example"],label='',buttons=True),
   bits=("Number of bits of prime", (8,12,..512))):
   maxp = 2^bits
   p = random_prime(maxp)
   k = GF(p)
   if bits>100:
       g = k(2)
   else:
       g = k.multiplicative_generator()
   a = ZZ.random_element(10, maxp)
   b = ZZ.random_element(10, maxp)

   print """
<html>
<style>
.gamodp {
background:yellow
}
.gbmodp {
background:orange
}
.dhsame {
color:green;
font-weight:bold
}
</style>
<h2>%s-Bit Diffie-Hellman Key Exchange</h2>
<ol style="color:#000;font:18px Arial, Helvetica, sans-serif">
<li>Alice and Bob agree to use the prime number p=%s and base g=%s.</li>
<li>Alice chooses the secret integer a=%s, then sends Bob (<span
class="gamodp">g<sup>a</sup> mod p</span>):<br/>%s<sup>%s</sup> mod %s = <span
class="gamodp">%s</span>.</li>
<li>Bob chooses the secret integer b=%s, then sends Alice (<span
class="gbmodp">g<sup>b</sup> mod p</span>):<br/>%s<sup>%s</sup> mod %s = <span
class="gbmodp">%s</span>.</li>
<li>Alice computes (<span class="gbmodp">g<sup>b</sup> mod p</span>)<sup>a</sup> mod
p:<br/>%s<sup>%s</sup> mod %s = <span class="dhsame">%s</span>.</li>
<li>Bob computes (<span class="gamodp">g<sup>a</sup> mod p</span>)<sup>b</sup> mod
p:<br/>%s<sup>%s</sup> mod %s = <span class="dhsame">%s</span>.</li>
</ol></html>
   """ % (bits, p, g, a, g, a, p, (g^a), b, g, b, p, (g^b), (g^b), a, p,
      (g^ b)^a, g^a, b, p, (g^a)^b)
```

New example

Number of bits of prime

8-Bit Diffie-Hellman Key Exchange

1. Alice and Bob agree to use the prime number p=163 and base g=2.
2. Alice chooses the secret integer a=25, then sends Bob (g^a mod p):

2^{25} mod 163 = 67.

3. Bob chooses the secret integer b=208, then sends Alice (g^b mod p):
2^{208} mod 163 = 87.

4. Alice computes (g^b mod $p)^a$ mod p:
87^{25} mod 163 = **10**.

5. Bob computes (g^a mod $p)^b$ mod p:
67^{208} mod 163 = **10**.