APPENDIX B

SAGE EXAMPLES

By Dan Shumow
University of Washington

This appendix contains a number of examples that illustrate cryptographic concepts, organized by the chapter in which those concepts were discussed. All the examples are in Sage.[1] See Appendix C for how to get started using Sage and for a brief introduction to Sage syntax and operations. We begin with a brief introduction to some basic Sage matrix and linear algebra operations.

You should be able to follow the examples in this section as written. However, if you have difficulty interpreting the Sage code, please refer to Section C.2 in Appendix C.

B.1 LINEAR ALGEBRA AND MATRIX FUNCTIONALITY

Sage includes linear algebra and matrix functionality. The following shows some of the basic functionality applicable to cryptography

In Sage you specify a matrix as a list of lists of numbers, passed to the `matrix` function. For example, passing a list of lists of integers as follows:

```
sage: M = matrix([[1, 3],[7,9]]); M
[1 3]
[7 9]
```

Alternately, passing a list of lists of rationals as follows:

```
sage: M = matrix([[1/2, 2/3, 3/4],[5, 7, 8]]); M
[1/2 2/3 3/4]
[  5   7   8]
```

[1] All of the Sage code in this appendix is available online available at this book's Premium Content Web site in .sage files, so that you can load and execute the programs if you wish.

You can specify that the input should be reduced by a modulus, using the IntegerModRing (functionality to be described later)

```
Sage: R = IntegerModRing(100)
sage: M = matrix(R, [[1],[102],[1003]]); M
[1]
[2]
[3]
```

Or that the input should be considered in a finite field (also to be described later.)

```
sage: F = GF(2);
sage: M = matrix(F, [[1, 2, 0, 3]]); M
[1 0 0 1]
```

Sage also supports multiplication, addition, and inversion of matrices as follows:

```
sage: M1 = matrix([[1, 2],[3,4]]);
sage: M2 = matrix([[1,-1],[1, 1]]);
sage: M1*M2
[3 1]
[7 1]

sage: M1 + M2
[2 1]
[4 5]

sage: M2^-1
[ 1/2  1/2]
[-1/2  1/2]
```

B.2 CHAPTER 2: CLASSICAL ENCRYPTION

The following functions are useful for classical cipher examples and exercises:

See Preface for access information.

```
en_alphabet = "abcdefghijklmnopqrstuvwxyz"

#
# This function converts a single character into its numeric value
#
def is_alphabetic_char(c):
    return (c.lower() in en_alphabet)

#
# This function returns true if and only if the character c is an
# alphabetic character
#
def char_to_num(c):
    return en_alphabet.index(c.lower())

#
# This function returns the character corresponding to x mod 26
# in the English alphabet
#
def num_to_char(x):
    return en_alphabet[x % 26]
```

Example 1. Implement Sage encryption/decryption functions that take a key (as an integer in 0, 1, 2, … , 25), and a string. The function should only operate on the characters ‘a’, ‘b’, … ‘z’ (both upper and lower case), and it should leave any other characters, unchanged.

Solution:

```
def CaesarEncrypt(k, plaintext):

    ciphertext = ""

    for j in xrange(len(plaintext)):

        p = plaintext[j]

        if is_alphabetic_char(p):
            x = (k + char_to_num(p)) % 26
            c = num_to_char(x)
        else:
            c = p

        ciphertext += c

    return ciphertext


def CaesarDecrypt(k, ciphertext):

    plaintext = ""

    for j in xrange(len(ciphertext)):
```

```
        c =  ciphertext[j]

        if is_alphabetic_char(c):
            x = (char_to_num(c) - k) % 26
            p = num_to_char(x)
        else:
            p = c

        plaintext += p

    return plaintext
```

Example 2. Implement a function that performs a brute force attack on a ciphertext, it should print a list of the keys and associated decryptions. It should also take an optional parameter that takes a substring and only prints out potential plaintexts that contain that decryption.

Solution:

```
def BruteForceAttack(ciphertext, keyword=None):

    for k in xrange(26):
        plaintext = CaesarDecrypt(k, ciphertext)

        if (None==keyword) or (keyword in plaintext):
            print "key", k, "decryption", plaintext

    return
```

Example 3. Show the output of your encrypt function (Example 1) on the following (key, plaintext) pairs:

- k = 6 plaintext = "Get me a vanilla ice cream, make it a double."
- k = 15 plaintext = "I don't much care for Leonard Cohen."
- k = 16 plaintext = "I like root beer floats."

Solution:

```
sage: k = 6; plaintext = 'Get me a vanilla ice cream, make it a double.'
sage: CaesarEncrypt(k, plaintext)
'mkz sk g bgtorrg oik ixkgs, sgqk oz g juahrk.'

sage: k = 15; plaintext = "I don't much care for Leonard Cohen."
sage: CaesarEncrypt(k, plaintext)
"x sdc'i bjrw rpgt udg atdcpgs rdwtc."
```

```
sage: k = 16; plaintext = "I like root beer floats."
sage: CaesarEncrypt(k, plaintext)
'y byau heej ruuh vbeqji.'
```

Example 4. Show the output of your decrypt function (Example 1) on the following (key, ciphertext) pairs:

- k = 12 ciphertext = 'nduzs ftq buzq oazqe.'
- k = 3 ciphertext = "fdhvdu qhhgv wr orvh zhljkw."
- k = 20 ciphertext = "ufgihxm uly numnys."

Solution:

```
sage: k = 12; ciphertext = "nduzs ftq buzq oazqe."
sage: CaesarDecrypt(k, ciphertext)
'bring the pine cones.'

sage: k = 3; ciphertext = "fdhvdu qhhgv wr orvh zhljkw."
sage: CaesarDecrypt(k, ciphertext)
'caesar needs to lose weight.'

sage: k = 20; ciphertext = "ufgihxm uly numnys."
sage: CaesarDecrypt(k, ciphertext)
'almonds are tastey.'
```

Example 5. Show the output of your attack function (Example 4) on the following ciphertexts, if an optional keyword is specified, pass that to your attack function:

- ciphertext = 'gryy guru gob tab gb nzoebfr puncry.' keyword = 'chapel'
- ciphertext = 'wziv kyv jyfk nyve kyv tpdsrcj tirjy.' keyword = 'cymbal'
- ciphertext = 'baeeq klwosjl osk s esf ozg cfwo lgg emuz.' no keyword

Solution:

```
sage: ciphertext = 'gryy gurz gb tb gb nzoebfr puncry.'
sage: BruteForceAttack(ciphertext, 'chapel')
key 13 decryption tell them to go to ambrose chapel.

sage: ciphertext = 'wziv kyv jyfk nyve kyv tpdsrcj tirjy.'
sage: BruteForceAttack(ciphertext, 'cymbal')
key 17 decryption fire the shot when the cymbals crash.
```

```
sage: ciphertext =  'baeeq klwosjl osk s esf ozg cfwo lgg emuz.'
sage: BruteForceAttack(ciphertext)
key 0 decryption baeeq klwosjl osk s esf ozg cfwo lgg emuz.
key 1 decryption azddp jkvnrik nrj r dre nyf bevn kff dlty.
key 2 decryption zycco ijumqhj mqi q cqd mxe adum jee cksx.
key 3 decryption yxbbn hitlpgi lph p bpc lwd zctl idd bjrw.
key 4 decryption xwaam ghskofh kog o aob kvc ybsk hcc aiqv.
key 5 decryption wvzzl fgrjneg jnf n zna jub xarj gbb zhpu.
key 6 decryption vuyyk efqimdf ime m ymz ita wzqi faa ygot.
key 7 decryption utxxj dephlce hld l xly hsz vyph ezz xfns.
key 8 decryption tswwi cdogkbd gkc k wkx gry uxog dyy wemr.
key 9 decryption srvvh bcnfjac fjb j vjw fqx twnf cxx vdlq.
key 10 decryption rquug abmeizb eia i uiv epw svme bww uckp.
key 11 decryption qpttf zaldhya dhz h thu dov ruld avv tbjo.
key 12 decryption posse yzkcgxz cgy g sgt cnu qtkc zuu sain.
key 13 decryption onrrd xyjbfwy bfx f rfs bmt psjb ytt rzhm.
key 14 decryption nmqqc wxiaevx aew e qer als oria xss qygl.
key 15 decryption mlppb vwhzduw zdv d pdq zkr nqhz wrr pxfk.
key 16 decryption lkooa uvgyctv ycu c ocp yjq mpgy vqq owej.
key 17 decryption kjnnz tufxbsu xbt b nbo xip lofx upp nvdi.
key 18 decryption jimmy stewart was a man who knew too much.
key 19 decryption ihllx rsdvzqs vzr z lzm vgn jmdv snn ltbg.
key 20 decryption hgkkw qrcuypr uyq y kyl ufm ilcu rmm ksaf.
key 21 decryption gfjjv pqbtxoq txp x jxk tel hkbt qll jrze.
key 22 decryption feiiu opaswnp swo w iwj sdk gjas pkk iqyd.
key 23 decryption edhht nozrvmo rvn v hvi rcj fizr ojj hpxc.
key 24 decryption dcggs mnyquln qum u guh qbi ehyq nii gowb.
key 25 decryption cbffr lmxptkm ptl t ftg pah dgxp mhh fnva.
```

B.3 CHAPTER 3: BLOCK CIPHERS AND THE DATA ENCRYPTION STANDARD

Example 1. This example implements simplified DES, which is described in Appendix G.

```
#
# The Expansions/Permutations are stored as lists of bit positions
#

P10_data = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6];

P8_data = [6, 3, 7, 4, 8, 5, 10, 9];

LS1_data = [2, 3, 4, 5, 1];

LS2_data = [3, 4, 5, 1, 2];

IP_data = [2, 6, 3, 1, 4, 8, 5, 7];

IPinv_data = [4, 1, 3, 5, 7, 2, 8, 6];
```

```
EP_data = [4, 1, 2, 3, 2, 3, 4, 1];

P4_data = [2, 4, 3, 1];

SW_data = [5, 6, 7, 8, 1, 2, 3, 4];

#
# SDES lookup tables
#

S0_data = [[1, 0, 3, 2],
           [3, 2, 1, 0],
           [0, 2, 1, 3],
           [3, 1, 3, 2]];

S1_data = [[0, 1, 2, 3],
           [2, 0, 1, 3],
           [3, 0, 1, 0],
           [2, 1, 0, 3]];

def ApplyPermutation(X, permutation):
    r"""
    This function takes a permutation list (list of bit positions.)
    And outputs a bit list with the bits taken from X.
    """
    # permute the list X
    l = len(permutation);
    return [X[permutation[j]-1] for j in xrange(l)];

def ApplySBox(X, SBox):
    r"""
    This function Applies the SDES SBox (by table look up
    """
    r = 2*X[0] + X[3];
    c = 2*X[1] + X[2];
    o = SBox[r][c];
    return [o & 2, o & 1];

#
# Each of these functions uses ApplyPermutation
# and a permutation list to perform an SDES
# Expansion/Permutation
#

def P10(X):
    return ApplyPermutation(X, P10_data);

def P8(X):
    return ApplyPermutation(X, P8_data);

def IP(X):
    return ApplyPermutation(X, IP_data);
```

```
def IPinv(X):
    return ApplyPermutation(X, IPinv_data);

def EP(X):
    return ApplyPermutation(X, EP_data);

def P4(X):
    return ApplyPermutation(X, P4_data);

def SW(X):
    return ApplyPermutation(X, SW_data);

def LS1(X):
    return ApplyPermutation(X, LS1_data);

def LS2(X):
    return ApplyPermutation(X, LS2_data);

#
# These two functions perform the SBox substitutions
#
def S0(X):
    return ApplySBox(X, S0_data);

def S1(X):
    return ApplySBox(X, S1_data);

def concatenate(left, right):
    r"""
    Joins to bit lists together.
    """
    ret = [left[j] for j in xrange(len(left))];
    ret.extend(right);
    return ret;

def LeftHalfBits(block):
    r"""
    Returns the left half bits from block.
    """
    l = len(block);
    return [block[j] for j in xrange(l/2)];

def RightHalfBits(block):
    r"""
    Returns the right half bits from block.
    """
    l = len(block);
    return [block[j] for j in xrange(l/2, l)];

def XorBlock(block1, block2):
    r"""
```

```
    Xors two blocks together.
    """
    l = len(block1);
    if (l != len(block2)):
        raise ValueError, "XorBlock arguments must be same length"
    return [(block1[j]+block2[j]) % 2 for j in xrange(l)];

def SDESKeySchedule(K):
    r"""
    Expands an SDES Key (bit list) into the two round keys.
    """
    temp_K = P10(K);

    left_temp_K = LeftHalfBits(temp_K);
    right_temp_K = RightHalfBits(temp_K);

    K1left = LS1(left_temp_K);
    K1right = LS1(right_temp_K);

    K1temp = concatenate(K1left, K1right);
    K1 = P8(K1temp);

    K2left = LS2(K1left);
    K2right = LS2(K1right);

    K2temp = concatenate(K2left, K2right);

    K2 = P8(K2temp);

    return (K1, K2);

def f_K(block, K):
    r"""
    Performs the f_K function supplied block and K.
    """
    left_block = LeftHalfBits(block);
    right_block = RightHalfBits(block);

    temp_block1 = EP(right_block);

    temp_block2 = XorBlock(temp_block1, K);

    left_temp_block2 = LeftHalfBits(temp_block2);
    right_temp_block2 = RightHalfBits(temp_block2);

    S0_out = S0(left_temp_block2);
    S1_out = S1(right_temp_block2);

    temp_block3 = concatenate(S0_out, S1_out);

    temp_block4 = P4(temp_block3)
```

```
    temp_block5 = XorBlock(temp_block4, left_block);

    output_block =  concatenate(temp_block5, right_block)

    return output_block;

def SDESEncrypt(plaintext_block, K):
    r"""
    Performs a single SDES plaintext block encryption.
    (Given plaintext and key as bit lists.)
    """

    (K1, K2) = SDESKeySchedule(K);

    temp_block1 = IP(plaintext_block);

    temp_block2 = f_K(temp_block1, K1);

    temp_block3 = SW(temp_block2);

    temp_block4 = f_K(temp_block3, K2);

    output_block = IPinv(temp_block4);

    return output_block;
```

B.4 CHAPTER 4: BASIC CONCEPTS IN NUMBER THEORY AND FINITE FIELDS

Example 1. The Euclidean Algorithm for the greatest common divisor

```
def EUCLID(a,b):
    r"""
    The Euclidean Algorithm for finding the gcd of a and b.
    This algorithm assumes that a > b => 0

    INPUT:

        a - positive integer

        b - nonnegative integer less than a

    OUTPUT:

        g - greatest common divisor of a and b
```

```
    """

    if (b < 0) or ( a <= b):
        raise ValueError, "Expected 0 < a < b"

    (A, B) = (a,b);

    while (True):

        if (0 == B):
            return A;

        R = A % B;
        A = B;
        B = R;
```

Example 2. The Extended Euclidean Algorithm for the greatest common divisor

```
def EXTENDED_EUCLID(m,b):
    r"""
    The Extended Euclidean Algorithm to find gcd(m,b).
    The input is expected to be such that 0 <= b < m.

    INPUT:

        m - positive integer

        b - nonnegative integer less than m

    OUTPUT:

        (g, b_inv) - g is the gcd of m and b, b_inv is the multiplicative
        inverse of b mod m.

    """

    if (m < b) or (b < 0):
        raise ValueError, "Expected input (0 < b < m)"

    (A1,A2,A3) = (1,0,m);
    (B1,B2,B3) = (0,1,b);

    while (True):

        if (0 == B3):
            return (A3, None)

        if (1 == B3):
            return (B3, B2)

        Q = floor(A3/B3)

        (T1,T2,T3) = (A1-Q*B1, A2-Q*B2, A3-Q*B3)
```

```
        (A1, A2, A3) = (B1, B2, B3)
        (B1, B2, B3) = (T1, T2, T3)
```

Example 3. Euclidean algorithm to find gcd of polynomials (with coefficients in a field.)

```
def POLYNOMIAL_EUCLID(A, B):
    r"""
    Euclidian algorithm for polynomial GCD:
    Given two polynomials over the same base field,
    Assuming degree(A) => degree(B) => 0

    INPUT:

        A - polynomial over a field.

        B - polynomial over the same field as A, and 0 <= degree(B) <=
            degree(A)
    OUTPUT:

        G - Greatest Common Divisor of A and B.

    """

    degA = A.degree();
    degB = B.degree();

    if ((degB < 0) or (degA < degB)):
        raise ValueError, "Expected 0 <= degree(B) <= degree(A)"

    while(True):

        if (0 == B):
            return A;

        R = A % B;

        A = B;
        B = R;
```

Example 4. Extended Euclidean algorithm for the gcd of two polynomials (with coefficients in the same field.)

```
def POLYNOMIAL_EXTENDED_EUCLID(m, b):
    r"""
    Extended Euclidian algorithm for polynomial GCD:
    Given two polynomials over the same base field,
    Assuming degree(m) => degree(b) => 0

    INPUT:

        m - polynomial over a field.
```

```
        b - polynomial over the same field as A, and 0 <= degree(B) <=
degree(M)
    OUTPUT:

        (g,b_inv) - The Pair where:

        g - Greatest Common Divisor of m and b.

        m_inv - Is None if G is not of degree 0,
                and otherwise it is the polynomial such that b(X)*b_inv(X) =
                1 mod m(X)

    """

    degm = m.degree();
    degb = b.degree();

    if(degb < 0) or (degm < degb):
        raise ValueError, "expected 0 <= degree(b) <= degree(m)"

    (A1, A2, A3) = (1, 0, m);
    (B1, B2, B3) = (0, 1, b);

    while (True):

        if (0 == B3):
            return (A3, None);

        if (0 == B3.degree()):
            return (B3/B3, B2/B3);

        Q = A3.quo_rem(B3)[0];

        (T1, T2, T3) = (A1 - Q*B1, A2 - Q*B2, A3 - Q*B3);
        (A1, A2, A3) = (B1, B2, B3);
        (B1, B2, B3) = (T1, T2, T3);
```

Example 5. Sage has built in functionality for the gcd function. The regular greatest common divisor function can simply be called as:

```
sage: gcd(15,100)
5

sage: gcd(90,65311)
1
```

You can also call it as a method on Integer objects:

```
sage: x = 10456890
sage: x.gcd(100)
```

```
10
```

The extended Euclidean greatest common divisor algorithm is also built into Sage. Calling xgcd(a,b) returns a tuple, the first element is the gcd, the second and third elements are coefficients u, v such that gcd(a,b) = u*a + v*b. This can be called as:

```
sage: xgcd(17,31)
(1, 11, -6)

sage: xgcd(10, 115)
(5, -11, 1)
```

This can also be called as a method on Integer objects

```
sage: x = 300
sage: x.xgcd(36)
(12, 1, -8)
```

Example 6. Sage includes robust support for working with finite fields and performing finite field arithmetic. To initialize a finite field with prime order, use the GF command passing the order as the parameter.

```
sage: F = GF(2)
sage: F
Finite Field of size 2

sage: F = GF(37)
sage: F
Finite Field of size 37

sage: p = 95131
sage: K = GF(p)
sage: K
Finite Field of size 95131
```

To initialize a field with a prime power order use the GF command with the following syntax (to keep track of the primitive element of the extension field.)

```
sage: F.<a> = GF(128)
sage: F
Finite Field in a of size 2^7
```

To do arithmetic in finite fields use the following syntax:

```
sage: K = GF(37)
sage: a = K(3)
sage: b = K(18)
sage: a - b
22
sage: a + b
21
sage: a * b
17
sage: a/b
31
sage: a^-1
25
sage: 1/a
25
```

To do arithmetic in a finite field with a prime power order, specify elements using the primitive element:

```
sage: F.<a> = GF(128)
sage: b = a^2 + 1
sage: c = a^5 + a^3 + 1
sage: b - c
a^5 + a^3 + a^2
sage: b + c
a^5 + a^3 + a^2
sage: b*c
a^3 + a^2 + a
sage: b/c
a^5 + a^3 + a^2 + a
sage: b^-1
a^5 + a^3 + a
sage: 1/b
a^5 + a^3 + a
```

Example 7. With Sage you can create rings of polynomials over finite fields and do arithmetic with them. To create polynomial rings over finite fields do the following:

```
sage: R.<x> = GF(2)[]
sage: R

Univariate Polynomial Ring in x over Finite Field of size 2 (using NTL)
sage: R.<x> = GF(101)[]
sage: R

sage: R.<x> = F[]
sage: R
Univariate Polynomial Ring in x over Finite Field in a of size 2^7
```

After initializing a polynomial ring, you can then just perform arithmetic as you would expect:

```
sage: R.<x> = GF(2)[]
sage: f = x^3 + x + 1
sage: g = x^5 + x
sage: f + g
x^5 + x^3 + 1
sage: f*g
x^8 + x^6 + x^5 + x^4 + x^2 + x
```

Division is accomplished by the quo_rem function:

```
sage: g.quo_rem(f)
(x^2 + 1, x^2 + 1)
```

You can also compute the greatest common divisor:

```
sage: f.gcd(g)
1

sage: g.gcd(g^2)
x^5 + x

sage: R.<x> = GF(17)[]
sage: f = 3*x^3 + 2*x^2 + x
sage: g = x^2 + 5
sage: f - g
3*x^3 + x^2 + x + 12
sage: f * g
3*x^5 + 2*x^4 + 16*x^3 + 10*x^2 + 5*x
```

```
sage: f.quo_rem(g)
(3*x + 2, 3*x + 7)
```

And computing gcds in this polynomial ring we see:

```
sage: f.gcd(g)
1

sage: f.gcd(x^2 + x)
x
```

When creating a Sage finite field with a prime power order, Sage finds an irreducible polynomial for you. For example:

```
sage: F.<a> = GF(32)
a^5 + a^2 + 1
```

However, there are many irreducible polynomials over GF(2) of degree 5, such as x^5 + x^3 + 1. Suppose that you want to create your own extension of the binary field with degree 5, and a irreducible polynomial of your choice. Then you can do so as follows:

```
sage: R.<x> = GF(2)[]
sage: F = GF(2).extension(x^5 + x^3 + 1, 'a')
sage: a = F.gen()
```

You need to do this last step to inject the primitive element into the interpreter's name space. This is done automatically when using the GF function to create and extension field, but not when you use the member function extension on a field object.

B.5 CHAPTER 5: ADVANCED ENCRYPTION STANDARD

Example 1. Simplified AES.

#

```
# These structures are the underlying
# Galois Field and corresponding Vector Space
# of the field used in the SAES algorithm
# These structures allow us to easily compute with these fields.
#

F = GF(2);
L.<a> = GF(2^4);
V = L.vector_space();
VF8 = VectorSpace(F, 8);

#
# The MixColumns and its Inverse matrices are stored
# as 2x2 matrices with elements in GF(2^4) (as are state matrices).
# The MixColumns operation (and its inverse) are performed by
# matrix multiplication.
#

MixColumns_matrix = Matrix(L, [[1,a^2],[a^2,1]]);

InverseMixColumns_matrix = MixColumns_matrix.inverse();

SBox_matrix = Matrix(L,
    [
    [       1 + a^3,           a^2,           a + a^3, 1 + a + a^3],
    [ 1 + a^2 + a^3,             1,               a^3,     1 + a^2],
    [       a + a^2,             a,                 0,       1 + a],
    [     a^2 + a^3, a + a^2 + a^3, 1 + a + a^2 + a^3, 1 + a + a^2]
    ]);

InverseSBox_matrix = Matrix(L,
    [
    [   a + a^3,     1 + a^2,       1 + a^3,       1 + a + a^3],
    [         1, 1 + a + a^2,           a^3, 1 + a + a^2 + a^3],
    [   a + a^2,           0,             a,             1 + a],
    [ a^2 + a^3,         a^2, 1 + a^2 + a^3,     a + a^2 + a^3]
    ]);

RCON = [
    VF8([F(0), F(0), F(0), F(0), F(0), F(0), F(0), F(1)]),
    VF8([F(0), F(0), F(0), F(0), F(1), F(1), F(0), F(0)])
       ];

def SAES_ToStateMatrix(block):
    r"""
    Converts a bit list into an SAES State Matrix
    """
    B = block;

    # form the plaintext block into a matrix of GF(2^n) elements
    S00 = L(V([B[0], B[1], B[2], B[3]]));
    S01 = L(V([B[4], B[5], B[6], B[7]]));
    S10 = L(V([B[8], B[9], B[10], B[11]]));
    S11 = L(V([B[12], B[13], B[14], B[15]]));

    state_matrix = Matrix(L, [[S00,S01],[S10,S11]]);

    return state_matrix;
```

```
def SAES_FromStateMatrix(state_matrix):
    r"""
    Converts an SAES state_matrix to a bit list.
    """

    output = [];

    # convert state matrix back into bit list
    for r in xrange(2):
        for c in xrange(2):
            v = V(state_matrix[r,c]);
            for j in xrange(4):
                output.append(Integer(v[j]));

    return output;

def SAES_AddRoundKey(state_matrix, K):
    r"""
    Adds a roundkey to an SAES state matrix.
    """

    K_matrix = SAES_ToStateMatrix(K);

    next_state_matrix = K_matrix + state_matrix;

    return next_state_matrix;


def SAES_MixColumns(state_matrix):
    r"""
    Performs the Mix Columns Operation with
    """
    next_state_matrix = MixColumns_matrix*state_matrix;
    return next_state_matrix;

def SAES_InverseMixColumns(state_matrix):
    r"""
    Performs the Inverse Mix Columns operation
    """
    next_state_matrix = InverseMixColumns_matrix*state_matrix;
    return next_state_matrix;

def SAES_ShiftRow(state_matrix):
    r"""
    Performs the Shift Row operation.
    """
    M = state_matrix;
    next_state_matrix = Matrix(L, [
                                   [M[0,0], M[0,1]],
                                   [M[1,1], M[1,0]]
                                   ]);
    return next_state_matrix;

def SAES_SBox(nibble):
    r"""
    Performs the SAES SBox look up in the SBox matrix (lookup table.)
    """
```

```
    v = nibble._vector_();
    c = Integer(v[0]) + 2*Integer(v[1]);
    r = Integer(v[2]) + 2*Integer(v[3]);
    return SBox_matrix[r,c];

def SAES_NibbleSubstitution(state_matrix):
    r"""
    Performs the SAES SBox on each element of an SAES state matrix.
    """
    M = state_matrix;
    next_state_matrix = Matrix(L,
                          [ [ SAES_SBox(M[0,0]), SAES_SBox(M[0,1])],
                            [ SAES_SBox(M[1,0]), SAES_SBox(M[1,1])] ]);
    return next_state_matrix;

def SAES_InvSBox(nibble):
    r"""
    Performs the SAES Inverse SBox look up in the SBox matrix (lookup table.)
    """
    v = nibble._vector_();
    c = Integer(v[0]) + 2*Integer(v[1]);
    r = Integer(v[2]) + 2*Integer(v[3]);
    return InverseSBox_matrix[r,c];

def SAES_InvNibbleSub(state_matrix):
    r"""
    Performs the SAES Inverse SBox on each element of an SAES state matrix.
    """
    M = state_matrix;
    next_state_matrix = Matrix(L,
                          [ [ SAES_InvSBox(M[0,0]), SAES_InvSBox(M[0,1])],
                            [ SAES_InvSBox(M[1,0]), SAES_InvSBox(M[1,1])] ]);
    return next_state_matrix;

def RotNib(w):
    r"""
    Splits an 8 bit list into two elements of GF(2^4)
    """
    N_0 = L(V([w[j] for j in xrange(4)]));
    N_1 = L(V([w[j] for j in xrange(4,8)]));
    return (N_1, N_0);

def SAES_g(w, i):
    r"""
    Performs the SAES g function on the 8 bit list w).
    """
    (N0, N1) = RotNib(w);
    N0 = V(SAES_SBox(N0));
    N1 = V(SAES_SBox(N1));
    temp1 = VF8( [ N0[0], N0[1], N0[2], N0[3],
                   N1[0], N1[1], N1[2], N1[3] ] );
    output = temp1 + RCON[i];
    return output;

def SAES_KeyExpansion(K):
    r"""
    Expands an SAES Key into two round keys.
    """
```

```
    w0 = VF8([K[j] for j in xrange(8)]);
    w1 = VF8([K[j] for j in xrange(8,16)]);

    w2 = w0 + SAES_g(w1, 0);
    w3 = w1 + w2;

    w4 = w2 + SAES_g(w3, 1);
    w5 = w3 + w4;

    K0 = [w0[j] for j in xrange(8)];
    K0.extend([w1[j] for j in xrange(8)]);

    K1 = [w2[j] for j in xrange(8)];
    K1.extend([w3[j] for j in xrange(8)]);

    K2 = [w4[j] for j in xrange(8)];
    K2.extend([w4[j] for j in xrange(8)]);

    return (K0, K1, K2);


#
# Encrypts one plaintext block with key K
#
def SAES_Encrypt(plaintext, K):
    r"""
    Performs a SAES Encryption on a single plaintext block.
    (Both block and key passed as bit lists.)
    """

    # get the key schedule
    (K0, K1, K2) = SAES_KeyExpansion(K);

    state_matrix0 = SAES_ToStateMatrix(plaintext);

    state_matrix1 = SAES_AddRoundKey(state_matrix0, K0);

    state_matrix2 = SAES_NibbleSubstitution(state_matrix1);

    state_matrix3 = SAES_ShiftRow(state_matrix2);

    state_matrix4 = SAES_MixColumns(state_matrix3);

    state_matrix5 = SAES_AddRoundKey(state_matrix4, K1);

    state_matrix6 = SAES_NibbleSubstitution(state_matrix5);

    state_matrix7 = SAES_ShiftRow(state_matrix6);

    state_matrix8 = SAES_AddRoundKey(state_matrix7, K2);

    output = SAES_FromStateMatrix(state_matrix8);

    return output;


#
# Decrypts one ciphertext block with key K
```

```
#
def SAES_Decrypt(ciphertext, K):
    r"""
    Performs a single SAES_Decrypt operation on a ciphertext block.
    (Both block and key passed as bit lists.)
    """

    # perform key expansion
    (K0, K1, K2) = SAES_KeyExpansion(K);

    # form the ciphertext block into a matrix of GF(2^n) elements
    state_matrix0 = SAES_ToStateMatrix(ciphertext);

    state_matrix1 = SAES_AddRoundKey(state_matrix0, K2);

    state_matrix2 = SAES_ShiftRow(state_matrix1);

    state_matrix3 = SAES_InvNibbleSub(state_matrix2);

    state_matrix4 = SAES_AddRoundKey(state_matrix3, K1);

    state_matrix5 = SAES_InverseMixColumns(state_matrix4);

    state_matrix6 = SAES_ShiftRow(state_matrix5);

    state_matrix7 = SAES_InvNibbleSub(state_matrix6);

    state_matrix8 = SAES_AddRoundKey(state_matrix7, K0);

    output = SAES_FromStateMatrix(state_matrix8);

    return output;
```

B.6 CHAPTER 6: PSEUDORANDOM NUMBER GENERATION AND STREAM CIPHERS

Example 1. Blum Blum Shub RNG

```
def BlumBlumShub_Initialize(bitlen, seed):
    r"""
    Initializes a Blum-Blum-Shub RNG State:

    A BBS-RNG State is a list with two elements:
    [N, X]
    N is a 2*bitlen modulus (product of two primes)
    X is the current state of the PRNG.

    INPUT:

        bitlen - the bit length of each of the prime factors of n

        seed - a large random integer to start out the prng
```

```
    OUTPUT:

        state - a BBS-RNG internal state

    """

    # note that may not be the most cryptographically secure
    # way to generate primes, because we do not know how the
    # internal sage random_prime function works.

    p = 3;
    while (p < 2^(bitlen-1)) or (3 != (p % 4)):
        p = random_prime(2^bitlen);

    q = 3;
    while (q < 2^(bitlen-1)) or (3 != (q % 4)):
        q = random_prime(2^bitlen);

    N = p*q;

    X = (seed^2 % N)

    state = [N, X]

    return state;


def BlumBlumShub_Generate(num_bits, state):
    r"""
    BlumBlumShum Random Number Generation function

    INPUT:

        num_bits - the number of bits (iterations) to generate with this RNG.

        state - an internal state of the BBS-RNG (a list [N, X])

    OUTPUT:

        random_bits - a num_bits length list of random_bits

    """

    random_bits = [];

    N = state[0]
    X = state[1]

    for j in xrange(num_bits):

        X = X^2 % N
        random_bits.append(X % 2)

    # update the internal state
    state[1] = X;

    return random_bits;
```

Example 2. Linear Congruential RNG

```
def LinearCongruential_Initialize(a, c, m, X0):
    r"""
    This functional initializes a Linear Congruential RNG state.

    This state is a list of four integers: [a, c, m, X]

    a,c,m are the parameters of the linear congruential instantiation
    X is the current state of the PRNG.

    INPUT:

        a - The coefficient
        c - The offset
        m - The modulus
        X0 - The initial state

    OUTPUT:

        state - The initial internal state of the RNG

    """

    return [a,c,m,X0]


def LinearCongruential_Generate(state):
    r"""
    Generates a single Linear Congruential RNG output and updates the state.

    INPUT:

        state - an internal RNG state.

    OUTPUT:

        X - a single output of the linear congruential RNG

    """

    a = state[0]
    c = state[1]
    m = state[2]
    X = state[3]

    X_next = (a*X + c) % m

    state[3] = X_next

    return X_next
```

B.7 CHAPTER 8: NUMBER THEORY

Example 1. Chinese Remainder Theorem.

```
def chinese_remainder_theorem(moduli, residues):
    r"""

    Function that implements the chinese remainder theorem.

    INPUT:

        moduli - list or positive integers

        residues - list of remainders such that remainder at position j
results when divided by the corresponding modulus at position j in moduli

    OUTPUT:

        x - integer such that division by moduli[j] gives remainder
residue[j]

    """
    if (len(moduli) != len(residues)):
        raise ValueError, "expected len(moduli) == len(residues)"

    M = prod(moduli);

    x = 0;

    for j in xrange(len(moduli)):
        Mj = moduli[j]
        Mpr = M/Mj

        (Mj_Mpr_gcd, Mpr_inv, Mj_inv) = xgcd(Mpr, Mj)

        Mpr_inv = Mpr_inv

        if (Mj_Mpr_gcd != 1):
            raise ValueError, "Expected all moduli are coprime."

        x += residues[j]*Mpr*Mpr_inv;

    return x;
```

Example 2. Miller Rabin Primality Test.

```
r"""
EXAMPLES:

    sage: MILLER_RABIN_TEST(101)
    False

    sage: MILLER_RABIN_TEST(592701729979)
    True

"""

def MILLER_RABIN_TEST(n):
```

```
    r"""

    This function implements the Miller-Rabin Test.

    It either returns "inconclusive" or "composite."

    INPUT:

        n - positive integer to probabilistically determine the primality of.

    OUTPUT:

        If the function returns False, then the test was inconclusive.
        If the function returns True, then the test was conclusive and n is
composite.

    """

    R = IntegerModRing(n); # object for integers mod n

    # (1) Find integers k, q w/ k > 0 and q odd so that (n-1) == 2^k * q
    q = n-1
    k = 0
    while (1 == (q % 2)):
        k += 1
        q = q.quo_rem(2)[0] # q/2 but with result of type Integer

    # (2) select random a in 1 < a < n-1
    a = randint(1,n-1)

    a = R(a) # makes it so modular exponentiation is done fast

    # if a^q mod n == 1 then return inconclusive
    if (1 == a^q):
        return False

    # (3) for j = 0 to k-1 do: if a^(2^j * q) mod n = n-1 return inconclusive
    e = q
    for j in xrange(k):
        if (n-1) == (a^e):
            return False
        e = 2*e

    # (4) if you've made it here return composite.
    return True
```

Example 3. Modular Exponentiation (Square and Multiply).

```
def ModExp(x,e,N):
    r"""
    Calculates x^e mod N using square and multiply.

    INPUT:

        x - an integer
        e - a nonnegative integer
```

```
        N - a positive integer modulus

    OUTPUT:

        y - x^e mod N

    """

    e_bits = e.bits()
    e_bitlen = len(e_bits)

    y = 1

    for j in xrange(e_bitlen):

        y = y^2 % N

        if (1 == e_bits[e_bitlen-1-j]):
            y = x*y % N

    return y
```

Example 4. Using built-in Sage functionality for CRT.

Sage has built in functions to perform the Chinese Remainder Theorem. There are several functions that produce a wide array of CRT functionality. The simplest function performs the CRT with two modulii. Specifically CRT (or the lowercase crt) when called as:

```
crt(a,b,m,n)
```

will return a number that is simultaneously congruent to a mod m and by mod n. All parameters are assumed to be Integers and the parameters m, n must be relatively prime. Some examples of this function are:

```
sage: CRT(8, 16, 17, 49)
-3120

sage: CRT(1,2,5,7)
16

sage: CRT(50,64,101,127)
-62166
```

If you want to get perform the CRT with a list of residues and moduli, Sage includes the function CRT_list.

```
CRT_list(v, modulii)
```

requires that v and modulii be lists of Integers of the same length. Furthermore, the elements of modulii must be relatively prime. Then the output is an integer that reduces to v[i] mod modulii[i] (for i in range(len(v))). For example, the last call to CRT would have been

```
sage: CRT_list([50,64],[101,127])
1969
```

Note that this answer is different. However, you can check that both answers satisfy the requirements of the CRT. Here are examples with longer lists

```
sage: CRT_list([8, 20, 13], [49, 101, 127])
608343

sage: CRT_list([10,11,12,13,14],[29,31,37,41,43])
36657170
```

The function CRT_basis can be used to precompute the values associated to the given set of modulii. If modulii is a list of relatively prime modulii, then CRT_basis(modulii) returns a list a. This list a is such that if x is a list of residues of the modulii, then the output of the CRT can be found by summing:

```
a[0]*x[0] + a[1]*x[1] + ... + a[len(a)-1]*x[len(a)-1]
```

In the case of the modulii used in the last call to CRT_list this function returns as follows:

```
sage: CRT_basis([29,31,37,41,43])
[32354576, 20808689, 23774055, 17163708, 23184311]
```

The last CRT function that Sage provides is CRT_vectors. This function performs CRT_list on several different lists (with the same set of modulii) and returns a list of the simultaneous answers. It is efficient in that it uses CRT_basis and does not recompute those values for each list. For example:

```
sage:
CRT_vectors([[1,10],[2,11],[3,12],[4,13],[5,14]],[29,31,37,41,43
])
[36657161, 36657170]
```

Example 5. Using built-in Sage functionality for Modular Exponentiation.

Sage can perform modular exponentiation using fast algorithms (like square and multiply) and without allowing the intermediate computations to become huge. This is done through IntegerModRing objects. Specifically, creating an IntegerModRing object indicates that arithmetic should be done with a modulus. Then you cast your integers in this ring to indicate that all arithmetic should be done with the modulus. Then for elements of this ring, exponentiation is done efficiently. For example:

```
sage: R = IntegerModRing(101)
sage: x = R(10)
sage: x^99
91

sage: R = IntegerModRing(1024)
sage: x = R(111)
sage: x^345
751
```

```
sage: x = R(100)
sage: x^200
0

sage: N = 127*101
sage: R = IntegerModRing(N)
sage: x = R(54)
sage: x^95
9177
```

Creating an IntegerModRing is similar to creating a FiniteField with GF(...) except that the modulus can be a general composite.

Example 6. Using built-in Sage functionality for Euler's totient.

Sage has the Euler Totient functionality built in. The function is called euler_phi because of the convention of using the Greek letter phi to represent this function. The operation of this function is simple. Just call euler_phi on an integer and it computes the totient function. This function factors the input, and hence requires exponential time.

```
sage: euler_phi(101)
100

sage: euler_phi(1024)
512

sage: euler_phi(333)
216

sage: euler_phi(125)
100

sage: euler_phi(423)
276
```

B.8 CHAPTER 9: PUBLIC-KEY CRYPTOGRAPHY AND RSA

Example 1. Using Sage we can simulate an RSA Encryption and Decryption.

```
sage: # randomly select some prime numbers
sage: p = random_prime(1000); p
191
sage: q = random_prime(1000); q
601
sage: # compute the modulus
sage: N = p*q
sage: R = IntegerModRing(N)
sage: phi_N = (p-1)*(q-1)
sage: # we can choose the encrypt key to be anything
sage: # relatively prime to phi_N
sage: e = 17
sage: gcd(d, phi_N)
1
sage: # the decrypt key is the multiplicative inverse
sage: # of d mod phi_N
sage: d = xgcd(d, phi_N)[1] % phi_N
sage: d
60353
sage: # Now we will encrypt/decrypt some random 7 digit numbers

sage: P = randint(1,127); P
97
sage: # encrypt
sage: C = R(P)^e; C
46685
sage: # decrypt
sage: R(C)^d
97

sage: P = randint(1,127); P
46
sage: # encrypt
sage: C = R(P)^e; C
75843
sage: # decrypt
sage: R(C)^d
46

sage: P = randint(1,127); P
3
sage: # encrypt
sage: C = R(P)^e; C
```

```
288
sage: # decrypt
sage: R(C)^d
3
```

Also, Sage can just as easily do much larger numbers:

```
sage: p = random_prime(1000000000); p
114750751
sage: q = random_prime(1000000000); q
8916569
sage: N = p*q
sage: R = IntegerModRing(N)
sage: phi_N = (p-1)*(q-1)
sage: e = 2^16 + 1
sage: d = xgcd(e, phi_N)[1] % phi_N
sage: d
237150735093473

sage: P = randint(1,1000000); P
955802
sage: C = R(P)^e
sage: R(C)^d
955802
```

Example 2. In Sage, we can also see an example of RSA signing/verifying.

```
sage: p = random_prime(10000); p
1601
sage: q = random_prime(10000); q
4073
sage: N = p*q
sage: R = IntegerModRing(N)
sage: phi_N = (p-1)*(q-1)
sage: e = 47
sage: gcd(e, phi_N)
1
sage: d = xgcd(e,phi_N)[1] % phi_N
sage: # Now by exponentiating with the private key
sage: # we are effectively signing the data
sage: # a few examples of this

sage: to_sign = randint(2,2^10); to_sign
650
sage: # the signature is checked by exponentiating
```

```
sage: # and checking vs the to_sign value
sage: signed = R(to_sign)^d; signed
2910116
sage: to_sign == signed^e
True

sage: to_sign = randint(2,2^10); to_sign
362
sage: signed = R(to_sign)^d; signed
546132
sage: to_sign == signed^e
True

sage: # we can also see what happens if we try to verify a bad
signature
sage: to_sign = randint(2,2^10); to_sign
605
sage: signed = R(to_sign)^d; signed
1967793
sage: bad_signature = signed - randint(2,100)
sage: to_sign == bad_signature^e
False
```

B.9 CHAPTER 10: OTHER PUBLIC-KEY CRYPTOSYSTEMS

Example 1. Here is an example of Alice and Bob performing a Diffie-Hellman Key Exchange done in Sage:

```
sage: # Alice and Bob agree on the domain parameters:
sage: p = 619
sage: F = GF(p)
sage: g = F(2)
sage: # Alice picks a random value x in 1...618
sage: x = randint(1,618); x
571
sage: # Alice computes X = g^x and sends this to Bob
sage: X = g^571; X
591
sage: # Bob picks a random value y in 1...618
sage: y = randint(1,618);y
356
sage: # Bob computes Y = g^y and sends this to Alice
sage: Y = g^y; Y
```

```
199
sage: # Alice computes Y^x
sage: Y^x
563
sage: # Bob computes X^y
sage: X^y
563
sage: # Alice and Bob now share a secret value
```

Example 2. In reality to prevent what is known as small subgroup attacks,

the prime p is chosen so that $p = 2q + 1$ where q is a prime as well.

```
sage: q = 761
sage: p = 2*q + 1
sage: is_prime(q)
True
sage: is_prime(p)
True
sage: F = GF(p)
sage: g = F(3)
sage: g^q
1
sage: # note that g^q = 1 implies g is of order q
sage: # Alice picks a random value x in 2...q-1
sage: x = randint(2,q-1); x
312
sage: # Alice computes X = g^x and sends it to Bob
sage: X = g^x; X
26
sage: # Bob computes a random value y in 2...q-1
sage: y = randint(2,q-1); y
24
sage: # Bob computes Y = g^y and sends it to Alice
sage: Y = g^y; Y
1304
sage: # Alice computes Y^x
sage: Y^x
541
sage: # Bob computes X^y
sage: X^y
541
sage: # Alice and Bob now share the secret value 541
```

Example 3. Sage has a significant amount of support for elliptic curves. This functionality can be very useful when learning, because it allows you to easily calculate things and get the big picture, while doing the examples by hand would cause you to get mired in the details. First you instantiate an elliptic curve, by specifying the field that it is over, and the coefficients of the defining Weierstrass equation. For this purpose, we write the Weierstrass equation as

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

Then the Sage function EllipticCurve(R, [a1, a2, a3, a4, a6]) creates the elliptic curve over the ring R.

```
sage: E = EllipticCurve(GF(17), [1,2,3,4,5])
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Finite
Field of size 17

sage: E = EllipticCurve(GF(29), [0,0,0,1,1])
sage: E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 29

sage: E = EllipticCurve(GF(127), [0,0,0,2,17])
sage: E
Elliptic Curve defined by y^2 = x^3 + 2*x + 17 over Finite Field of size 127

sage: F.<theta> = GF(2^10)
sage: E = EllipticCurve(F, [1,0,0,1,0])
sage: E
Elliptic Curve defined by y^2 + x*y = x^3 + x over Finite Field in theta of
size 2^10
```

Example 4. Koblitz curves. A Koblitz curve is an elliptic curve over a binary field defined by an equation of the form

$$y^2 + xy = x^3 + ax^2 + 1$$

where a = 0 or 1. FIPS 186-3 recommends a number of Koblitz curves for use with the Digital Signature Standard (DSS). Here we give an example of a curve of similar form to the Koblitz curves:

```
sage: F.<theta> = GF(2^17)
sage: E = EllipticCurve(F, [1,0,theta,0,1])
sage: E
Elliptic Curve defined by y^2 + x*y + theta*y = x^3 + 1 over
Finite Field in theta of size 2^17
```

Example 5. Sage can even easily instantiate curves of cryptographic sizes, like K163, which is one of the FIPS 186-3 curves.

```
sage: F.<theta> = GF(2^163)
sage: E = EllipticCurve(F, [1,0,1,0,1])
sage: E
Elliptic Curve defined by y^2 + x*y + y = x^3 + 1 over Finite
Field in theta of size 2^163
```

However, you should be warned that when instantiating a curve of cryptographic sizes, some of the functions on the curve object will not work because they require exponential time to run. While you can compute some things with these objects, it is best to leave your experimentation to the smaller sized curves. You can calculate some values of the curve, such as the number of points:

```
sage: E = EllipticCurve(GF(107), [0,0,0,1,0])
sage: E.order()
108
```

You can also determine the generators of a curve:

```
sage: E = EllipticCurve(GF(101), [0,0,0,1,0])
sage: E.gens()
((7 : 42 : 1), (36 : 38 : 1))
```

Note that this output is printed (x : y : z). This is a minor technical consideration because Sage stores points in what is known as "projective coordinates". The precise meaning is not important,

because for non-infinite points the value z will always be 1 and the first two values in a coordinate will be the x and y coordinates, exactly as you would expect. This representation is useful because it allows the point at infinity to be specified as a point with the z coordinate equal to 0:

```
sage: E(0)
(0 : 1 : 0)
```

This shows how you can recognize a point at infinity as well as specify it. If you want to get the x and y coordinates out of a point on the curve, you can do so as follows:

```
sage: P = E.random_point(); P
(62 : 38 : 1)
sage: (x,y) = P.xy(); (x,y)
(62, 38)
```

You can specify a point on the curve by casting an ordered pair to the curve as:

```
sage: P = E((62,-38)); P
(62 : 63 : 1)
```

Now that you can find the generators on a curve and specify points you can experiment with these points and do arithmetic as well. Continuing to use E as the curve instantiated in the previous example, we can set G1 and G2 to be the generators:

```
sage: (G1, G2) = E.gens()
sage: P = E.random_point(); P
(49 : 29 : 1)
```

You can compute the sum of two points as in the following examples:

```
sage: G1 + G2 + P
(69 : 96 : 1)

sage: G1 + P
(40 : 62 : 1)

sage: P + P + G2
(84 : 25 : 1)
```

You can compute the inverse of a point using the unary minus (–) operator:

```
sage: -P
(49 : 72 : 1)

sage: -G1
(7 : 59 : 1)
```

You can also compute repeated point addition (adding a point to itself many times) with the * operator:

```
sage: 13*G1
(72 : 23 : 1)

sage: 2*G2
(9 : 58 : 1)

sage: 88*P
(87 : 75 : 1)
```

And for curves over small finite fields you can also compute the order (discrete log of the point at infinity with respect to that point).

```
sage: G1.order()
10

sage: G2.order()
10

sage: P.order()
10
```

Example 6. Using the Sage elliptic curve functionality to perform a simulated elliptic curve Diffie-Hellman (ECDH) key exchange.

```
sage: # calculate domain parameters
sage: F = GF(127)
sage: E = EllipticCurve(F, [0, 0, 0, 3, 4])
sage: G = E.gen(0); G
(94 : 6 : 1)
sage: q = E.order(); q
122

sage: # Alice computes a secret value x in 2...q-1
sage: x = randint(2,q-1); x
33
sage: # Alice computes a public value X = x*G
sage: X = x*G; X
(55 : 89 : 1)

sage: # Bob computes a secret value y in 2...q-1
sage: y = randint(2,q-1); y
55
sage: # Bob computes a public value Y = y*G
sage: Y = y*G; Y
(84 : 39 : 1)

sage: # Alice computes the shared value
sage: x*Y
(91 : 105 : 1)
sage: # Bob computes the shared value
sage: y*X
(91 : 105 : 1)
```

However, in practice most curves that are used have a prime order:

```
sage: # Calculate the domain parameters
sage: F = GF(101)
sage: E = EllipticCurve(F, [0, 0, 0, 25, 7])
sage: G = E((97,34))
sage: q = E.order()

sage: # Alice computes a secret values x in 2...q-1
sage: x = randint(2,q-1)
sage: # Alice computes a public value X = x*G
sage: X = x*G

sage: # Bob computes a secret value y in 2...q-1
sage: y = randint(2,q-1)
sage: # Bob computes a public value Y = y*G
sage: Y = y*G

sage: # Alice computes the shared secret value
sage: x*Y
(23 : 15 : 1)

sage: # Bob computes the shared secret value
sage: y*X
(23 : 15 : 1)
```

B.10 CHAPTER 11: CRYPTOGRAPHIC HASH FUNCTIONS

Example 1. The following is an example of the MASH hash function in Sage. MASH is a function based on the use of modular arithmetic. It involves use of an RSA-like modulus M, whose bit length affects the security. M should be difficult to factor, and for M of unknown factorization, the security is based in part on the difficulty of extracting modular roots. M also determines the block size for processing messages. In essence, MASH is defined as:

$$H_i = ((x_i \oplus H_{i-1})^2 \text{ OR } H_{i-1}) \pmod{M}$$

where

A = 0xFF00…00

H_{i-1} = the largest prime less than M

x_i = the i^{th} digit of the base M expansion of input n. That is, we express n as a number of

base M. Thus:

$$n = x_0 + x_1M + x_2M^2 + \ldots$$

The following is an example of the MASH hash function in Sage

```
#
# This function generates a mash modulus
# takes a bit length, and returns a Mash
# modulus l or l-1 bits long (if n is odd)
# returns p, q, and the product N
#
def generate_mash_modulus(l):

    m = l.quo_rem(2)[0]

    p = 1
    while (p < 2^(m-1)):
        p = random_prime(2^m)

    q = 1
    while (q < 2^(m-1)):
        q = random_prime(2^m)

    N = p*q

    return (N, p, q)

#
# Mash Hash
# the value n is the data to be hashed.
# the value N is the modulus
# Returns the hash value.
#
def MASH(n, N):

    H = previous_prime(N)

    q = n

    while (0 != q):
      (q, a) = q.quo_rem(N)
      H = ((H+a)^2 + H) % N

    return H
```

The output of these functions running;

```
sage: data = ZZ(randint(1,2^1000))
sage: (N, p, q) = generate_mash_modulus(20)
sage: MASH(data, N)
220874
sage: (N, p, q) = generate_mash_modulus(50)
sage: MASH(data, N)
455794413217080
sage: (N, p, q) = generate_mash_modulus(100)
sage: MASH(data, N)
268864504538508517754648285037
sage: data = ZZ(randint(1,2^1000))
sage: MASH(data, N)
236862581074736881919296071248
sage: data = ZZ(randint(1,2^1000))
sage: MASH(data, N)
395463068716770866931052945515
```

B.11 CHAPTER 13: DIGITAL SIGNATURES

Example 1. Using Sage, we can perform a DSA sign and verify.

```
sage: # first we generate the domain parameters
sage: # generate a 16 bit prime q
sage: q = 1;
sage: while (q < 2^15): q = random_prime(2^16)
....:
sage: q
42697
sage: # generate a 64 bit p, such that q divides (p-1)
sage: p = 1
sage: while (not is_prime(p)):
....:     p = (2^48 + randint(1,2^46)*2)*q + 1
....:
sage: p
12797003281321319017
sage: # generate h and g
sage: h = randint(2,p-2)
sage: h
5751574539220326847
sage: F = GF(p)
sage: g = F(h)^((p-1)/q)
```

```
sage: g
9670562682258945855

sage: # generate a user public / private key
sage: # private key
sage: x = randint(2,q-1)
sage: x
20499
sage: # public key
sage: y = F(g)^x
sage: y
7955052828197610751

sage: # sign and verify a random value
sage: H = randint(2,p-1)

sage: # signing
sage: # random blinding value
sage: k = randint(2,q-1)
sage: r = F(g)^k % q
sage: r = F(g)^k
sage: r = r.lift() % q
sage: r
6805
sage: kinv = xgcd(k,q)[1] % q
sage: s = kinv*(H  + x*r) % q
sage: s
26026

sage: # Verifying
sage: w = xgcd(s,q)[1]; w
12250
sage: u1 = H*w % q; u1
6694
sage: u2 = r*w % q; u2
16706
sage: v = F(g)^u1 * F(y)^u2
sage: v = v.lift() % q
sage: v
6805
sage: v == r
True

sage: # sign and verify another random value
sage: H = randint(2,p-1)

sage: k = randint(2,q-1)
```

```
sage: r = F(g)^k
sage: r = r.lift() % q
sage: r
3284
sage: kinv = xgcd(k,q)[1] % q
sage: s = kinv*(H  + x*r) % q
sage: s
2330

sage: # Verifying
sage: w = xgcd(s,q)[1]; w
4343
sage: u1 = H*w % q; u1
32191
sage: u2 = r*w % q; u2
1614
sage: v = F(g)^u1 * F(y)^u2
sage: v = v.lift() % q
sage: v
3284
sage: v == r
True
```

Example 2. The following functions implement DSA domain parameter generation, key generation, and DSA Signing.

```
#
# Generates a 16 bit q and 64 bit p, both prime
# such that q divides p-1
#
def DSA_generate_domain_parameters():

    g = 1

    while (1 == g):

        # first find a q
        q = 1
        while (q < 2^15): q = random_prime(2^16)

        # next find a p
        p = 1
        while (not is_prime(p)):
            p = (2^47 + randint(1,2^45)*2)*q + 1
```

```
        F = GF(p)

        h = randint(2,p-1)

        g = (F(h)^((p-1)/q)).lift()

    return (p, q, g)


#
# Generates a users private and public key
# given domain parameters p, q, and g
#
def DSA_generate_keypair(p, q, g):

    x = randint(2,q-1)

    F = GF(p)

    y = F(g)^x
    y = y.lift()

    return (x,y)


#
# given domain parameters p, q and g
# as well as a secret key x
# and a hash value H
# this performs the DSA signing algorithm
#
def DSA_sign(p, q, g, x, H):

    k = randint(2,q-1)

    F = GF(p)

    r = F(g)^k
    r = r.lift() % q

    kinv = xgcd(k,q)[1] % q

    s = kinv*(H + x*r) % q

    return (r, s)
```