APPENDIX C

SAGE EXERCISES

By Dan Shumow
University of Washington

This appendix contains a number of exercises that reinforce cryptographic concepts, organized by the chapter in which those concepts were discussed. All the exercises use Sage. We begin with a discussion of how to get started using Sage and a brief introduction to the syntax and operations.

C.1 GETTING STARTED WITH SAGE

Sage is a free open source program that collects many open source math packages into one easily usable environment.

The following are step-by-step instructions to installing and getting started using Sage for the examples and exercises in this book.[1]

1. Go to http://www.sagemath.org/download/
2. You have two options:
   a. Building from source: If you are well versed in compilers and building software, you can build from source. Select this option.
   b. Installing Binaries: You can install precompiled binaries, the process is different on several different operating systems.

   - **Linux** Download the Linux binaries, download, and follow the instructions in the README file.
   - **Mac OS X** Download the Mac OS X binaries and follow the instructions in the README file.
   - **Windows:** With Windows the process is a little bit more complicated. At the time of printing the only complete option for Sage on Windows requires running ubuntu in a virtual machine. The directions are contained in the windows section of the download. However, copied here for reference they are:
     i) Download the VMWare player: http://www.vmware.com/products/player/ (this is a free download for students / educators.)
     ii) Download the VMWare image from the Sage website and follow the directions in the README file.
     There is also a native port of windows, in progress, at the time of this printing. You can try it and see if it works for your purposes at: http://windows.sagemath.com
3. Once you have Sage installed, On Linux or Mac OS X you can just type Sage from a shell prompt and it will run the interpreter (if you installed the Sage script in the correct location, as in the README files.) On windows, you run Sage by starting the VMWare player to open the Sage virtual hard drive. Once the VMWare player is started, you can use the player to enter data into the command line, you can SSH to your virtual machine (useful for copy and paste functionality) and use the notebook.
4. Sage also has notebook functionality, similar to that of Maple or Mathematica worksheets. This runs through the web browser. On Linux and Mac OS X,

[1]Please note that Sage is an open source package that is constantly under development, and much functionality changes from release to release. If any of the steps in this section do not work, please check http://www.sagemath.org for new up-to-date information.

you start the notebook by typing notebook() from the command prompt, or by running Sage with the -notebook argument. In the VMWare image, this is run by selecting notebook from the login options when the VMWare image starts up.

5. If you wish to execute the Sage examples from Appendix B, you can now download the relevant Sage files.[2] If you are using a Linux or Mac OS X machine, then you just download your files to a folder and run Sage to access them. However, if you are using the VMWare player then you need to get the files into your virtual machine. This can be done using the shared folder's option in VMWare player, or copying the files using wget or scp from inside the virtual machine. You can access the underlying Ubuntu operating system in the Sage virtual machine by selecting the manage option when the VMWare image starts up.
6. As mentioned in step 3, Sage is an interpreted language, and you interact with it through a prompt. However you can also write batch scripts. These files have the suffix .sage and each line is a line that you would type into the interpreter. You can load these into the interpreter by using the "load" and "attach" commands. The command "load" runs the file once. On the other hand, the command "attach" monitors the underlying .sage file and reloads it if there are any changes.
7. The Sage interpreter keeps track of your underlying path, running attach and load is relative to this path. You can change the current path by using 'cd' like you would in a shell. Suppose that you want to run a file example.sage, you can do this by typing:

   ```
   load example.sage
   ```

   While the current directory of the Sage interpreter is the directory containing example.sage.

There is significantly more information and documentation on Sage and how to run it at http://www.sagemath.org/doc. This page includes a tutorial, a reference manual, a Sage programmer's manual, and an installation guide. See this documentation for the most up to date information on Sage. Even more documentation and help is available at http://www.sagemath.org/help.html. Particularly worthwhile is the downloadable book *Sage for Newbies*.

Sage is a rich, powerful facility, and the amount of documentation may seem overwhelming. However, if you study the examples in Appendix B and the discussion in this section and the next, you should be able to write Sage code to solve the problems with little reference to the documentation. Furthermore, any time devoted to learning Sage is a worthwhile investment, because Sage is a general-purpose mathematical tool that you will be able to use throughout your academic and professional career.

[2]All of the Sage code in Appendix B is available online at this book's Web site in .sage files, so that you can load and execute the programs if you wish. See Preface for access information.

C.2 PROGRAMMING WITH SAGE

The following is a basic introduction to Sage programming. Sage is a collection of open source mathematics packages that are loaded into a Python interpreter. Input to Sage is lightly preprocessed and sent to a Python interpreter. Thus, all programming in Sage is essentially just Python programming. Readers familiar with Python can skip much of this section, however Sage does modify the Python environment some, so reading some of the sections especially about numeric data types may be useful. The following are some of the basic Sage programming constructs.[3]

Input to the Interpreter

Python is an interpreted language, so you can interact with the program line by line.

Python is object-oriented, so as long as you are not using built in data types, you can use the following technique to learn about the member functions and variables of the class you are working with. Suppose **`foo`** is a variable of **`FooClass`** type, then typing **`foo.<tab>`**, (<tab> means, hit tab) will auto complete. If you begin typing the name of a variable, hitting tab will list all members that start with what you have already typed, if there is only one such member, it will autocomplete.

Suppose **`foo`** has a member bar, to learn about this member, you can type **`foo.bar?`** and hit enter, this will display documentation on the function bar if it exists. You can also type **`foo.bar??`** and hit enter to display the documentation and the source code.

You can also use the print function to try to print a Sage object. So if **`foo`** is some variable, the expression **`print foo`** will try to convert **`foo`** into a string and print it.

Also the function **`type(foo)`** will return the type of the variable foo, which is also useful.

Data Types

One basic data type is **`Str`**. This is the built-in data type for strings in Python. They are entered into the interpreter by putting a literal string in single quotes `''` or double quotes `""`. These are sequential, and can be accessed as such. For example if **`foo = 'foo'`**, then **`foo[0] = 'f'`** and **`foo[1]='o'`**.

Sage provides the following numeric data types.

- **`int`**: This is the built-in, fixed precision signed integer data type of Python. This is the default data type used to access sequential data (like tuples and lists) as well as the default loop counter.
- **`long`**: This is the built-in, arbitrary precision integer data type of Python. If an operation on operands of type **`int`** overflows, the result will be a **`long`**.
- **`Integer`**: This is a Sage data type that implements arbitrary precision integers. This is the default type ascribed to integers typed into the Sage interpreter. The object for the integers is ZZ. Variables of type **`int`** or **`long`** can be cast to an **`Integer`** as follows, if **`foo`** is an **`int`** or a **`long`** then **`ZZ(foo)`** casts **`foo`** to an **`Integer`**. (**`Integer(foo)`** will work as well.)

[3]Note that Sage does change from release to release, so be sure to check http://www.sagemath.org/doc for the most up to date documentation.

- **`Rational`**: This is the Sage data type that implements arbitrary precision rational numbers. If you type something like 2/3 into the Sage interpreter, this is the default type that the value will be given. The object for the rational numbers is **`QQ`**. Variables of type **`int`**, **`long`**, or **`Integer`** can be cast to a **`Rational`** as follows, if **`foo`** is such a numeric type, then **`QQ(foo)`** casts **`foo`** to a **`Rational.`** (**`Rational(foo)`** will work as well.) This is important to bear in mind, as in many programming languages 2/3 would evaluate to the integer 0.

Two important sequential data types in Sage are **`list`** and **`tuple`**. The **`list`** type is the built-in Sage type for lists (arrays). The syntax for lists is an open and close square brackets, with items separated by commas.

- The empty list is

  ```
  foo = []
  ```

- A list of the first three integers is

  ```
  foo = [1, 2, 3]
  ```

- A list of some different types is

  ```
  foo = ['2/3', 2, 2/3]
  ```

  The first element is a string, the second is an **`Integer`** and the third is a **`Rational`**.

The function **`range([start,] end)`** returns the list of integers from start through **`end-1`**. If start is omitted 0 is assumed. Also, you can initiate a list with a variable number of arguments explicitly by doing:

```
foo = [<expression in j> for j in xrange(M)]
```

Where **`M`** is some integral data type, and **`<expression in j>`** is some Python expression that is allowed to reference the loop counter **`j`**. You access list elements by square brackets after the name of the variable, so **`foo[i]`** returns the *i*th element of a list. Lists are mutable, so you can assign values to the elements as follows: **`foo[i] = bar`**. You can also extend lists by using the append function, as in: **`foo.append(bar)`**. You can get a length of a list with the built in function **`len(...)`**, by calling it as **`len(foo)`**.

The **`tuple`** data type is the built in Sage type for immutable lists of elements. They syntax is like that for lists but parenthesis are used instead of square brackets. As with the examples for lists:

- The empty tuple is

  ```
  foo = ()
  ```

- A tuple of the first three integers is

  ```
  foo = (1, 2, 3)
  ```

- A tuple of some different types is

  ```
  foo = ('2/3', 2, 2/3)
  ```

Tuples are accessed just as lists, so **`foo[i]`** returns the *i*th element of the tuple **`foo`**. As mentioned, tuples are immutable, so you cannot assign values to the elements after the tuple has been initialized.

Mathematical Operators

The usual mathematical operators work in Sage. So, +, −, and * all work as you would expect. As noted above / performs division, but if the operands are integers it promotes them to rational numbers. The % operator performs modular reduction **`a % n`** is the remainder of *a* divided by *n*, for *a* and *n* integral data types. You can accomplish integral division as follows. If *a* is an integer, and you want the quotient and remainder after division by *n*, **`a.quo_rem(n)`** returns a tuple **`(q,r)`** where **`q`** is the quotient and **`r`** is the remainder.

One of the major differences between Sage and Python is that the operator ^ is not xor, as it is in Python. Rather, this means exponentiation. So **`a^n`** is *a* raised to the *n*th power. Alternately, this can be performed with the ** operator as **`a**n`**.

Control Statements

If-else statements are written as follows:

```
if <boolean statement> :
<tab> <block of code>
elif <boolean statement> :
<tab> <block of code>
else:
<tab> <block of code>
```

Where **`<boolean statement>`** indicates a valid statement in Python that evaluates to a Boolean expression, and **`<block of code>`** indicates a multi-line block of Python code. It is important to note that with if statements (and also loop statements), the blocks of code must be indented, and the subsequent control statement must return to the original level of indentation. This is how the interpreter knows how to match if elif and else statements. In the if and elif statements, the Boolean expression must be followed by a semicolon. For else statements the semicolon immediately follows the keyword else. For the Boolean statements the standard operators of Boolean (and, or) are spelled out exactly as 'and' and 'or'. Meaning for a number less than 0 and greater than 3 one writes: **`(x < 0) or (3 < x)`**. For a number greater than 0 and less than 3 one writes: **`(0 < x) and (x < 3)`**.

The syntax for a **for loop** is as follows:

```
for <variable> in <iteratable object>:
<tab> <block of code>
```

In Python **`<iteratable objects>`** are sequential objects like lists or tuples (there are some others as well, but lists and tuples are the main case.) For example

```
for A in foo:
  print A
```

prints the list of the elements in the list **`foo`**. Or

```
for j in range(10):
  print j
```

prints the integers from 1 to 10.

The most common exception to using a list or a tuple as an iterable object is iterating through a list of integers using the **`xrange`** function. For example:

```
for j in xrange(len(foo)):
  print ‘j=’, j, foo[j]
```

prints a list of the indices and the elements at that index in the array.

The **`xrange`** function allows loops to iterate through the integers **`[0, 1, ..., (len(foo)-1)]`** without instantiating the list as the function range would. The function xrange takes the parameters are the same as the range function does, the only difference is the output.

While loops have the syntax:

```
while <boolean expression> :
<tab> <block of code>
```

For example:

```
while ( x < 1):
  y = y + x
  x = x/2
```

With both while and for loops, the **`break`** keyword cause execution of the loop to stop, and **`continue`** causes control to begin executing at the next iteration of the loop.

Functions

Creating functions is very easy:

```
def <function-name>(< comma separated list of parame-
ters>):
<tab> <block of code>
```

Just like control statements, the body of the function must be delimited by indentation. The **`return`** statement specifies the value of the function to **`return`**. If the function does not have a **`return`** statement, or the end of the body of the function is reached without hitting a **`return`** statement, then the function returns the value **`None`**. For example, the following function:

```
def f1(x, y):
  if (0 == x % 2):
    z = x^2 + x + 1
else:
    z = x-y
return y
```

For more information on Python programming, see http://www.Python.org/. At this time Sage uses Python 2.x, and not Python 3.0 or higher. This is not likely to change, but as the differences in the language are significant, if the examples here are not working, it may be worth checking out if the underlying version of Python that Sage uses has changed.

C.3 CHAPTER 2: CLASSICAL ENCRYPTION

2.1 Implement Sage functions that perform affine cipher encryption/decryption, given a key that consists of a pair of integers a, b, both in $\{1, 2, . . , 25\}$ with a not divisible by 2 or 13. The functions should work on strings, and leave any non-alphabetic characters unchanged. Show the operation of your functions on an example. See problem 2.1 in Chapter 2 for a definition of an affine cipher.

2.2 This question is to implement some functions useful to performing classical cipher attacks.

a. Implement a Sage function that performs frequency attacks on a mono-alphabetic substitution ciphers. This function should take a ciphertext string, compute a histogram of the incidence of each letter (ignoring all non alphabet characters), and return a list of pairs (letter, incidence percentage) sorted by incidence percentage.

b. Implement a Sage function that takes a partial mono-alphabetic substitution and a ciphertext and returns a potential plaintext. The partial mono-alphabetic substitution should be specified as follows: As a 26 character string where the character at position i is the substitution of ith character of the alphabet, OR an underscore '_' if the corresponding substitution is unknown. The potential plaintext should be the ciphertext with values specified by the mono-alphabetic substitution replaced by the lower-case plaintext. If the corresponding character is unknown (i.e. '_' in the monoalphabetic substitution cipher) print the cipher text as an uppercase character.)

c. Use your functions from (a) and (b) to decrypt the following ciphertext: "ztmn pxtne cfa peqef kecnp cjt tmn zcwsenp ontmjsw ztnws tf wsvp xtfwvfefw, c feb fcwvtf, xtfxevqea vf gvoenwk, cfa aeavxcwea wt wse rntrtpvwvtf wscw cgg lef cne xnecwea eymcg."

2.3 Implement Sage functions to perform encryption/decryption with 2×2 Hill Cipher. The key should be an invertible Sage matrix over the integers mod 26.

Do not just call the built in Sage functionality for the Hill cipher. Show the operation of your functions on a plaintext of your choice.

2.4 Implement a Sage function to perform encryption/decryption with an $m \times m$ Hill Cipher. The key should be an invertible Sage matrix over the integers mod 26. Do not just call the built in Sage functionality for the Hill cipher. Show the operation of your function on the functions you write on a plaintext of your choice. You may use any functions you wrote for the previous question to answer this question.

2.5 This question is to implement and use a known plaintext attack on the Hill cipher. You may use functions from examples or previous questions, but do not use the built in Sage functions for the Hill Cipher. [Hint: The built in functions for MatrixSpace and FreeModule objects may be useful, but if they are too confusing to use, do not get caught up on them.]

a. Implement a known plaintext attack on the hill cipher.

b. Use the function that you wrote in part (a) to attack the following plaintext/ciphertext pairs:
plaintext = "friday" ciphertext = "izrvey"
plaintext = "diamondisinstatue" ciphertext = "zisxlhdiwdingthyqq"
plaintext = "thesecretdietistofuhotdogs" ciphertext = "qbayzelwilksscipqps vkafvssyy"

C.4 CHAPTER 3: BLOCK CIPHERS AND THE DATA ENCRYPTION STANDARD

3.1 This problem references the Sage implementation from Appendix B.3 Example 1.

a. Copy the diagram of the function f_K of the Simplified DES Encryption details (Figure G.3 in Appendix G) and label each wire with the corresponding variable name from the Sage code that implements SDES Encryption.

b. Copy the diagram of the Simplified DES Encryption Key Generation (Figure G.2) and label each wire with the corresponding variable names from the Sage code that implements the SDES Key Generation.

3.2 a. Let temp_block denote a Sage variable that contains the output of the first application of function f_K ($\mathrm{f_K}$ in the Sage example code) while encrypting with Simplified DES. Using subroutines from the example Sage code, write Sage code to recover the input block passed to Simplified DES Decrypt. That is, reverse the first steps in Simplified DES Encrypt. You may assume that you have the first round key in a variable K1.

b. Using subroutines from the Sage example code for Simplified DES, write a function to compute Simplified DES Decrypt.

3.3 a. Consider EP, the expansion permutation. Find an inverse contraction permutation. That is, find a function that takes 8 bits down to 4 and inverts EP. Note that these are not unique. Implement this function EPinv as in the example Sage code.

b. Take the function f_K from the example Sage code and modify it so that instead of calling the SBoxes, it calls EPinv after the round key is XORed in. Rename the modified function f_K_NoSBox.

c. Modify the functions SDESEncrypt [see example Sage code], and SDES-Decrypt (see question 3.1] so that they then call f_K_NoSBox (from part b). Call the new functions SDESEncryptNoSBox and SDESDecryptNoSBox.

d. Do these new functions function as Encrypt/Decrypt functions of each other? (i.e. will SDESDecryptNoSBox give you back the input of SDESEncryptNoSBox, given that they are using the same key)?

e. Does SDESEncryptNoSBox make a good Encryption function, why or why not? *Hint:* Can you mount a known or chosen plaintext attack on the functions you wrote in part (d)?

C.5 CHAPTER 4: BASIC CONCEPTS IN NUMBER THEORY AND FINITE FIELDS

4.1 In the examples functions for the Euclidean and extended Euclidean GCD, the first input must be greater than the second. Furthermore, each argument must be a positive integer. Implement these functions such that these assumptions need not be made about the input. Also for the extended Euclidean GCD, if the gcd of a and b is 1, return a inverse mod b and b inverse mod a (Your Sage functions may call the example Sage functions, or you may write these implementations from scratch. Do not merely call the built in Sage functionality.) Show your functions work on a few inputs.

4.2 Suppose that polynomials are represented by lists of coefficients, where the coefficient at index i is the coefficient of x^i. Using this representation, write Sage functions that perform the following polynomial operations (don't just call the underlying Sage functions):

a. Scalar multiply, given a scalar c, and a polynomial f, computes $c \cdot f$.

b. Addition, given two polynomials f, g, computes $h = f + g$.

c. Subtraction, given two polynomials f, g computes $h = f - g$.

d. Multiplication, given two polynomials f, g computes $h = f^*g$.

e. For each of the above functions that you wrote show the output of the function on 1 set of inputs.

4.3 Either using the functions you wrote in the preceding question or the built in polynomial arithmetic in Sage, as well as, either the given polynomial extended gcd, or the built in Sage extended gcd, implement a four function calculator for GF(24) with modulus $x^4 + x + 1$. Consider elements of GF(2^4) to be degree 4 (fixed precision) polynomials in the primitive element, i.e., GF(2^4), elements are represented by lists of 4 binary values. You may use the underlying polynomial functions in Sage, or any functions you wrote for the previous questions.

a. addition

b. scalar multiplication

c. multiplication

d. inversion

4.4 This question asks about using the Sage functionality for computing in Finite Fields.

a. Use Sage to create a finite field with 17 elements. In this field calculate:
The difference: $13 - 16$

The sum: 11 + 10
The quotient:1/2
The product: 3 * 8
The multiplicative inverse of: 5

b. Use Sage to create a finite field with 32 elements. Let 'a' denote the primitive element. In this field Calculate:
The difference: (a^2 + a) − (a + 1)
The multiplicative inverse of: a^4 + a + 1
The quotient (a^2 + 1)/(a^4 + a + 1)

c. Use Sage to create a finite field with 5^3 elements. Let 'alpha' denote the primitive element. In this field Calculate:
The sum: (3*alpha^2 + 4*alpha) − (alpha^2 + 3)
The multiplicative inverse of: (alpha + 1)
The product: (alpha + 2)*(alpha + 3)

d. Use Sage to create a finite field with 503,777,509 elements. In this field calculate:
The quotient: 123,456,789/456,555,333
The multiplicative inverse of : 987,654,321
The difference: 789,123,456 - 444,333,111

4.5 This question is to use the built-in gcd functionality of Sage.

a. Using the gcd functionality in Sage, compute the greatest common divisor of
24 and 300
4567 and 4731907
100 and 1015

b. Using the xgcd functionality in Sage, compute the extended greatest common divisor
36 and 624
4321 and 9226177
45 and 12345

c. Find two numbers, both greater than 100,000 that have a greatest common divisor of exactly 3.
Show the output of Sage that verifies your answer is correct.

4.6 The purpose of this question is to show familiarity with the Sage polynomial arithmetic functionality.

a. Use Sage to initialize a polynomial ring over the field with two elements. Let f = x^2 + 1 and g = x^3 + x^2 + x + 1. Compute f + g, f *g, the quotient and remainder of g divided by f, and the greatest common divisor of f and g.

b. Use Sage to initialize a polynomial ring over the field with 31 elements. Let f = x^5 + 17*x + 13 and g = x^3 + 10*x^2 + 24*x + 3. Compute f − g, f*g, the quotient and remainder of dividing f by g, and the greatest common divisor of f and g.

c. Use the GF(. . .) function to initialize a finite field with 16 elements, and suppose that a is the generator of this field. Then initialize a polynomial ring over this field. Compute the quotient and remainder of dividing
(a^3 + a + 1)x^4 + a*x^2 + (a^2 + a)*x + (a + 1) by (a)x^2 +
(a^3 + 1)*x + a^2

d. In Sage initialize a degree 3 extension of the finite field with 5 elements with defining polynomial x^3 + x + 4. Further suppose that 'theta' is the primitive element of this field. Compute 1/theta.

C.6 CHAPTER 5: ADVANCED ENCRYPTION STANDARD

5.1 The purpose of this question is to become more familiar with the algorithm for generating the Simplified AES S-Box. Each part of this problem is to write one part of the algorithm in Sage, and in the last part put them all together. The construction closely follows the description of the algorithm specification in the text.

a. Consider the positive integers between 0 and 15 (inclusive) as 4 bit strings, so that 3 is 1100 (this ordering is known as little endian.) Define the mapping from {0, 1, 2, . . . , 15} to GF(2^4) by mapping the element with bit string $b_0 b_1 b_2 b_3$ to the element $b_0 + b_1a + b_2a^2 + b_3a^3$ of GF(2^4). The following snippet of Sage code sets F to the finite field with two elements L the finite field with 16 elements (extension of F with modulus $a^4 + a + 1$) and primitive element a (we use a here because a is a special value in Sage.) And V is the vector space of dimension 4 over F (you can think of this as 4 bit strings with addition defined on them.)

```
F = GF(2);
L.<a> = GF(2^4);
V = L.vector_space();
```

As in the example code for Simplified AES we can map a bit list b to the corresponding element of L by L(V(b)). Write a Sage function that maps a positive integer in {0, 1, 2, . . . , 15} to an element of L. (Hint: If x is a Sage Integer, z.bits() is a little-endian list of the bits of z, however z only has as many elements as the bit length of z. So, for example if z is 0, this function returns an empty list. However, L(V(b)) only works if b is a bit list of length 4. You will have to work around this.)

b. Use the function from part (a) to write a Sage function to initialize a 2 dimensional array (either a list of lists or a matrix over L) so that the element at position (r, c) is the element of GF(2^4) mapped to by $4r + c$.

c. Write a function that takes M, a 2-dimensional array of elements in L = GF(2^4) (either a list of lists or a matrix over L) and maps each nonzero element to its inverse and 0 to 0. That is, your function should return M', the 2-dimensional array of elements in L where the element at row r and column c of M' is the inverse of M_{rc} (or 0 if $M_{rc} = 0$). (Hint: If z is a nonzero element of L, in Sage, z^(-1) is the multiplicative inverse of z. If z is zero, this will raise an error.)

d. Write a function that takes a 2-dimensional array of elements in L = GF(2^4) and for each element converts it to an element of V and applies the Linear transformation in step 4 of the S-Box generation algorithm. Then returns the resulting 2-dimensional array. The Sage code to initialize A and b would be:

```
A = Matrix(F, [
        [ 1, 0, 1, 1],
        [ 1, 1, 0, 1],
        [ 1, 1, 1, 0],
        [ 0, 1, 1, 1] ]);
b = V([1, 0, 0, 1]);
```

And the linear equation is A*v + b. Then take the resulting element of V and map it back to an element of L (if v is an element of V, then L(v) is the corresponding element of L.) (Hint: If z is an element of L and v = z.vector(), the linear transformation as is defined in the algorithm expects that the bits of v are reverse of how Sage orders them. To deal with this, you will either have to reverse the bits of v, or appropriately modify the A matrix.)

e. Use the functions you just wrote to write a function that initializes the SBox matrix for simplified DES. Check your answer versus the SBoxes of the simplified AES function in the book.

5.2 In the previous question we computed the SBox for Simplified DES. There are multiple ways to compute the inverse SBox. You can find each element of L in the SBox and figure out which element maps to it. Or you can reverse each of the steps in the previous algorithm. Write a Sage function to calculate the inverse SBox matrix.

5.3 The algorithm for computing the Simplified AES SBox table does exactly that, it computes a table. However, this algorithm shows us how we can compute the SBox directly, without doing a table look up. Write a Sage function to compute the Simplified AES SBox and Inverse SBox directly. Meaning, write functions that take elements of L and return the element of L that the SBox table (or Inverse SBox) lookup would map to. (This is more than a textbook exercise. Some people consider the AES SBox lookups to be insecure because they can leak information through the cache. Such vulnerabilities are called Side Channels. Computing SBoxes without lookups is one way to mitigate this type of attack. Although there is a conditional statement in this SBox computation, which could be exploited by a side channel attack.)

C.7 CHAPTER 6: PSEUDORANDOM NUMBER GENERATION AND STREAM CIPHERS

6.1 (With regard to the code for this exercise, see www.pearsonhighered.com/stallings) Breaking Blum Blum Shub is provably (polynomial time) equivalent to factoring. While this question does not prove this, it does show how to create a Sage function that gives considerable evidence for this fact. Specifically, we will show that given a function that gives you the previous Blum Blum Shub state from a Blum Blum Shub state, that we can write a probabilistic program that factors. The following function will break Blum Blum Shub (for small N):

```
def previous_BBS_state(state):
    r"""
    This function returns the previous Blum-
BlumShub state.
```

```
    Note that this is a toy function and will only
work on small N.
"""

    N = state[0];
    R = IntegerModRing(N);
    X = R(state[1]);

    if (not X.is_square()):
        print  "Not  a  valid  Blum-Blum-Shub  RNG
        state."
        return None

    return [N, X.sqrt().lift()];
```

a. The first part of the problem is to notice that if you have integers x, y such that $x \neq \pm y \pmod N$ and $(x^2 - y^2) = 0 \bmod N$, then the usual difference of squares equation gives that $(x^2 - y^2) = (x - y)(x + y)$. And so we can hope that gcd $(x - y, N)$ or gcd $(x + y, N)$ yield a nontrivial factor of N. Write a Sage function that takes x,y such that $x \neq \pm y \pmod N$ and $(x^2 - y^2) = 0 \bmod N$ and tries to find a nontrivial factor of N.

b. Using the function you wrote in part (a) and the supplied function previous_BBS_state, write a function that takes a number N (that is a product of two primes p, q both congruent to 3 mod 4) and returns the factors p and q. [Hint: you have to create your own BBS state, so you will have to choose your square. How do you choose a square such that you know you have x, y?]

6.2 Write a Sage function that takes 3 successive outputs from a linear congruential RNG, as well as the modulus m of the internal state, that returns a and c OR indicates that it cannot find these values. Generate a linear congruential state, and 3 successive outputs and show your function working.

6.3 The purpose of this function is to become more familiar with the ANSI X9.17 PRNG. For this problem you may use any solutions to other problems, or example code.

a. Implement a function for a variant of the ANSI X9.17 PRNG using the simplified DES block encrypt, instead of two key triple-DES. Your function should take the current state (the seed, V, and the date/time, DT, variables as 8 bit long bit lists) as well as the SDES key as a 10 bit long bit list. Note that because this function uses SDES, instead of two key triple DES, you do not need two keys.

b. Implement a function for ANSI X9.17 PRNG using the simplified AES block encrypt. Your function should take the current state (the seed, V, and the date/time, DT, variables as 16 bit long bit lists) as well as a key as 16 bit long bit lists. Note that because this function uses SAES, instead of two key triple DES, you do not need two keys.

C.8 CHAPTER 8: NUMBER THEORY

8.1 Write a Sage function to implement Euler's Totient function [Hint: You may find the Sage "factor" function useful here.]

8.2 Note that the sample code for the Miller Rabin test returns True if the test finds, conclusively, that n is composite, otherwise the function returns False to indicate that the function did not find anything conclusively. As noted in this book, we can decide with high probability if n is prime or composite if we run this test multiple times. This exercise is to implement a version of the Miller Rabin test that does so.

a. Implement a function that performs the "witness procedure" of Miller Rabin, that is, the code that checks whether or not an integer in $\{1, 2, \ldots, n - 1\}$ has the specified properties.

b. Use the function that you wrote for part (a) to implement a function that takes a positive integer $n(> 2)$, and a list of integers in $\{1, 2, \ldots, n - 1\}$ and performs the "witness procedure" on each one. If any one of these determines that a is composite, then return False (to indicate a is composite) otherwise return True, to indicate that (with high probability) n is prime.

8.3 Previously we saw that factoring can be reduced to breaking Blum Blum Shub's security. In this problem we will see the other direction, namely that Blum Blum Shub's security can be broken by factoring. For this problem you may use the fact that finding a square root mod a prime is a solved problem. In fact if $p \equiv 3 \pmod 4$ and x is a square mod p then the square root of x is given by $x^{(p+1)/4} \bmod p$. Either use this formula or the built in Sage functionality to compute square roots in a prime field to write a function that takes a Blum Blum Shub internal state, and the two prime factors p, q (both 3 mod 4) of the modulus, and outputs a list of at most 4 possibilities for the previous Blum Blum Shub State. Generate a Blum Blum Shub state (with your own p, q) and show that your function works. [Hint: use the CRT.]

8.4 Sage has a command "time" that works similar to "print." Specifically "time <expr>" runs the expression <expr> and displays some timing information. This exercise is to use this time command to try some experiments timing modular exponentiation with different parameters. For varying values of m and n (positive integers) generate a prime p at most 2^m, and a random positive integer a less than p, then time calculating ModExp (a, e, p), for $e = 2^n$ and $2^n - 1$. (See Appendix B.7 Example 3.) For each different value of e run the experiment several times. Try this with at least two different values for (m, n). Be sure to try varying the sizes of these parameters drastically (i.e. on the order of 10s and 100s.) What do you notice? What does this tell you? [Hint: If you do the experiment correctly, you should make an observation that forms the basis for side channel attacks, a powerful type of attack on crypto systems.]

8.5 The purpose of this question is to show how to generalize the Square and Multiply Exponentiation method to different radixes besides 2. This approach to modular exponentiation is known as a "fixed window" exponentiation.

a. Write a Sage function that, given an integer x, a modulus N, and a base b, computes a list of length b, where the ith element of the list is $x^i \bmod N$. You may use the ModExp function or any other method to compute the exponentiation (but you don't have to.) (See Appendix B.7 Example 3.)

b. Write a Sage function that takes an integer x, an exponent e, a base b, and a modulus N. This function should compute a power table using the function you wrote in part (a) and then use it by using the base b expansion of e to determine where to index into the table. You may use modular exponentiation, but only to calculate y^b mod N, for any integer y.

c. Given that you use ModExp as a routine in the function you wrote in part (b) what can you conclude about the optimal base to use for modular exponentiation?

8.6 Suppose we want to create a Random Number Generator with hardness based on the Discrete Log problem. In this problem we will investigate such an RNG and show that it has some weaknesses. First, suppose that we have primes P, Q such that $P = 2 \cdot Q + 1$. Now suppose that we have two points X and Y with multiplicative order Q mod P. This means that $X^Q \equiv Y^Q \equiv 1$ mod P. Let s[i] denote the value of the internal state at time i. We generate the following values as follows:

The intermediate data value: t[i] = s[i]

The next internal state s[i + 1] = $X^{t[i]}$ mod P

The output of the Generate function o[i] = $Y^{t[i]}$ mod P

The following diagram shows the flow of the RNG.

$$\begin{tikzcd}
 & \mathbf{X^{t[i]} \bmod P} \arrow[dl] & & \\
\mathbf{s[i+1]/s[i]} \arrow[r] & \mathbf{t[i]} \arrow[u] \arrow[r] & \mathbf{Y^{t[i]} \bmod P} \arrow[r] & \mathbf{o[i]}
\end{tikzcd}$$

We will call this RNG the Dual DL RNG (for Dual Discrete Log Random Number Generator) For the following questions feel free to use Sage's built in modular exponentiation functionality, the example function for modular exponentiation, or any functions from previous problems.

a. Implement a Sage function that takes primes P, Q and integers X, Y of multiplicative order Q mod P and generates a random initial internal state s (an integer reduced mod Q). Have this function return a list with entries $[P,Q,X,Y,s]$.

b. Implement a Sage function that takes as a parameter a five element list corresponding to the internal state initialized by the function you wrote for part (a). This function should generate a single block of output, and update the list parameter's last element to correspond to the next RNG state.

c. Suppose that we have $P = 15116301544809716639$, $Q = 7558150772404858319$, $X = 10655637283854386401$, $Y = 5886823825742381258$, and furthermore we know that $X^e \equiv Y$ mod P, where $e = 1534964830632783921$. Find the positive integer f such that $Y^f \equiv X$ mod P. [Hint: remember that $XQ \equiv 1$ mod P. Find the positive integer f such that $e \cdot f = 1 + k \cdot Q$.]

d. Now, using the values for P, Q, X, Y from part (c), write a Sage function that, given one output of the generate function (from part (b)) and gives the output from the next call to the generate function. Use the functions you wrote in part (a) and (b) to show that your function works. [Hint: What happens if you exponentiate the output of the generate function by the value you found in part (c)?]

e. Write a version of the function that you wrote in part (b) that takes only P, Q, and X. Have it generate the value Y in a manner such that you know the positive integer f such $Y^f \equiv X \bmod P$. Your function should return a tuple (rngstate, f) where rngstate, is a valid rng state like the function from part (b) returns.

f. Generalize your attack function from part (d) to work given a block of output, with the Y and f values you generated in part (e).

g. How would you modify this RNG to overcome this problem?

8.7 The example version of the Chinese Remainder Theorem has several inefficiencies. Observe that in the Chinese Remainder Theorem the first step is to initialize the M array, where the value of M[i] is the product of all the moduli except moduli[i]. This is performed at the beginning of every function call, which is somewhat inefficient, because it could just be done once, for a single set of moduli. Furthermore, the output of this function is larger than it needs to be, indeed, it need be no larger than the product of all the moduli. In this question, do not merely call built in Sage functions.

a. Write a function to pre-compute the M array, it should also compute the product of all the moduli.

b. Write a version of the CRT function that takes the precomputed M array and a list of residues. Make sure that the output of this function is no larger than it needs to be.

8.8 The purpose of this question is to become more familiar with the Chinese Remainder Theorem functionality in Sage. Use Sage to compute the following questions about the CRT.

a. Find a number that reduces to 3 and 6 modulo 10 and 17, respectively

b. Find a number that reduces to 17, 89, 77, 65, and 100 modulo 23, 199, 503, 647, and 593, respectively

c. Find a number that reduces to 98189, 78089, and 13418 mod 519787, 722299 and 166169, respectively.

d. Compute the CRT basis of the moduli 100, 501, 999.

e. Find three numbers that reduce mod the moduli 49, 99, 1003, and 33191 to

i) 1,2,3,4

ii) 2,3,5,7

iii) 101, 99, 102, 98

f. Use Sage to compute an integer that is relatively prime to 1 through 5 modulo the first 5 primes, respectively.

8.9 The purpose of this question is to become more familiar with the Sage functionality for modular exponentiation. Use Sage to compute:

a. 123^456 mod 789

b. 100^797 mod 797

c. 15^30 mod 1000

d. 111^222 mod 987654321

e. 1217^2833 mod 3836311

f. Compute N, a product of two primes, both greater than 1,000,000 and then compute

8.10 The purpose of this function is to show how to use the Euler totient functionality built into Sage. Using the built-in functionality in Sage, compute the

Euler totient function on the following inputs:

a. 781
b. 10245
c. 110
d. Find an exponent x and one or two integers such that raising to the x power mod 547689 results in 1. Find at least one integer such that modular exponentiation with x and this modulus does not result in 1.
e. Find an exponent x and one or two integers such that raising to the x power mod 999999 results in 1. Find at least one integer such that modular exponentiation with x and this modulus does not result in 1.

C.9 CHAPTER 9: PUBLIC-KEY CRYPTOGRAPHY AND RSA

9.1 Use Sage to answer the following questions. Show all your Sage input/output:

a. Suppose your RSA public key factors as $p = 6569$ and $q = 8089$, and the public exponent e is 11. Suppose you were sent the Ciphertext 28901722. Perform the RSA Decryption and recover the plaintext.
b. Suppose that you want to encrypt the number 449 and send it to someone with public key $N = 37617577$, and $e = 529$
c. Suppose that you forgot your public exponent, but you know that the prime factors of your key's modulus are 1723 and 5381 and your private exponent is 223. Find the public exponent.
d. Use Sage to generate an RSA public/private key pair and perform an encryption and decryption.

9.2 Use Sage to solve the following problems: In part (a)-(c) determine if the following signatures are good or bad:

a. $N = 13962799$ and $e = 3$ value to sign $= 821$ and signature $= 8674413$
b. $N = 34300129$ and $e = 61$ value to sign $= 2478$ and signature $= 27535246$
c. $N = 5898461$ and $e = 23$ value to sign $= 419$ and signature $= 2607727$
d. Suppose that you have an RSA modulus with prime factors $p = 3181$ and $q = 2677$ and the public exponent is 163. Calculate the signature of 521 and then verify it.

9.3 The purpose of this question is to implement RSA encrypt and decrypt functions with Sage.

a. Implement an RSA key generation function.
b. Implement an RSA encrypt function.
c. Implement an RSA decrypt function.
d. Show that your functions work by simulating an RSA encrypt and decrypt with them.

9.4 The purpose of this question is to implement Sage functions for creating and verifying RSA signatures. For these questions you may use any answers from previous questions.

a. Implement a Sage function that takes an integer and an RSA private key and produces an RSA signature of it.

b. Implement a Sage function that takes an RSA signature and a hash value and determines if the signature is valid.
c. Show your functions work by simulating a sign and verify. Show at least one sign and verify and also show an example that if the hash or signature are incorrect, your verify function correctly fails. (You may use the key generation function from an earlier problem.)

C.10 CHAPTER 10: OTHER PUBLIC-KEY CRYPTOSYSTEMS

10.1 For all of the following questions related to Diffie-Hellman show all of your Sage input and output.

a. Suppose that you are Bob and you have agreed on the domain parameters $p = 70849$ and $g = 2$. Further suppose that Alice has sent the value $X = 39674$. Compute a secret value y and compute Y, and the shared secret.
b. Suppose that Alice and Bob have agreed on the domain parameters $p = 6779$ and $g = 3$, further suppose that Alice chooses the secret value $x = 384$ and Bob chooses the secret value $y = 152$. Perform a simulated key exchange as in the example.
c. Find a prime q and a prime p such that $p = 2q + 1$, find an element in the finite field with p elements that has multiplicative order q. Perform a simulated DH Secret Exchange as in the examples.

10.2 a. Implement a Sage function that takes a bound and returns 4 elements: p, q, g, and F. Satisfying: p and q are prime, such that $p = 2*q + 1$, g is an integer with multiplicative order q in the finite field with p elements, F is a Sage field object with p elements.
b. Implement a Sage function that takes the output from your function in part (a) and returns the pair (X, x) where $X = g$^$x \bmod p$ and x is greater than 1 and less than q.
c. Implement a Sage function that takes a public value from the other party in the DH key exchange and the secret value and returns the shared secret.
d. Show an example key exchange with your functions from parts (a) − (c).

10.3 The purpose of this question is to use Sage to explore how solving the discrete logarithm can break DH. In Sage, if a is an element of a finite field, and g generates a, then if the order of the finite field is small enough ***a*.log(g)** will return the discrete log of g with respect to a. Use this functionality to solve the following problems.

a. Suppose $p = 499$, $g = 7$, and $X = 297$. Find x such that $X = g$^x.
b. Suppose $p = 863$, $g = 5$, $X = 543$, and $Y = 239$. Find x and y such that $X = g$^x and $Y = g$^y.
c. Suppose $p = 7589$, $g = 2$, $X = 6075$ and $Y = 1318$. Find the shared secret value.

10.4 Recall the Dual DL PRNG (Problem 8.6). There is an actual crypto algorithm, called the Dual EC DL PRNG, where instead of an element in a multiplicative group mod a prime and exponentiation, we consider a point on an elliptic curve over a prime order finite field and scalar multiplication (see NIST SP-800-90, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.*) We need to define some auxiliary functions:

- $x(P)$: maps the x-coordinate of an elliptic curve point, P, to the integer the smallest positive integer that maps to x mod P.
- $\mathrm{LSB}_m(a)$: returns the least significant m bits of integer a.

And we also denote the following values:

- p: a prime, with n bits.
- E: an elliptic curve over a finite field with p elements, given by equation $y^2 = x^3 + ax + b$.
- P: a point on E, with prime order q (for maximum security q should be roughly the same size as p.)
- Q: a point in the cyclic subgroup of E generated by P.

At the beginning of iteration i we have internal state s[i], and we define the following values:

1. t[i] = s[i]
2. s[i + 1] = x(t[i] • P)
3. r[i] = x(t[i] • Q)
4. o[i] = LSB_{n-8}(r[i])

   Here o[i] is the output of the *i*th iteration block, and s[i + 1]

The following diagram shows the flow for generating one block of output with this Crypto Algorithm.

$$\begin{tikzcd}
 & \mathbf{x(t[i] \cdot P)} \arrow[dl] & & & & & \\
\mathbf{s[i+1]/s[i]} \arrow[rr] & & \mathbf{t[i]} \arrow[ul] \arrow[r] & \mathbf{x(t[i] \cdot Q)} \arrow[r] & \mathbf{r[i]} \arrow[r] & \mathrm{LSB}_{\mathbf{n-8}}(\mathbf{r[i]}) \arrow[r] & \mathbf{o[i]}
\end{tikzcd}$$

The following problems outline a similar problem with this algorithm as the one described in Problem 8.6.

a. Implement a Sage function to generate a single output block from this algorithm (Your function should take an internal state represented as a list with the following elements [E,P,Q,si], where E is a Sage Elliptic Curve object, P is a point on E, with prime order q, and Q is a point on E, generated by Q.

b. Write a Sage function that takes an output of this PRNG (i.e., the x coordinate of a point with the top 8 bits truncated off) and returns the possible values for R = t[i] • Q that could have generated that output [Hint: try the is_x_coordinate function on Elliptic Curve objects.]

c. Suppose you have E defined by y^2 = x^3 + 2x + 4, P = (42,980956284 88211854), Q = (6396452788131036613,9671497098832291002), and you know that the P has order q = 1227273995918533091 and also Q = 99689 • P. Write a Sage function that takes an output from one iteration of this function and returns a list of the possible next internal states.

- **d.** Suppose you know that o[i] = 58246156843038996, and o[i + 1] = 64511473570997445, use the fact that you have two subsequent outputs to determine the possible internal states that could have generated these two outputs.

10.5 For all of the following questions show your Sage input/output.

- **a.** Compute the order of the curve defined by $y^2 = x^3 + 7{*}x + 25$ over the finite field with 47 elements.
- **b.** On the curve defined by $y^2 + x{*}y = x^3 + x$ over $GF(2^8)$ compute the inverse of the point (1,1).
- **c.** On the curve defined by $y^2 + y = x^3 + x^2 + x + 1$ over the finite field with 701 elements, find a generator and show its order.
- **d.** On the curve defined by $y^2 = x^3 + 4187{*}x + 3814$ over finite field of size 6421 compute the sum of the points (3711,373) and (4376,2463).
- **e.** On the elliptic curve defined by $y^2 = x^3 + 3361{*}x + 6370$ over finite field of size 8461 compute 1001 times the point (1735, 3464).
- **f.** On the elliptic curve defined by $y^2 = x^3 + 1800{*}x + 1357$ over finite field of size 8191, let P1 = (1794, 1318) and P2 = (3514, 409), compute the sum of 13 times P1 plus 28 times P2.

10.6 In this problem, use the domain parameters. E is the elliptic curve defined by $y^2 = x^3 + 8871{*}x + 7063$ over the finite field with order 70177. The generator point G = (49359,30149) has order 70393. Show your Sage input/output.

- **a.** Suppose you are Bob and Alice has sent the point (10117, 64081) compute an integer y the point Y and the shared secret.
- **b.** Suppose that Alice chooses the secret value x = 2532 and Bob chooses the secret value y = 15276.
- **c.** Perform a full simulated secret agreement between Alice and Bob.

10.7 The purpose of this question is to implement Sage functions to perform ECDH.

- **a.** Write a function that takes a curve, and a base point on the curve and generates the secret value x and the public value X as per ECDH.
- **b.** Write a function that takes a public value and a secret value and computes the shared secret.
- **c.** Assume that your domain parameters are:
  Elliptic Curve defined by $y^2 = x^3 + 26484{*}x + 15456$ over Finite Field of size 63709
  q = 63839
  G = (53819,6786)
  Show your functions work by simulating an ECDH key exchange.

10.8 Recall that for cryptographic purposes, we use curves with prime order. The purpose of this question is to show why. Let E be the elliptic curve defined by $y^2 = x^3 + 7489{*}x + 12591$ over Finite Field of size 23431. This curve has order 23304. Let the base point be (20699, 19493).

- **a.** Compute 10 random multiples of this base point. What do you notice?
- **b.** Why is this bad? (*Hint:* What would happen if this was Alice or Bob's public point?)

C.11 CHAPTER 11: CRYPTOGRAPHIC HASH FUNCTIONS

11.1 The following describes a simple hash function: Choose p, q primes and compute $N = pq$. Choose g relatively prime to N and less than N. Then a number n is hashed as follows:

$$H = g^n \bmod N$$

If there is an m that hashes to the same value as n, then

$$g^m \equiv g^n \bmod N$$

so

$$g^{m-n} \equiv 1 \bmod N$$

which implies that

$$m - n \equiv 0 \bmod \phi(N)$$

So breaking this amounts to finding a multiple of $\phi(N)$, which is the hard problem in RSA.

a. Write a function that takes a bitlength n and generates a modulus N of bitlength n and g less than N and relatively prime to it.

b. Show the output of your function from part (a) for a few outputs.
Using N, g, n as arguments write a function to perform the hashing.

For parts (c) − (e) compute the simple hash:

c. $N = 600107, g = 154835, n = 239715$

d. $N = 548155966307, g = 189830397891, n = 44344313866$

e. $N = 604766153, g = 12075635, n = 443096843$

f. Write a function that creates a collision given p and q. Show that your function works for a couple of examples.

C.12 CHAPTER 13: DIGITAL SIGNATURES

13.1 Use Sage to solve the following problems. For these questions assume that we are using DSA with domain parameters:

$p = 7{,}877{,}914{,}592{,}603{,}328{,}881$

$q = 44449$

$g = 2{,}860{,}021{,}798{,}868{,}462{,}661$

Use these domain parameters to determine if the signatures are valid in parts (a) − (c).

a. public key $y = 3798043471854149631$, hash value $H = 59367$, and signature $(r,s) = (31019,4047)$

b. public key $y = 1829820126190370021$, hash value $H = 77241$, and signature $(r,s) = (24646,43556)$

c. public key $y = 4519088706115097514$, hash value $H = 48302$, and signature $(r,s) = (36283,32514)$

Perform a signing operation in parts (d)-(e).

d. private key $x = 8146$, hash value $H = 22655$
e. private key $x = 1548$, hash value $H = 32782$

13.2 The purpose of this question is to implement a DSA signature verification function.

a. Implement a function that takes domain parameters p, q, and g. Also, a Hash value H (in $\{1, 2, \ldots, p - 1\}$), a public key y, and a signature (r,s).
b. Use the function you wrote in part (a) as well as the functions from the DSA examples to simulate a DSA signature and verify as in the examples.