UNIVERSITY OF BRISTOL
DEPARTMENT OF COMPUTER SCIENCE
http://www.cs.bris.ac.uk

Assessed coursework
Applied Security (COMS30901)

Mod
(Assignment description)

Note that:

1. **The deadline for this assignment is** 21/02/14**.**
2. **This assignment represents** 25 **percent of the total marks available for COMS30901. It has been deemed an individual assignment: make sure you are aware of and adhere to the University and Faculty regulations which govern this type of work.**
3. **There are numerous support resources available, for example:**
   - **the online unit forum, which is hosted at**

     https://www.cs.bris.ac.uk/forum/index.jsp?title=COMS30901

     and on which lecturers, lab. demonstrators and students post questions and answers,
   - **the responsible lecturer, namely Dan Page, who is available in designated office hours, by appointment, or via email at**

     page@cs.bris.ac.uk

1 Introduction

Efficient implementation of modular multi-precision integer arithmetic (i.e., arithmetic in $\mathbb{Z}_N$ for some N) is a fundamental requirement for numerous *real* asymmetric cryptosystems; examples include RSA where $N = p \cdot q$ is a product of two primes, and ElGamal and ECC where N is *itself* a prime (so formally we use $\mathbb{F}_p$, where $p = N$). To quote an excellent essay on skills in CS,

> *RSA is easy enough to implement that everyone should do it.*
>
> – M. Might, http://matt.might.net/articles/what-cs-majors-should-know/

While the quote is a little tongue-in-cheek, the goal of this assignment is essentially as hinted. Specifically, your task is to develop an optimised implementation of selected public-key encryption and decryption operations. The practical nature of this assignment is crucial: it *should* help you more fully understand various topics covered only in theory elsewhere.

You should implement your solution in C, using the default version of GCC available to compile and thoroughly test it. Although there may be multiple valid approaches at each stage, your solution should aim to be as efficient as possible wrt. execution time: within reason trade-offs favouring time over space should be made, and issues of physical security (e.g., countermeasures against timing attacks) are secondary here (despite being important more generally). You are encouraged to make use of the GNU Multi-Precision (GMP) library[1], but beyond this third-party libraries (bar the standard C libraries) are outlawed.

2 Submission and marking

- This assignment is intended to help you learn something; where there is some debate about the right approach, the assignment demands *you* make an informed decision *yourself* and back it up with a reasonable argument based on *your own* background research.
- The assignment description refers to `marksheet.txt`. Download this ASCII text file from

  www.cs.bris.ac.uk/Teaching/Resources/COMS30901/cw/Mod/question/marksheet.txt

  then complete and include it in your submission; this is not optional, and failure to do so may result in a loss of marks.
- All implementations will be marked on a platform equivalent to those available in the CS lab. (MVB-2.11): take care to check your solution compiles, executes and has been tested using the default operating system and development tool-chain versions available.
- Where appropriate, include instructions that carefully describe how to compile and execute your submission; the ideal solution would include a `Makefile` (or equivalent) where not already provided.
- To make the marking process easier, your solution should only write error messages to `stderr` (or equivalent). The only input read from `stdin` (resp. output written to `stdout`, or equivalents) should be that specified by the assignment description.
- You should submit your work via the SAFE submission system at

  wwwa.fen.bris.ac.uk/COMS30901/

  including all source code, written solutions and any auxiliary files you think are important (e.g., example inputs and outputs). In the unlikely event that the submission system fails, email your work to the lecturer responsible for the assignment: the same rules wrt. deadlines and late work will still apply.
- All input and output values are represented as hexadecimal integer strings. Appendix A includes a detailed discussion of this issue, and examples of how to convert between representations.

[1] If you use the CS lab, or servers such as `snowy`, GMP version 5.0.5 is accessible within `/usr/local/`.

- Take note of the strict requirement wrt. functional correctness: the marks outlined in `marksheet.txt` demand computation of the correct result before the issue of efficiency is even considered. As such, developing a robust testing strategy for your solution is vital; this is discussed in some detail by Appendix B.

3 Material

If you download and unarchive the file

`www.cs.bris.ac.uk/Teaching/Resources/COMS30901/cw/Mod/question/question.tar.gz`

somewhere secure within your file system, you should find the following:

- `Makefile`, a GNU `make` based build system for the source code,
- A small set of test vectors for each stage (those associated with stage 2 are named `stage2.input` and `stage2.output` for example), and
- `modmul.[ch]`†, a driver program with incomplete functions for each stage.

You *should* be able to complete the assignment by limiting alterations to the files marked with †; either way, only existing files you do alter (including `marksheet.txt`) and additional files you create need to be submitted.

4 Description

4.1 Stage 1

This stage requires you to implement textbook (i.e., no message formatting or padding) RSA encryption. The input should be read from `stdin`: each 3-line group represents a challenge (one field per-line), of the form

- N, a 1024-bit modulus,
- e, a public exponent, and
- m st. $0 \leq m < N$, a plaintext to be encrypted.

The output should be written to `stdout`: for each challenge produce a 1-line output representing c, a ciphertext which is the encryption of m under the associated public key material. For example, you should be able to execute your program from a BASH shell using a command of the form

```
bash$ ./modmul stage1 < ${INPUT} > ${OUTPUT}
```

where `${INPUT}` and `${OUTPUT}` name input and output files respectively.

4.2 Stage 2

This stage requires you implement textbook (i.e., no message formatting or padding) RSA decryption. The input should be read from `stdin`: each 9-line group represents a challenge (one field per-line), of the form

- N, a 1024-bit modulus,
- d, a private exponent,
- p and q, 512-bit prime numbers st. $N = p \cdot q$,
- $d_p = d \pmod{p-1}$,
- $d_q = d \pmod{q-1}$,
- $i_p = p^{-1} \pmod{q}$,

- $i_q = q^{-1} \pmod{p}$, and
- c st. $0 \leq c < N$, a ciphertext to be decrypted.

Note that p and q are balanced in the sense that since N has 16 base-2^{64} digits (i.e., is a 1024-bit integer), both p and q have 8 base-2^{64} digits (i.e., are 512-bit integers).

The output should be written to `stdout`: for each challenge produce a 1-line output representing m, a plaintext which is the decryption of c under the associated private key material. For example, you should be able to execute your program from a BASH shell using a command of the form

```
bash$ ./modmul stage2 < ${INPUT} > ${OUTPUT}
```

where `${INPUT}` and `${OUTPUT}` name input and output files respectively.

4.3 Stage 3

This stage requires you implement textbook (i.e., no message formatting or padding) ElGamal encryption. The input should be read from `stdin`: each 5-line group represents a challenge (one field per-line), of the form

- p, a 1024-bit "large" modulus,
- q, a 160-bit "small" modulus,
- g, a generator of $\mathbb{F}_p^*$ with order q,
- h, a public key, and
- m st. $0 \leq m < p$, a plaintext to be encrypted.

The output should be written to `stdout`: for each challenge produce a 2-line output representing (c_1, c_2), a ciphertext which is the encryption of m under the associated public key material. For example, you should be able to execute your program from a BASH shell using a command of the form

```
bash$ ./modmul stage3 < ${INPUT} > ${OUTPUT}
```

where `${INPUT}` and `${OUTPUT}` are names of input and output file respectively.

Remember that the ciphertext for a given m will not be unique: it depends on the ephemeral key selected. To make testing this stage easier each test vector provided uses of a *fixed* ephemeral key $k = 1$, but clearly your final solution should select a random k to correctly implement the scheme.

4.4 Stage 4

This stage requires you implement textbook (i.e., no message formatting or padding) ElGamal decryption. The input should be read from `stdin`; each 6-line group represents a challenge (one field per-line), of the form

- p, a 1024-bit "large" modulus,
- q, a 160-bit "small" modulus,
- g, a generator of $\mathbb{F}_p^*$ with order q,
- x, a private key, and
- c_1 and c_2 st. $0 \leq c_1, c_2 < p$, a ciphertext to be decrypted.

The output should be written to `stdout`: for each challenge produce a 1-line output representing m, a plaintext which is the decryption of (c_1, c_2) under the associated private key material. For example, you should be able to execute your program from a BASH shell using a command of the form

```
bash$ ./modmul stage4 < ${INPUT} > ${OUTPUT}
```

where `${INPUT}` and `${OUTPUT}` are names of input and output file respectively.

A Representation and conversion

Somewhat bizarrely, one of the most confusing aspects of the assignment is arguably understanding how to correctly provide input and interpret output. This is crucial because values which *look* similar as human-readable strings may be interpreted differently depending on their machine-readable representation. To demonstrate the conversion to and from the pertinent data types, we use examples written in Python. Although they can be translated to other languages, use of Python is motivated by two features:

1. programs are typically executed interactively via an interpreter, allowing quick (re)evaluation and experimentation, and
2. such interpreters are widely available.

A.1 Decimal and hexadecimal integer strings

A.1.1 Representation

An integer string (or literal) is written as a string of characters, each of which represents a digit; the set of possible digits, and the value being represented, depends on a base b. We read digits from right-to-left: the least-significant (resp. most-significant) digit is the right-most (resp. left-most) character within the string. As such, the 20-character string

$$\hat{x} = \texttt{09080706050403020100}$$

represents the integer value

$$\begin{aligned}\hat{x} \mapsto\ & \hat{x}_{19} \cdot b^{19} + \hat{x}_{18} \cdot b^{18} + \hat{x}_{17} \cdot b^{17} + \hat{x}_{16} \cdot b^{16} + \hat{x}_{15} \cdot b^{15} + \\ & \hat{x}_{14} \cdot b^{14} + \hat{x}_{13} \cdot b^{13} + \hat{x}_{12} \cdot b^{12} + \hat{x}_{11} \cdot b^{11} + \hat{x}_{10} \cdot b^{10} + \\ & \hat{x}_{9} \cdot b^{9} + \hat{x}_{8} \cdot b^{8} + \hat{x}_{7} \cdot b^{7} + \hat{x}_{6} \cdot b^{6} + \hat{x}_{5} \cdot b^{5} + \\ & \hat{x}_{4} \cdot b^{4} + \hat{x}_{3} \cdot b^{3} + \hat{x}_{2} \cdot b^{2} + \hat{x}_{1} \cdot b^{1} + \hat{x}_{0} \cdot b^{0}\end{aligned}$$

Of course the actual value depends on the base b in which we interpret the representation $\hat{x}$:

- for a decimal integer string $b = 10$, meaning

$$\begin{aligned}\hat{x} \mapsto\ & \hat{x}_{19} \cdot 10^{19} + \hat{x}_{18} \cdot 10^{18} + \hat{x}_{17} \cdot 10^{17} + \hat{x}_{16} \cdot 10^{16} + \hat{x}_{15} \cdot 10^{15} + \\ & \hat{x}_{14} \cdot 10^{14} + \hat{x}_{13} \cdot 10^{13} + \hat{x}_{12} \cdot 10^{12} + \hat{x}_{11} \cdot 10^{11} + \hat{x}_{10} \cdot 10^{10} + \\ & \hat{x}_{9} \cdot 10^{9} + \hat{x}_{8} \cdot 10^{8} + \hat{x}_{7} \cdot 10^{7} + \hat{x}_{6} \cdot 10^{6} + \hat{x}_{5} \cdot 10^{5} + \\ & \hat{x}_{4} \cdot 10^{4} + \hat{x}_{3} \cdot 10^{3} + \hat{x}_{2} \cdot 10^{2} + \hat{x}_{1} \cdot 10^{1} + \hat{x}_{0} \cdot 10^{0} \\ \mapsto\ & 0_{(10)} \cdot 10^{19} + 9_{(10)} \cdot 10^{18} + 0_{(10)} \cdot 10^{17} + 8_{(10)} \cdot 10^{16} + 0_{(10)} \cdot 10^{15} + \\ & 7_{(10)} \cdot 10^{14} + 0_{(10)} \cdot 10^{13} + 6_{(10)} \cdot 10^{12} + 0_{(10)} \cdot 10^{11} + 5_{(10)} \cdot 10^{10} + \\ & 0_{(10)} \cdot 10^{9} + 4_{(10)} \cdot 10^{8} + 0_{(10)} \cdot 10^{7} + 3_{(10)} \cdot 10^{6} + 0_{(10)} \cdot 10^{5} + \\ & 2_{(10)} \cdot 10^{4} + 0_{(10)} \cdot 10^{3} + 1_{(10)} \cdot 10^{2} + 0_{(10)} \cdot 10^{1} + 0_{(10)} \cdot 10^{0} \\ \mapsto\ & 9080706050403020100_{(10)}\end{aligned}$$

  whereas

- for a hexadecimal integer string $b = 16$, meaning

$$\begin{aligned}\hat{x} \mapsto\ & \hat{x}_{19} \cdot 16^{19} + \hat{x}_{18} \cdot 16^{18} + \hat{x}_{17} \cdot 16^{17} + \hat{x}_{16} \cdot 16^{16} + \hat{x}_{15} \cdot 16^{15} + \\ & \hat{x}_{14} \cdot 16^{14} + \hat{x}_{13} \cdot 16^{13} + \hat{x}_{12} \cdot 16^{12} + \hat{x}_{11} \cdot 16^{11} + \hat{x}_{10} \cdot 16^{10} + \\ & \hat{x}_{9} \cdot 16^{9} + \hat{x}_{8} \cdot 16^{8} + \hat{x}_{7} \cdot 16^{7} + \hat{x}_{6} \cdot 16^{6} + \hat{x}_{5} \cdot 16^{5} + \\ & \hat{x}_{4} \cdot 16^{4} + \hat{x}_{3} \cdot 16^{3} + \hat{x}_{2} \cdot 16^{2} + \hat{x}_{1} \cdot 16^{1} + \hat{x}_{0} \cdot 16^{0} \\ \mapsto\ & 0_{(16)} \cdot 16^{19} + 9_{(16)} \cdot 16^{18} + 0_{(16)} \cdot 16^{17} + 8_{(16)} \cdot 16^{16} + 0_{(16)} \cdot 16^{15} + \\ & 7_{(16)} \cdot 16^{14} + 0_{(16)} \cdot 16^{13} + 6_{(16)} \cdot 16^{12} + 0_{(16)} \cdot 16^{11} + 5_{(16)} \cdot 16^{10} + \\ & 0_{(16)} \cdot 16^{9} + 4_{(16)} \cdot 16^{8} + 0_{(16)} \cdot 16^{7} + 3_{(16)} \cdot 16^{6} + 0_{(16)} \cdot 16^{5} + \\ & 2_{(16)} \cdot 16^{4} + 0_{(16)} \cdot 16^{3} + 1_{(16)} \cdot 16^{2} + 0_{(16)} \cdot 16^{1} + 0_{(16)} \cdot 16^{0} \\ \mapsto\ & 42649378395939397566720_{(10)}\end{aligned}$$

A.1.2 Example

Consider the following Python program

```
a = "09080706050403020100"

b = long( a, 10 )
c = long( a, 16 )

d = ( "%d" % ( c ) )
e = ( "%X" % ( c ) )

f = ( "%X" % ( c ) ).zfill( 20 )

print "type( a ) = %-13s a = %s" % ( type( a ), str( a ) )
print "type( b ) = %-13s b = %s" % ( type( b ), str( b ) )
print "type( c ) = %-13s c = %s" % ( type( c ), str( c ) )
print "type( d ) = %-13s d = %s" % ( type( d ), str( d ) )
print "type( e ) = %-13s e = %s" % ( type( e ), str( e ) )
print "type( f ) = %-13s f = %s" % ( type( f ), str( f ) )
```

which, when executed, produces

```
bash$ python integer.py
type( a ) = <type 'str'>  a = 09080706050403020100
type( b ) = <type 'long'> b = 90807060504030201 00
type( c ) = <type 'long'> c = 42649378395939397566720
type( d ) = <type 'str'>  d = 42649378395939397566720
type( e ) = <type 'str'>  e = 9080706050403020100
type( f ) = <type 'str'>  f = 09080706050403020100
```

The idea is that

- a is an integer string (i.e., a sequence of characters),
- b and c are conversions of a into integers (actually a Python `long`, which is the multi-precision integer type used), using decimal and hexadecimal respectively, and
- d and e are conversions of c into strings (i.e., a sequence of characters), using decimal and hexadecimal respectively.

Note that a and e do not match: the conversion has left out the left-most zero character, since this is not significant wrt. the integer value. To resolve this issue where it is problematic, the `zfill` function can be used to left-fill the string with zero characters until it is of the required length (here 20 characters in total, forming f which does then match).

B Strategies for effective development and debugging

B.1 Development

The overarching goal of this assignment is that you understand as much of the studied arithmetic stack as possible, by implementing as much of it as possible. However, it is designed to be modular in the sense that you can pick up partial marks for targeting effort on parts of the stack one step at a time.

Rather than try to implement everything at once therefore, a more sensible development approach would be as follows:

1. first implement the basic schemes, and optimise RSA decryption using the CRT, then
2. replace all GMP-based modular exponentiations with your own implementation of a non-binary algorithm (e.g., based on one of several possible "windowed" options), then
3. replace all GMP-based modular multiplications with your own implementation of Montgomery multiplication, thereby turning your exponentiation implementation into an Montgomery-based alternative.

Taking this approach a) focuses your effort on the sub-tasks that are of most value in terms of marks, while b) ensuring you can make step-by-step progress, initially depending heavily then later more lightly on GMP.

B.2 Debugging

Use of test vectors (i.e., sets of input and corresponding known-correct output), such as those provided, only represents one strategy among many for testing your implementation. As such, it is worth highlighting some other examples:

Comparison with an oracle The problem with test vectors is that they (typically) only capture input and output, treating the associated operation as a black-box: for a given input x, if your implementation produces an $r \neq f(x)$ for some operation it can be hard to tell why this is. One strategy to cope with this problem is to generate intermediate results using a reference implementation of the operation, rather than rely on the reference input and output for it. Starting with the known-correct input (resp. output), the idea is to move forward (resp. backward) through the computation step-by-step until it is clear where exactly there is a difference: this allows you to identify and hence resolve the cause.

Appendix A highlights the fact that Python (among other examples) has native support for multi-precision integers, meaning it is well suited to act as or supporting a reference implementation. In some cases, an obvious approach is to use similar functionality in GMP as such a reference.

Use of randomisation and Built-In Self-Test (BIST) Within this context, a BIST can be harnessed by using various natural arithmetic identities: instead of requiring use of test vectors, the idea is to generate random inputs and check whether after application of an operation (or series thereof), such an identity holds. In short, each time one selects a random set of inputs and successfully applies such a check, one obtains some confidence about the implementations involved: since the process is easily to automate, this confidence can more easily be achieved.

For example, for a random x

$$1 \equiv x/x \equiv x \cdot \frac{1}{x} \equiv x \cdot x^{-1} \pmod{N}$$

so one can check whether implementations of modular inversion and multiplication are correct. Likewise, for a random x

$$x \equiv \mathbb{Z}_N\text{-MontMul}(\mathbb{Z}_N\text{-MontMul}(x, \rho^2 \pmod{N}), 1)$$

i.e., converting x into Montgomery representation and then back again should give the same x, so one can check whether an implementation of Montgomery multiplication is correct.

B.3 Miscellaneous

- Understanding and keeping in mind the required formal type (in the sense of the C type system) of data is important since this will allow you to avoid problems (e.g., implicit type coercion).

  For example, in the CS lab you should find

  ```
  sizeof( mp_limb_t )           =  8
  sizeof( unsigned int )        =  4
  sizeof( unsigned long )       =  8
  sizeof( unsigned long int )   =  8
  sizeof( unsigned long long )  =  8
  ```

  implying that $w = 8 \cdot 8 = 64$: you are using a 64-bit processor so therefore $b = 2^w = 2^{64}$. Also note that it *should* be possible to use a limb as the second operand in functions such as `mpz_mul_ui` (since GMP requires an operand of type `unsigned long int` but this matches `mp_limb_t`).

- Debugging an implementation with inputs that change per-execution (or which is non-deterministic through use of any auxiliary inputs, such as randomness) will produce the proverbial hard-to-hit moving target. Your first step should therefore be to

  - select a set of inputs that trigger the error reliably, and
  - make sure that each successive execution uses these fixed inputs (including any auxiliary inputs).

  The inputs you select will ideally be minimal in the sense they make the debugging task as easy as possible. For example, on a 64-bit processor one might reason that

  - using smaller 256-bit inputs will typically be easier than larger 1024-bit inputs, but
  - using overly small 64-bit inputs does not make sense, since they are less likely to provide good coverage (e.g., they only ever require one limb).

- Make sure any parameters or pre-computation you rely on are valid and correctly computed before focusing on the operation itself. Various cases should be clear within the example of Montgomery multiplication, e.g.,

  - check the fixed parameters satisfy various requirements, such as $\gcd(N, b) = 1$, and
  - check the pre-computed Montgomery parameters are correct, for example using Python (with the PyCrypto library for support) per

  ```
  import Crypto.Util.number as number

  b = 10
  N = 667
  n_N = int( math.ceil( math.log( N, b ) ) )
  rho = b ** n_N
  rho_2 = pow( rho, 2, N )
  omega = ( -number.inverse( N, rho ) ) % b

  print b, N, n_N, rho, rho_2, omega
  ```

- Localising any error to as minimal a part of the implementation as possible makes any debugging task easier. You might hear the term minimal working (or non-working) example in this context: the idea is to aggressively trim an implementation, removing or fixing parts while the error still persists, to remove extraneous functionality and focus on the root cause.

  Effective unit testing can often assist this process. For example, one might rigorously test a stand-alone implementation of modular multiplication *before* using it as a component within modular exponentiation: eliminating errors in the former will mean you can more confidently rule it out as the cause of errors in the later.