Assignment 4
Public Key Cryptography

Prof. Darrell D. E. Long
CSE 13S – Fall 2021

Due: TBD at 11:59 pm

1 Introduction

> *Doo Jdxo lv glylghg lqwr wkuhh sduwv, rqh ri zklfk wkh Ehojdh lqkdelw, wkh Dtxlwdql dqrwkhu, wkrvh zkr lq wkhlu rzq odqjxdjh duh fdoohg Fhowv, lq rxu Jdxov, wkh wklug.*
>
> —Julius Cæsar

Cryptography, once restricted to government, spies, and the military is now pervasive in our lives. Most web sites that you visit are protected using SSL. Your SSH connections are protected in the same way.

How is this accomplished? Through a mixture of *public key* and *symmetric key* cryptography. The earliest known practical public-key cryptography algorithm is RSA, after its inventors Ronald Rivest, Adi Shamir, and Leonard Adleman (Figure 2), who published it in 1978. About five years earlier, on 20 November, 1973, Clifford Cocks (Figure 1), working for GCHQ (the British equivalent of the NSA), invented a very similar algorithm. His classified memorandum "A note on 'non-secret' encryption" was to remain secret for 24 years. In fact, when you read the Cocks memorandum, you will see that the *idea* of public key encryption was proposed by J. H. Ellis three years earlier in 1970. Unknown in the public literature, the idea was independently proposed by Ralph Merkle for public key distribution, which inspired asymmetric cryptography by Whitfield Diffie and Martin Hellman, and finally leading to RSA.

Public-key cryptography, or asymmetric cryptography, is a cryptographic system that uses pairs of keys: public keys (known to others) and private keys (known only by the owner). The generation of such key pairs depends on cryptographic algorithms that are based on mathematical objects called *one-way functions*. Security requires keeping the private key private; the public key can be distributed widely.

Any person can encrypt a message using the intended receiver's public key, but that encrypted message can only be decrypted with the receiver's private key. This allows a server to create a cryptographic key for suitable symmetric-key cryptography and then use a client's openly shared public key to encrypt the newly generated symmetric key. The server can then send this encrypted symmetric key over an insecure channel to the client; only the client can decrypt it using its private key. With the client and server both having the same symmetric key, they can safely use symmetric key encryption to communicate. This scheme has the advantage of not having to pre-share symmetric keys while gaining the higher speed of symmetric-key cryptography.

Symmetric-key algorithms use the same cryptographic keys for the encryption of plaintext and the decryption of ciphertext. The keys may be identical, or there may be a simple transformation between the two keys. The keys represent a shared secret between two or more parties. The requirement that both parties have access to the secret key is one of the main disadvantages of symmetric-key encryption compared to public-key encryption.

Let's briefly look at the Cocks algorithm before moving on to the more popular RSA algorithm. We have two principals: *Alice* (A), who is the receiver, and *Bonnie* (B), who is the sender.

(a) Alice:

i. Chooses two primes p and q such that $p \nmid (q-1)$ and $q \nmid (p-1)$. That is, p does not divide $q-1$ and q does not divide $p-1$.

ii. Transmits the computed product $n = pq$ to the sender, which we write as $A \xrightarrow{n} B$.

(b) Bonnie:

i. Has a message consisting of numbers $c_1, c_2, \ldots, c_r$ where $0 < c_i < n$.

ii. Sends these encoded as d_i where $d_i = c_i^n \pmod{n}$. When written as part of a protocol, $B \xrightarrow{c_1,\ldots,c_r} A$.

(c) Alice:

i. Computes using Euclid's Algorithm p' such that $p \times p' \equiv 1 \pmod{q-1}$, and q' such that $q \times q' \equiv 1 \pmod{p-1}$.

ii. Decodes $c_i = d_i^{p'} \pmod{q} = d_i^{q'} \pmod{p}$.

Figure 1: Clifford Cocks

As with the RSA algorithm, as you will see, the strength of the algorithm relies on the assumed difficulty of factoring large composite integers. We say *assumed difficulty* since there is, like $\mathscr{P} \stackrel{?}{=} \mathscr{NP}$, no proof of this widely held assumption. A proof that $\mathscr{P} \neq \mathscr{NP}$ would be welcome, but unsurprising, while a proof that $\mathscr{P} = \mathscr{NP}$ would probably have theoreticians jumping out of windows.

The paper published by Rivest, Shamir and Adleman in 1978,

> Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM* 21.2 (1978): 120–126.

is one of the most important papers ever published. It enabled the modern Internet and changed the world. You would do well to take the time to read it.

2 RSA Algorithm

The magic words are Squeamish Ossifrage.

—Ronald L. Rivest

The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, known as the *factoring problem*. Breaking RSA encryption is known as the *RSA problem*. Whether it is as difficult as the

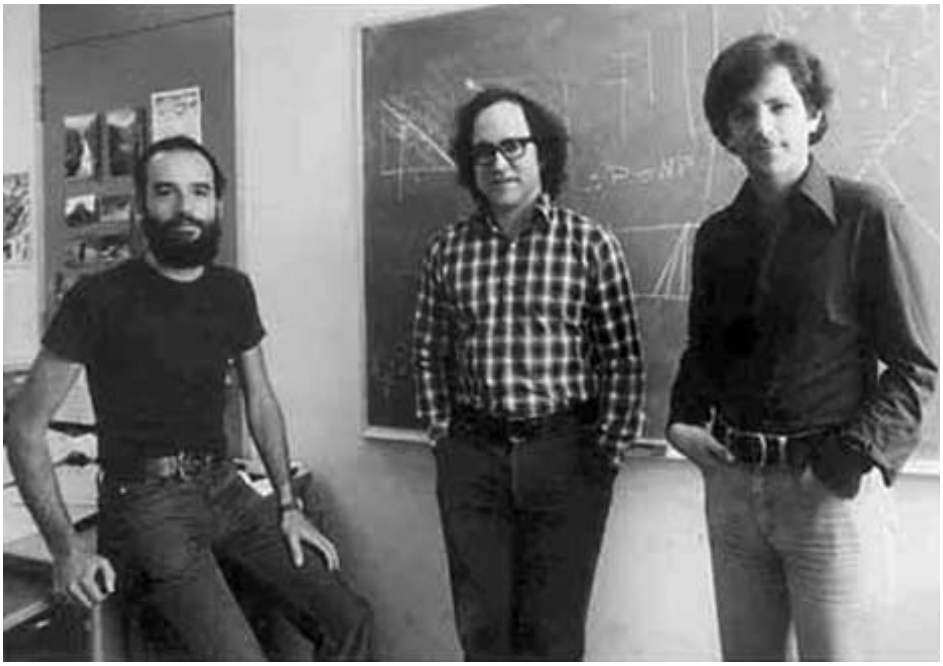

Figure 2: Adi Shamir, Ronald Rivest, and Leonard Adelman

factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used, but RSA would be vulnerable to attack by a *quantum computer*.

RSA involves a public key and a private key. Everyone can know the public key, and it is used for encrypting messages. The intention is that messages encrypted with the public key can only be decrypted by using the private key. The integers n and e represent the public key. The private key is represented by the integer d.

The public key consists of the modulus n and the public exponent e. The private key consists of the private exponent d, which must be kept secret; p, q, and $\varphi(n)$ must also be kept secret since they are used to calculate d. In fact, p and q ($\varphi(n)$ is calculated from them) can be discarded after d has been computed.

We proceed by choosing two large random primes p and q, these numbers must be kept secret. We then publish the number $n = pq$. You might wonder why we can do this, and the reason is that it is *believed* to be hard to factor large composite integers into their constituent primes. The *fundamental theorem of arithmetic* tells us that every integer has a *unique* prime factorization.

Choose a *random* integer $2 < e < n \ni \gcd(e, \varphi(n)) = 1$, where $\varphi(n) = (p-1)(q-1)$. $\gcd(a,b)$ indicates the greatest common divisor of a and b, which is commonly computed using Euclid's algorithm, which is defined recursively as:

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ \gcd(a, b-a) & \text{if } a < b \\ \gcd(a-b, b) & \text{if } a > b \end{cases}$$

though we can calculate it much more rapidly using division. A good choice for e is $2^{16} + 1 = 65537$. You will understand why—to some extent—after you finish §6.3. Here's a hint: How many 1 bits are in that number?

The φ function is called the *totient* of n, and that denotes the number of positive integers up to a given integer n that are relatively prime to n. Note that for any prime p, $\varphi(p) = p-1$, and so $\varphi(n) = \varphi(pq) = (p-1)(q-1)$. We can share e with impunity. We say that our *public key* is the pair $\langle e, n \rangle$.

We now calculate a unique secret integer $d \in \{0, \ldots, n-1\}$ such that $d \times e \equiv 1 \pmod{\varphi(n)}$. How do we find this d? It turns out that we have known how to do it for more than 2300 years—we use an algorithm attributed to *Euclid*. How is it that we can easily calculate d while our adversary cannot? We know a secret that he does not: we

know $\varphi(n)$ while he only knows n. We call d our *private key*.

We now define two functions: $D(m) = m^e \pmod n$ and $E(c) = c^d \pmod n$. We will show in §3 that $\forall m \in \{0,\ldots,n-1\}$ that $D(E(m)) = E(D(m)) = m$. Since D and E are *mutual inverses* and this will enable us to perform not only encryption but also digital signatures.

3 Mathematics of RSA

If $\mathscr{P} = \mathscr{NP}$, then all of modern cryptography collapses. On this happy thought...

Michael O. Rabin, November 1998

The mathematics of RSA are based on arithmetic in a group of integers modulo n, denoted $\mathbb{Z}/n$. This is the set $\{0,\ldots,n-1\}$ and all sets that are isomorphic to it. For example, $\{n,\ldots,2n-1\} \pmod n = \{0,\ldots,n-1\} \pmod n$, and there are an infinite number of such sets. Since they are *all the same* we will only concern ourselves with the one with the smallest numbers. What do we mean when we say that are the same? We mean that if $x \equiv k \pmod n$ then $an + x \equiv k \pmod n, \forall a \geq 0, a \in \mathbb{Z}$. In other words, additional integer products of n do not matter.

The Euler-Fermat theorem says that if $a \in \mathbb{N}$ and $n \in \mathbb{N}$ are *coprime*, that is $\gcd(a,n) = 1$, then

$$a^{\varphi(n)} \equiv 1 \pmod n.$$

This will allow us, for example, to take a message M and have $M^{\varphi(n)} \equiv 1 \pmod n$.

What is $\varphi(n)$? It is the Euler totient function, and gives the number of positive integers than that or equal to n that are relatively prime to n. For any prime number p,

$$\varphi(p) = p - 1.$$

For the RSA algorithm, we choose two large primes p and q, and we make $n = pq$. What does this mean for us with respect to $\varphi(n)$?

$$\begin{aligned}\varphi(n) &= \varphi(p) \times \varphi(q) \\ &= (p-1)(q-1) \\ &= n - (p+q) + 1.\end{aligned}$$

We choose an encryption key e such that is it relatively prime to $\varphi(n)$, that is, $\gcd(e, \varphi(n)) = 1$. We can then deduce our decryption key d such that $e \times d \equiv 1 \pmod{\varphi(n)}$. Our encryption algorithm is simply $E(M) = M^e \pmod n = C$, and our decryption algorithm is $D(C) = c^d \pmod n = M$.

We have the additional property of being mutual inverses: $E(D(x)) = D(E(x)) = x$, and that will give us the ability to provide *digital signatures*. Observe that,

$$\begin{aligned}D(E(M)) &\equiv E(M)^d \equiv M^{ed} \pmod n \\ E(D(M)) &\equiv D(M)^e \equiv M^{de} \pmod n.\end{aligned}$$

Continuing in this manner, observe that,

$$M^{ed} \equiv M^{k\varphi(n)+1} \pmod n$$

for some multiplier $k \geq 1$. This is true because, prior to the modular reduction, ed must be one greater than some multiple of $\varphi(n)$; applying the modulus is what makes $ed \equiv 1 \pmod{\varphi(n)}$. We can rewrite this as

$$M^{ed} \equiv (M^{\varphi(n)})^k \times M \pmod n.$$

Here we apply Euler's theorem, which states that if a and n are coprime integers, then $a^{\varphi(n)} \equiv 1 \pmod{n}$. So, assuming that M is coprime with n, we can simplify the above equation to

$$\begin{aligned} M^{ed} &\equiv (1)^k \times M && \pmod{n} \\ &\equiv (1) \times M && \pmod{n} \\ &\equiv M && \pmod{n} \end{aligned}$$

which shows that the encryption and decryption functions are mutual inverses.

4 Your Task

You will be creating three programs for this assignment:

1. A key generator: `keygen`
2. An encryptor: `encrypt`
3. A decryptor: `decrypt`

The `keygen` program will be in charge of key generation, producing RSA public and private key pairs. The `encrypt` program will encrypt files using a public key, and the `decrypt` program will decrypt the encrypted files using the corresponding private key.

You will need to implement two libraries and a random state module that will be used in each of your programs. One of the libraries will be hold functions relating to the mathematics behind RSA, and the other library itself will contain implementations of routines for RSA. You also need to learn to *use* a library: the GNU multiple precision arithmetic library.

5 GNU Multiple Precision Arithmetic

As you should know by now, **C**, unlike languages like **Python**, does not natively support arbitrary precision integers. The security of RSA, however, relies on large integers. So, we elect to use the GNU multiple precision arithmetic library, usually referred to as GMP. You can find the manual and documentation for the library here:

`https://gmplib.org/manual`.

Take some time to look through the manual, taking note of which functions may be useful.

You will need to install both `gmp` and `pkg-config`. The latter is a utility used to assist in finding and linking libraries, instead of having the program hard-code where to find specific headers and libraries during program compilation. To install these packages on Ubuntu 20.04, run the following:

```
$ sudo apt install pkg-config libgmp-dev
```

Get started on this *as soon as possible*. Make sure to attend section for assistance on using `pkg-config` in a `Makefile` to direct the compilation process for your programs.

You may notice that GMP already provides number theoretic functions, several of which *could* be used in RSA. You may not use any GMP-implemented number theoretic functions. You *must* implement those functions yourself.

The following two sections (§6 and §7) will present the functions that you have to implement, but they both will require the use of random, arbitrary-precision integers.

GMP requires us to explicitly initialize a random state variable and pass it to any of the random integer functions in GMP. Not only that, we also need to call a dedicated function to clean up any memory used by the initialized random state. To remedy this, you will implement a small random state module, which contains a single `extern` declaration to a global random state variable called `state`, and two functions: one to initialize the state, and one to clear it. The interface for the module will be given in `randstate.h` and the implementation must go in `randstate.c`.

`void randstate_init(uint64_t seed)`

Initializes the global random state named `state` with a Mersenne Twister algorithm, using `seed` as the random seed. This entails a call to `gmp_randinit_mt()` and a call to `gmp_randseed_ui()`.

`void randstate_clear(void)`

Clears and frees all memory used by the initialized global random state named `state`. This should just be a single call to `gmp_randclear()`.

6 Number Theoretic Functions

Number Theory is the branch of mathematics that studies the nature and properties of numbers. Though many have made important contributions to the field, including Gauß in his *Disquisitiones Arithmeticae* (which he completed when he was 21 years old), the most important for public-key cryptography are Fermat and Euler (Figure 3).

You will first need to implement the functions that drive the mathematics behind RSA before you can tackle your RSA library. The interface for these functions will be given in `numtheory.h` and should be defined in corresponding **C** file. Read each of the subsections carefully to understand, on some level, the theory behind each of the number theoretic functions. Pseudocode is provided to assist you.

Figure 3: Leonhard Euler (1707–1783) and Pierre de Fermat (1607–1665)

6.1 Modular Exponentiation

As shown in §2, we must compute a^n where both $a, n \in \mathbb{N}$ for RSA. We could simply multiply:

$$a^n = \overbrace{a \times a \times \cdots \times a \times a}^{n}.$$

The number of multiplications is $n-1$, which is $O(n)$. While correct, this approach is naïve and *extremely inefficient*. Since we are working with very large numbers in RSA, we must be able to compute modular exponentiation quickly. So the question is, can we do better? We can in fact do much better, computing a^n in $O(\log_2(n))$ steps.

Recall that we can write any integer as a polynomial

$$n = c_m 2^m + c_{m-1} 2^{m-1} + \cdots + c_1 2^1 + c_0 2^0 = \sum_{0 \le i \le m} c_i 2^i,$$

where $n \geq 2^m$ and $c_i \in \{0,1\}$. And so,

$$a^n = a^{c_m 2^m + c_{m-1} 2^{m-1} + \cdots + c_1 2^1 + c_0 2^0}.$$

Since $a^{b+c} = a^b \times a^c$, then we can rewrite the formula as

$$a^n = a^{c_m 2^m} \times a^{c_{m-1} 2^{m-1}} \times \cdots \times a^{c_1 2^1} \times a^{c_0 2^0} = \prod_{0 \leq i \leq m} a^{c_i 2^i}.$$

As an example, consider $a^{13} = a^{2^3+2^2+2^0} = a^{8+4+1} = a^8 \times a^4 \times a^1$. You will want to try a few more to get a feeling for it before you attempt to write code.

This leaves us with the problem of computing the a^{2^i} terms. We start with $a = a^1$ and if we square it then $(a^1)^2 = a^2$. Each time we square, $(a^2)^2 = a^4, (a^4)^2 = a^8, \ldots$ the exponents are a power of 2. We only have to square our previous result $\log_2 n$ times at most.

You will notice that the numbers get *very large, very fast.* Although we want enormous numbers for cryptography, we do not want numbers that would be impossible to even write down if we used every atom in the universe. Recall that 10^k is k digits long. That means that if $k = 10^{1000}$ then there are that many digits (there are approximately 10^{82} atoms in the observable universe). Consequently, we will usually do such computations (mod n) for some modulus n, meaning that all numbers are in $\{0, \ldots, n-1\}$.

To implement modular exponentiation, you should simply follow the steps to perform exponentiation by squaring as shown above and reduce your results modulo n after each operation that is likely to yield a large result (you do not need to do it, if for example, you just add a small constant). The following pseudocode shows the repeated squaring and modular reduction at each step.

```
POWER-MOD(a, d, n)
1  v ← 1
2  p ← a
3  while d > 0
4      if ODD(d)
5          v ← (v × p) mod n
6      p ← (p × p) mod n
7      d ← ⌊d/2⌋
8  return v
```

The function that you are expected to implement to perform modular exponentiation should be declared as follows:

```
void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)
```

Performs fast modular exponentiation, computing `base` raised to the `exponent` power modulo `modulus`, and storing the computed result in `out`.

6.2 Primality Testing

> *There are many methods—none of them as good as the randomized primality test.*
>
> —Michael O. Rabin, October 1997

The simplest primality test is trial division: given an input number, n, check whether it is evenly divisible by any prime number between 2 and $\sqrt{n}$. Thus, this simple algorithm[1] is $O(\sqrt{n})$, but can we do better? The answer is

[1] How big is $\sqrt{n}$? A typical encryption key has more than 600 decimal digits. Thus, $\sqrt{10^{600}} = 10^{300}$. Suppose we can do one trial division every *nanosecond*, then that's $10^{300-9} = 10^{291}$ seconds. There are 22,896,000 or about 10^7 seconds per year, so it will take about 10^{284} years (the Big Bang was about 13.7×10^9 years ago).

subtle. To be certain, we must try all of the primes from up to $\sqrt{n}$; there is no way to escape it. But we can do much better if we are willing to accept an answer of *probably*.

Since it is infeasible to use a *deterministic* algorithm, we can solve many problems with *high probability* by using a *randomized algorithm*. Such algorithms explore random parts of the problem space so that we have high (but not perfect) confidence that they have solved the problem.

Probabilistic tests are more rigorous than heuristic tests in that they provide provable bounds on the probability of being fooled by a composite number. All practical primality tests are probabilistic tests. These tests use, apart from the tested number n, some other number a (called a *witness*) that is chosen at random from some sample space. The usual randomized primality tests never report a prime number as composite, but a composite number may be reported as prime.

The simplest probabilistic primality test is the Fermat primality test. It works as follows: Given an integer n, choose some integer a coprime to n and calculate $a^{n-1} \pmod n$. If the result is different from 1, then n is composite. If it is 1, then n may be prime. If $a^{n-1} \equiv 1 \pmod n$ n is not prime, then n is called a pseudoprime to base a. In practice, we observe that, if $a^{n-1} \equiv 1 \pmod n$, then n is usually prime.

Better yet is the Solovay-Strassen probabilistic primality test, developed by Robert M. Solovay and Volker Strassen in 1977. It is of particular importance since it made practical public-key algorithms such as RSA.


Figure 4: Gary L. Miller and Michael O. Rabin

The Miller–Rabin primality test, invented by Gary Miller and Michael O. Rabin (Figure 4), is an even more sophisticated probabilistic test, which detect all composites (once again, this means: for every composite number n, at least $\frac{3}{4}$ of numbers a are witnesses of compositeness of n). The accuracy of these tests is compared in Figure 5.

The Miller–Rabin primality test works as follows: Given an integer n, choose some positive integer $a < n$. Let $2^s d = n-1$, where d is odd. If $a^d \not\equiv \pm 1 \pmod n$ and $a^{2^r d} \not\equiv \pm 1 \pmod n$ for all $0 \le r \le s-1$, then n is composite and a is a witness for the compositeness. Otherwise, n *might* be prime. That is, we might be wrong $\frac{1}{4}$ of the time. If we repeat the test 100 times then our chance of being wrong is $(\frac{1}{4})^{100} = 2^{-200}$ and that is usually more than good enough. The Miller–Rabin test is considered a strong pseudoprime test, where a pseudoprime is a number that is determined to be *probably prime* by a probabilistic test, but not actually prime. Deterministic primality tests, such as the AKS primality test, do not give false positives.

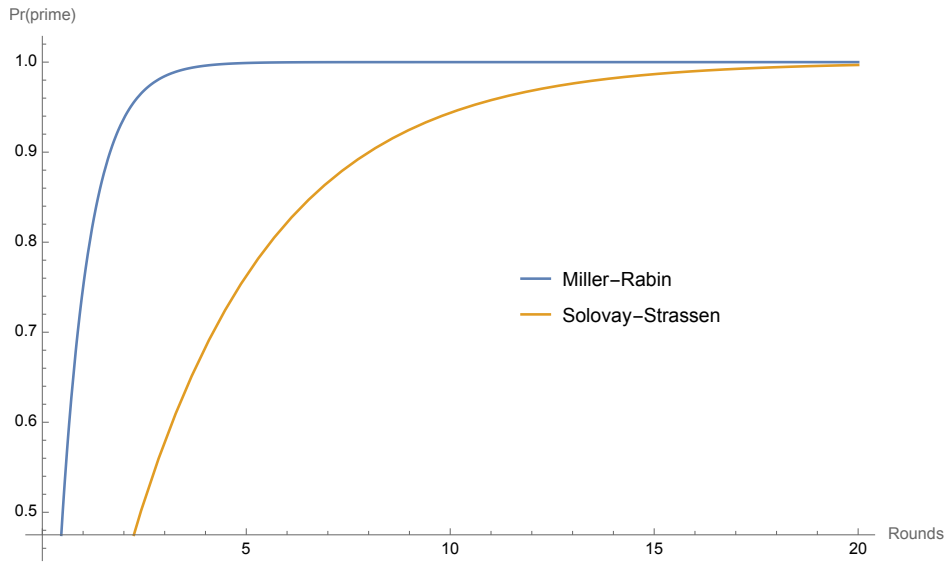

Figure 5: Pr[prime(p)] after successfully passing a given number of rounds.

```
MILLER-RABIN(n, k)
 1  write n − 1 = 2^s r such that r is odd
 2  for i ← 1 to k
 3       choose random a ∈ {2, 3, ..., n − 2}
 4       y = POWER-MOD(a, r, n)
 5       if y ≠ 1 and y ≠ n − 1
 6            j ← 1
 7            while j ≤ s − 1 and y ≠ n − 1
 8                 y ← POWER-MOD(y, 2, n)
 9                 if y == 1
10                      return FALSE
11                 j ← j + 1
12            if y ≠ n − 1
13                 return FALSE
14  return TRUE
```

The function that you are expected to implement to perform primality testing should be declared as follows:

`void is_prime(mpz_t n, uint64_t iters)`

Conducts the Miller-Rabin primality test to indicate whether or not `n` is prime using `iters` number of Miller-Rabin iterations. This function is needed when creating the two large primes `p` and `q` in RSA, verifying if a large integer is a prime.

In addition to the `is_prime()` function, you are also required to implement the following function:

`void make_prime(mpz_t p, uint64_t bits, uint64_t iters)`

Generates a new prime number stored in `p`. The generated prime should be at least `bits` number of bits long. The primality of the generated number should be tested using `is_prime()` using `iters` number of iterations.

6.3 Modular Inverses

The Euclidean algorithm, also called Euclid's algorithm, is an efficient method for computing the greatest common divisor (gcd) of two integers, the largest number that divides them both with a zero remainder. The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if their difference replaces the larger number with the smaller number. Since this replacement reduces the larger of the two numbers, repeating this process gives successively smaller pairs of numbers until the two numbers become equal. We can accomplish this much faster if we compute the remainder, which is equivalent to subtracting the smaller number from the larger until it is no longer larger. You will first want to implement the following function to compute the greatest common divisor of two integers, which should be defined as follows:

`void gcd(mpz_t d, mpz_t a, mpz_t b)`

Computes the greatest common divisor of `a` and `b`, storing the value of the computed divisor in `d`.

GCD(a, b)

```
1  while b ≠ 0
2      t ← b
3      b ← a mod b
4      a ← t
5  return a
```

The extended Euclidean algorithm is an extension to the Euclidean algorithm, and computes, in addition to the greatest common divisor (gcd) of integers a and b, also the coefficients of Bézout's identity, which are integers x and y such that

$$ax + by = \gcd(a, b).$$

The extended Euclidean algorithm is particularly useful when a and b are coprime. With that provision, x is the modular multiplicative inverse of $a \pmod{b}$, and y is the modular multiplicative inverse of $b \pmod{a}$.

Bézout's identity asserts that a and n are coprime if and only if there exist integers s and t such that

$$ns + at = 1.$$

Reducing this identity modulo n gives

$$at \equiv 1 \pmod{n}.$$

To adapt the extended Euclidean algorithm to the problem of computing the multiplicative inverse, note that the Bézout coefficient of n is not needed and so does not need to be computed. Also, for getting a positive and result that is less than n, use the fact that the integer t provided by the algorithm satisfies $|t| < n$. That is, if $t < 0$, add n to it at the end.

MOD-INVERSE(a, n)

```
 1  (r, r′) ← (n, a)
 2  (t, t′) ← (0, 1)
 3  while r′ ≠ 0
 4      q ← ⌊r/r′⌋
 5      (r, r′) ← (r′, r − q × r′)
 6      (t, t′) ← (t′, t − q × t′)
 7  if r > 1
 8      return no inverse
 9  if t < 0
10      t ← t + n
11  return t
```

The function that you are expected to implement to compute modular inverses should be declared as follows:

`void mod_inverse(mpz_t i, mpz_t a, mpz_t n)`

Computes the inverse `i` of `a` modulo `n`. In the event that a modular inverse cannot be found, set `i` to 0. Note that this pseudocode uses parallel assignments, which **C** *does not* support. Thus, you will need to use auxiliary variables to fake the parallel assignments.

7 An RSA Library

`void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters)`

Creates parts of a new RSA public key: two large primes `p` and `q`, their product `n`, and the public exponent `e`.

1. Begin by creating primes `p` and `q` using `make_prime()`. We first need to decide the number of bits that go to each prime respectively such that $\log_2(n) \geq$ `nbits`. Let the number of bits for `p` be a random number in the range $[\texttt{nbits}/4, (3 \times \texttt{nbits})/4)$. The remaining bits will go to `q`. The number of Miller-Rabin iterations is specified by `iters`.
2. Next, compute $\varphi(n) = (p-1)(q-1)$.
3. We now need to find a suitable public exponent `e`. In a loop, generate random numbers of around `nbits` using `mpz_urandomb()`. Compute the `gcd()` of each random number and the computed totient. Stop the loop you have found a number coprime with the totient: that will be the public exponent.

`void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)`

Writes a public RSA key to `pbfile`. The format of a public key should be `n`, `e`, `s`, then the `username`, each of which are written with a trailing newline. The values `n`, `e`, and `s` should be written as *hexstrings*. See the GMP functions for formatted output for help with writing hexstrings.

`void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)`

Reads a public RSA key from `pbfile`. The format of a public should be `n`, `e`, `s`, then the `username`, each of which should have been written with a trailing newline. The values `n`, `e`, and `s` should have been written as *hexstrings*. See the GMP functions for formatted input for help with reading hexstrings.

`void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)`

Creates a new RSA private key `d` given primes `p` and `q` and public exponent `e`. To compute `d`, simply compute the inverse of `e` modulo $\varphi(n) = (p-1)(q-1)$.

`void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)`

Writes a private RSA key to `pvfile`. The format of a private key should be `n` then `d`, both of which are written with a trailing newline. Both these values should be written as hexstrings.

`void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)`

Reads a private RSA key from `pvfile`. The format of a private key should be `n` then `d`, both of which should have been written with a trailing newline. Both these values should have been written as *hexstrings*.

`void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)`

Performs RSA encryption, computing ciphertext `c` by encrypting message `m` using public exponent `e` and modulus `n`. Remember, encryption with RSA is defined as $E(m) = c = m^e \pmod{n}$.

`void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)`

Encrypts the contents of `infile`, writing the encrypted contents to `outfile`. The data in `infile` should be in encrypted in *blocks*. Why not encrypt the entire file? Because of `n`. We are working modulo `n`, which means that the value of the block of data we are encrypting must be strictly less than `n`. We have two additional restrictions on the values of the blocks we encrypt:

1. The value of a block cannot be 0: $E(0) \equiv 0 \equiv 0^e \pmod{n}$.
2. The value of a block cannot be 1. $E(1) \equiv 1 \equiv 1^e \pmod{n}$.

A solution to these additional restrictions is to simply *prepend* a single byte to the front of the block we want to encrypt. The value of the prepended byte will be `0xFF`. This solution is not unlike the padding schemes such as PKCS and OAEP used in modern constructions of RSA. To encrypt a file, follow these steps:

1. Calculate the block size k. This should be $k = \lfloor (\log_2(n) - 1)/8 \rfloor$.
2. Dynamically allocate an array that can hold k bytes. This array should be of type `(uint8_t *)` and will serve as the block.
3. Set the zeroth byte of the block to `0xFF`. This effectively prepends the workaround byte that we need.
4. While there are still unprocessed bytes in `infile`:
   (a) Read at most $k-1$ bytes in from `infile`, and let j be the number of bytes actually read. Place the read bytes into the allocated block starting from index 1 so as to not overwrite the `0xFF`.
   (b) Using `mpz_import()`, convert the read bytes, including the prepended `0xFF` into an `mpz_t m`.
   (c) Encrypt `m` with `rsa_encrypt()`, then write the encrypted number to `outfile` as a *hexstring* followed by a trailing newline.

`void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)`

Performs RSA decryption, computing message `m` by decrypting ciphertext `c` using private key `d` and public modulus `n`. Remember, decryption with RSA is defined as $D(c) = m = c^d \pmod{n}$.

`void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)`

Decrypts the contents of `infile`, writing the decrypted contents to `outfile`. The data in `infile` should be decrypted in *blocks* to mirror how `rsa_encrypt_file()` encrypts in blocks. To decrypt a file, follow these steps:

1. Calculate the block size k. This should be $k = \lfloor (\log_2(n) - 1)/8 \rfloor$.
2. Dynamically allocate an array that can hold k bytes. This array should be of type `(uint8_t *)` and will serve as the block.
3. While there are still unprocessed bytes in `infile`:
   (a) Scan in a hexstring, saving the hexstring as a `mpz_t c`. Remember, each block is written as a hexstring with a trailing newline when encrypting a file.
   (b) Using `mpz_export()`, convert `c` back into bytes, storing them in the allocated block. Let j be the number of bytes actually converted.
   (c) Write out $j-1$ bytes starting from index 1 of the block to `outfile`. This is because index 0 must be prepended `0xFF`. Do not output the `0xFF`.

`void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)`

Performs RSA signing, producing signature `s` by signing message `m` using private key `d` and public modulus `n`. Signing with RSA is defined as $S(m) = s = m^d \pmod{n}$.

`bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)`

Performs RSA verification, returning `true` if signature `s` is verified and `false` otherwise. Verification is the inverse of signing. Let $t = V(s) = s^e \pmod{n}$. The signature is verified if and only if `t` is the same as the expected message `m`.

8 Key Generator

Your key generator program should accept the following command-line options:

- `-b`: specifies the minimum bits needed for the public modulus `n`.
- `-i`: specifies the number of Miller-Rabin iterations for testing primes (default: 50).
- `-n pbfile`: specifies the public key file (default: rsa.pub).
- `-d pvfile`: specifies the private key file (default: rsa.priv).
- `-s`: specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch, given by `time(NULL)`).
- `-v`: enables verbose output.
- `-h`: displays program synopsis and usage.

The program should follow these steps:

1. Parse command-line options using `getopt()` and handle them accordingly.
2. Open the public and private key files using `fopen()`. Print a helpful error and exit the program in the event of failure.
3. Using `fchmod()` and `fileno()`, make sure that the private key file permissions are set to `0600`, indicating read and write permissions for the user, and no permissions for anyone else.
4. Initialize the random state using `randstate_init()`, using the set seed.
5. Make the public and private keys using `rsa_make_pub()` and `rsa_make_priv()`, respectively.
6. Get the current user's name as a string. You will want to use `getenv()`.
7. Convert the username into an `mpz_t` with `mpz_set_str()`, specifying the base as 62. Then, use `rsa_sign()` to compute the signature of the username.
8. Write the computed public and private key to their respective files.
9. If verbose output is enabled print the following, each with a trailing newline, in order:
   (a) username
   (b) the signature `s`
   (c) the first large prime `p`
   (d) the second large prime `q`

(e) the second large prime `q`

(f) the public modulus `n`

(g) the public exponent `e`

(h) the private key `d`

All of the `mpz_t` values should be printed with information about the number of bits that constitute them, along with their respective values in *decimal*. See the reference key generator program for an example.

10. Close the public and private key files, clear the random state with `randstate_clear()`, and clear any `mpz_t` variables you may have used.

9 Encryptor

Your encryptor program should accept the following command-line options:

- `-i`: specifies the input file to encrypt (default: `stdin`).
- `-o`: specifies the output file to encrypt (default: `stdout`).
- `-n`: specifies the file containing the public key (default: `rsa.pub`).
- `-v`: enables verbose output.
- `-h`: displays program synopsis and usage.

The program should follow these steps:

1. Parse command-line options using `getopt()` and handle them accordingly.
2. Open the public key file using `fopen()`. Print a helpful error and exit the program in the event of failure.
3. Read the public key from the opened public key file.
4. If verbose output is enabled print the following, each with a trailing newline, in order:

   (a) username

   (b) the signature `s`

   (c) the public modulus `n`

   (d) the public exponent `e`

   All of the `mpz_t` values should be printed with information about the number of bits that constitute them, along with their respective values in *decimal*. See the reference encryptor program for an example.

5. Convert the username that was read in to an `mpz_t`. This will be the expected value of the verified signature. Verify the signature using `rsa_verify()`, reporting an error and exiting the program if the signature couldn't be verified.
6. Encrypt the file using `rsa_encrypt_file()`.
7. Close the public key file and clear any `mpz_t` variables you have used.

10 Decryptor

Your decryptor program should accept the following command-line options:

- `-i`: specifies the input file to decrypt (default: `stdin`).
- `-o`: specifies the output file to decrypt (default: `stdout`).
- `-n`: specifies the file containing the private key (default: `rsa.priv`).
- `-v`: enables verbose output.
- `-h`: displays program synopsis and usage.

The program should follow these steps:

1. Parse command-line options using `getopt()` and handle them accordingly.
2. Open the private key file using `fopen()`. Print a helpful error and exit the program in the event of failure.
3. Read the private key from the opened private key file.
4. If verbose output is enabled print the following, each with a trailing newline, in order:
   (a) the public modulus `n`
   (b) the private key `e`

   Both these values should be printed with information about the number of bits that constitute them, along with their respective values in *decimal*. See the reference decryptor program for an example.
5. Decrypt the file using `rsa_decrypt_file()`.
6. Close the private key file and clear any `mpz_t` variables you have used.

11 Deliverables

You will need to turn in the following source code and header files:

1. `decrypt.c`: This contains the implementation and `main()` function for the `decrypt` program.
2. `encrypt.c`: This contains the implementation and `main()` function for the `encrypt` program.
3. `keygen.c`: This contains the implementation and `main()` function for the `keygen` program.
4. `numtheory.c`: This contains the implementations of the number theory functions.
5. `numtheory.h`: This specifies the interface for the number theory functions.
6. `randstate.c`: This contains the implementation of the random state interface for the RSA library and number theory functions.
7. `randstate.c`: This specifies the interface for initializing and clearing the random state.
8. `rsa.c`: This contains the implementation of the RSA library.
9. `rsa.h`: This specifies the interface for the RSA library.

Your code must pass `scan-build` *cleanly*. If there are any bugs or errors that are false positives, document them and explain why they are false positives in your `README.md`. Improper explanations will *not* be considered. You will also need to turn in the following:

1. `Makefile`:
    - `CC = clang` must be specified.
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified.
    - `pkg-config` to locate compilation and include flags for the GMP library must be used.
    - `make` must build the `encrypt`, `decrypt`, and `keygen` executables, as should `make all`.
    - `make decrypt` should build only the `decrypt` program.
    - `make encrypt` should build only the `encrypt` program.
    - `make keygen` should build only the `keygen` program.
    - `make clean` must remove all files that are compiler generated.
    - `make format` should format all your source code, including the header files.
2. `README.md`: This must use proper Markdown syntax and describe how to use your program and `Makefile`. It should also list and explain any command-line options that your program accepts. Any false positives reported by `scan-build` should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.
3. `DESIGN.pdf`: This document *must* be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.

12 Submission

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

*et ecce simia pallidus et qui nomen illi **C** et inferus sequebatur eum*