RSA Encryption and Decryption

Kevin Liao and Josh Smith

November 28, 2016

1 Introduction

The RSA cryptosystem is one of the most widely used public-key cryptosystems in use today for securing information. Fundamentally, it allows two parties to exchange a secret message who have never communicated in the past. To accomplish this, RSA utilizes a pair of keys, a public key for encryption and a private key for decryption. The encryption and decryption keys are distinct, and so RSA is often referred to as an asymmetric cryptosystem.

For this project, we propose to study the RSA cryptosystem to understand how and why it works. As one of the most mature cryptosystems, RSA has been studied extensively, and there are plenty of interesting resources on attacks and how to prevent them [1]. These attacks provide an excellent exposition for the dangers of improperly implementing RSA, which makes such a project well-suited for learning.

We will focus on the number theory behind the algorithm, well-known attacks on the RSA cryptosysem, and secure coding practices associated with implementing cryptosystems more broadly. Our ultimate goal is to implement the RSA encryption and decryption algorithms according to cryptographic considerations for security and performance, which we hope will provide a better understanding of the nuances of cryptographic coding in practice.

2 Implementation

We first detail our implementation of RSA key generation, and then detail our implementation of encryption and decryption.

2.1 Key Pair Generation

We follow the Digital Signature Standard (DSS) [2] issued by the National Institute of Standards and Technology (NIST) to generate key pairs.

2.1.1 Pseudorandom Number Generator

In order to generate random primes, it is important that we use a cryptographically secure pseudorandom number generator. We decide to use the UNIX-based special file `/dev/random`, which generates high-quality pseudorandom numbers that are well-suited for key generation.

The semantics for `/dev/random` vary based on the operating system. In Linux, `/dev/random` is generated from entropy created by keystrokes, mouse movements, IDE timings, and other kernel processes. In macOS, `/dev/random` data is generated using the Yarrow-160 algorithm, which is a cryptographic pseudorandom number generator. Yarrow-160 outputs random bits using a combination of the SHA1 hash function and three-key triple-DES.

We believe `/dev/random`, as prescribed, is sufficient for our purposes, but the entropy pool can be further improved using specialized programs or hardware random number generators.

2.1.2 Primality Testing

We use the Miller-Rabin probabilistic primality test to validate the generation of prime numbers. There are two approaches for using Miller-Rabin primality testing: (1) using several iterations of Miller-Rabin alone; (2) using several iterations of Miller-Rabin followed by a Lucas primality test. For simplicity, we use the iterative Miller-Rabin implementation available in the GNU MP Library. Instead, we find it more interesting to learn how to use Miller-Rabin testing correctly in practice, as specified in the DSS.

For example, different modulus lengths for RSA require varying rounds of Miller-Rabin testing. We reproduce the number of rounds necessary for various auxiliary prime (see Section 2.1.3) lengths in Table 1, and we follow this in our implementation.

Auxiliary Prime Length	Rounds of M-R Testing
> 100 bits	28
> 140 bits	38
> 170 bits	41

Table 1: The table shows the number of Miller-Rabin rounds necessary as a function of the lengths of auxiliary primes p_1, p_2, q_1, and q_2.

2.1.3 Criteria for Key Pairs

The key pair for RSA consists of the public key (n, e) and the private key (n, d). The RSA modulus n is the product of two distinct prime numbers p and q. RSA's security rests on the primality and secrecy of p and q, as well as the secrecy of the private exponent d. The methodology for generating these parameters varies based on the desired number of bits of security and the desired quality of primes. However, several desideratum must hold true for all methods.

Public Exponent e. The following constraints must hold true for the public exponent e.

1. The public verification exponent e must be selected prior to generating the primes p and q, and the private signature exponent d.

2. The public verification exponent e must be an odd positive integer such that $2^{16} < e < 2^{256}$.

It is immaterial whether or not e is a fixed value or a random value, as long as it satisfies constraint 2 above. For simplicity, we fix $e = 2^{16} + 1 = 65537$.

Primes p and q. The following constraints must hold true for random primes p and q.

1. Both p and q shall be either provable primes or probable primes.
2. Both p and q shall be randomly generated prime numbers such that all of the following subconstraints hold:
   - $(p + 1)$ has a prime factor p_1
   - $(p - 1)$ has a prime factor p_2
   - $(q + 1)$ has a prime factor q_1
   - $(q - 1)$ has a prime factor q_2

   where p_1, p_2, q_1, q_2 are auxiliary primes of p and q. Then, one of the following shall also apply:

   (i) p_1, p_2, q_1, q_2, p, and q are all provable primes

   (ii) p_1, p_2, q_1, q_2 are provable primes, and p and q are probable primes

   (iii) p_1, p_2, q_1, q_2, p, and q are all probable primes

For our implementation, we choose to generate probable primes p and q with conditions based on auxiliary probable primes p_1, p_2, q_1, and q_2. In other words, we choose the method (iii) listed above. While this method offers the lowest quality of primes, it offers the best performance. It would be interesting future work to benchmark key generation times and quality of primes among these three methods.

Method (iii) supports key sizes of length 1024, 2048, and 3072, which offers more utility over method (i), which offers only key sizes of length 2048 and 3072. For different key sizes, various lengths of auxiliary primes must be satisfied, which is reproduced in Table 2. Table 2 can be joined with Table 1 for a comprehensive view of parameters as a function of the key size *nlen*.

Key Size (*nlen*)	Minimum Length of Auxiliary Primes
1024 bits	> 100 bits
2048 bits	> 140 bits
3072 bits	> 170 bits

Table 2: The table shows the minimum length of auxiliary primes p_1, p_2, q_1, and q_2 as a function of the key size *nlen*.

Regarding our actual implementation of method (iii), we closely follow the constraints above and how probable primes are generated from probable auxiliary primes as specified in the DSS [2]. There are further constraints to the above, which are specific to method (iii), that we satisfy but do not fully detail here. However, one important aspect of method (iii) is that it leverages the Chinese Remainder Theorem to improve performance for key generation.

Private exponent d. The following constraints must hold true for the private exponent d.

1. The private exponent d must be a positive integer between

$$2^{nlen/2} < d < LCM(p-1, q-1). \tag{1}$$

2. $1 \equiv (ed) \pmod{LCM(p-1, q-1)}$.

Implementing constraints for the private exponent d is relatively straightforward. However, we do consider that in the rare case when $d \leq 2^{nlen/2}$, new primes must be generated.

2.2 Encryption and Decryption

3 Crypto Learning

Here, we overview a number of strengths and weaknesses of our RSA implementation. In particular, we discuss attacks that we do protect against, and attacks that would cause our implementation to fail.

3.1 Attacks via Insecure PRNGs

We generate pseudorandom numbers using the `/dev/random` file, as specified in Section 2.1.1. This is considered a cryptographically secure method for generating pseudorandom numbers and is widely used in practice. Even so, there exist several theoretical attacks on Linux's implementation of this PRNG.

Gutterman *et al.* perform an analysis of Linux's pseudorandom number generator (LRNG) and expose a number of security vulnerabilities [3]. More specifically, they reverse engineer LRNG and show that given the current state of the generator, it is possible to reconstruct previous states, thereby compromising the security of past usage. Further, they show that it is possible to measure and analyze the entropy created by the kernel. Bernstein presents a related attack in which monitoring one source of entropy could compromise the randomness of other sources of entropy [4].

While the latter attacks are theoretical, and to our knowledge have not been successful in practice, Gutterman also presents a denial of service attack that our implementation is susceptible to [3]. Since Linux's implementation of `/dev/random` may block the output of bits when the entropy is low, one simple attack would be to simply read all the bits from `/dev/random`, thereby blocking other users' access to new bits for a long period of time. More interestingly, an attack can also be performed remotely by triggering system requests for `get_random_bytes`, which will block both `/dev/random` and the non-blocking `/dev/urandom` pool.

One possible solution is to limit the per user consumption of random bits. Alternatively, we could avoid using `/dev/random` altogether and instead generate pseudorandom numbers via hardware random number generators.

3.2 Common Modulus Attack

While the common modulus attack is simple, it is a case in point for the dangers of misusing RSA [1].

In order to prevent having to generate a different modulus n for different users, a developer might choose to fix n for a number of users or for all users. This is insecure, since a user could use his/her own exponents e and d to factor the fixed n, thereby recovering the private key d from some other user. Thus, the common modulus attack shows that the RSA modulus should not be fixed. Our implementation precludes this attack by generating a random modulus every time. This is done through calls to the `gen_primes` function.

3.3 Low Private Exponent Attack

In order to reduce the decryption time, a developer might choose a smaller value for the private exponent d rather than a random value. Choosing a small d can improve decryption performance (modular exponentation) by a factor of at least 10 for a 1024-bit modulus. However, Weiner shows that such a simplification is completely insecure [5]. Boneh and Durfee further improve the bounds of Weiner's attack, showing that $d < n^{0.292}$ is susceptible to attack [6]. There are two techniques to prevent this attack; both of which our implementation supports.

The first technique is to use a large public exponent e. Weiner shows that as long as $e > n^{1.5}$, this attack cannot be performed. In our implementation, we fix $e = 65537$. Thus, for $nlen = 1024$, our implementation supports this technique. However, this technique does not hold true for $nlen = 2048$ or $nlen = 3072$. This can be easily fixed by increasing e to satisfy $nlen = 3072$, however, the downside is that it will increase encryption time. Nonetheless, the second technique, using the Chinese Remainder Theorem to speed up decryption, is fully supported by our implementation.

3.4 Low Public Exponent Attack

Similar to the latter attack, in order to reduce the encryption time, a developer might choose a smaller value for the public exponent e. This engenders a number of attacks on low public exponents, most of which are based on Coppersmith's theorem [7]. While the smallest e possible is 3, $e \geq 2^{16} + 1$ is recommended to prevent certain attacks. This is the value of e that we use in our implementation. It is simple to increase e for security, but this will result in a performance decline.

3.5 Partial Key Exposure Attack

Suppose that for a given private key (n, d), some portion of the private exponent d is exposed. Boneh *et al.* show that recovering the rest of the private exponent d is possible when the corresponding private exponent e is small. Specifically, they show that it is possible to reconstruct all of d as long as $e < \sqrt{n}$. In our implementation, $e = 65537$ and all $nlen$ are secure from such an attack. However, partial key exposure attacks do illustrate the importance of keeping the entire private key secret. This is one consideration that our implementation is lacking, and it will be interesting to explore this in the future.

3.6 Side-Channel Attacks

Kocher's seminal cryptanalysis of RSA via a timing attack shows that a clever attacker could measure the amount of time it takes for RSA decryption, thereby recovering the private exponent d [8]. Our implementation does not protect against such timing attacks, but there are two solutions that can be considered.

The first is to introduce a delay so that decryption (modular exponentiation, in particular) takes a fixed amount of time. However, this would cause a decline in performance. The second solution is based on blinding, by which a randomization is introduced such that decryption is performed on a random message unknown to the attacker. Thus, such timing attacks cannot be performed.

Kocher also discovered another side-channel attack by measuring the amount of power consumed during decryption. Since multiprecision multiplication causes greater power consumption, it is simple to detect the number of multiplications, thereby revealing information about the private exponent d.

4 Secure Coding

We next overview secure coding practices that we considered for our implementation, as well as practices that could have further improved our code. These are mostly based on the SEI CERT C Coding Standard [9].

4.1 Integers and Floats

Handling multiple precision integers and multiple precision floats and understanding conversions between these data types is crucial in implementing RSA.

In regards to integers, we use different types of integers (i.e. `int`, ûnsigned long int, and `mpz_t` (multiple precision integers) for different purposes. For general purpose counters, we can safely use the `int` data type. For representing the size of an object, we can safely use the `size_t` data type, since this generally covers the entire address space. For any integers that may be used in multiple precision arithmetic, we err on the side of caution and use the `unsigned long int` data type. Then finally, for any integers that require multiple precision, we use the `mpz_t` data type from the GMP Library.

In regards to floating point numbers, we simply use the `mpf_t` data type from the GMP Library, since their use is limited and the multiple precision float data type offers enough utility for the required use cases.

Further, we also perform adequate range checking, integer overflow checking, and truncation checking. For the generation of key parameters, it is crucial that we perform range checking thoroughly, since a single misstep could lead to an incorrect encryption or decryption. Additionally, we err on the side of caution and instantiate integers as either `long int` or `mpz_t` to prevent integer overflows. Finally, we pay attention to any truncation that may occur as a result of conversions between integers and floats. For example, it is important to consider that while a multiple precision integer square root function is available, the result is truncated to an integer. Thus, we must handle such operations more precisely using the `mpf_t` (multiple precision float) object.

4.2 Memory Management

Since memory owned by our process can be accessed and reused by another process in the absence of proper memory management, this could potentially reveal information about secret keys to other processes. Even further, systems with multiple users make it possible for one user to sniff keys from another users' process. Thus, proper memory management is crucial for the secrecy of private keys.

In this regard, we free dynamically allocated memory whenever it is no longer needed. This occurs throughout our implementation in two fashions. First, consider when a new block of memory is allocated using `malloc`. Once the allocated block of memory is no longer in use, memory is freed using the function call `free`. Second, when using the GMP Library to instantiate multiple precision numbers, these numbers are also dynamically allocated. Thus, this memory must either be freed using the function call `mpz_clear` (for integers) or "zeroized" to ensure that no information about the secret keys are revealed.

Another aspect of memory management we consider is allocating sufficient memory for an object. In `malloc` calls, we consistently and correctly use `sizeof` to size the memory allocation. For initializing multiple precision numbers, this is done automatically.

4.3 Characters and Strings

One secure coding practice that we should have considered is to cast characters to `unsigned char` before converting them to larger integer sizes. One instance of this is when generating pseudorandom numbers from `/dev/random`, since we sample random characters from this file and then convert it to a pseudorandom multiple precision integer. More broadly, any arguments to character-handling functions should be represented as an `unsigned char`. However, this is only applicable to platforms in which `char` data types have the same representation as `signed char` data types.

4.4 Error Handling

Another secure coding practice that we should have considered is to handle errors throughout the entire program. Although there are instances in which we do handle errors, our program would be much more robust if it detected and handled all standard library errors and GMP Library errors. Having a consistent and comprehensive error-handling policy would improve our implementation's resilience in the face of erroneous or malicious inputs, hardware or software faults, and unexpected environment changes. This would be advantageous both to the developers as well as the end-users of our implementation.

4.5 Test Suite

It would have been beneficial to set up a comprehensive test suite, which could rigorously test the modules within our implementation. Alternatively, we could have used a fuzzer to exercise the logic of our implementation. In the future, we can leverage static analysis techniques and a binary fuzzer, such as American Fuzzy Lop (AFL), to discover any bugs or vulnerabilities in our code.

5 Summary

References

[1] Dan Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[2] PUB FIPS. 186-4. *Digital Signature Standard (DSS)*, 2013.

[3] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.

[4] Daniel Bernstein. Entropy attacks! 2014.

[5] Michael J Wiener. Cryptanalysis of short rsa secret exponents. *IEEE Transactions on Information theory*, 36(3):553–558, 1990.

[6] Dan Boneh and Glenn Durfee. New results on the cryptanalysis of low exponent rsa. *IEEE Transactions on Information Theory*, 46(4):1339–1349, 2000.

[7] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.

[8] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[9] Robert C Seacord. *The CERT C secure coding standard*. Pearson Education, 2008.

A Code

Listing 1: Code for `rsa.h`.

```
/*
 * Data Types
 */
struct RSAPublicKey {
        mpz_t modulus;
        mpz_t publicExponent;
};

struct RSAPrivateKey {
        mpz_t modulus;
        mpz_t privateExponent;
};

/*
 * Methods
 */
char*   I2OSP                           (mpz_t x, int xLen);
void    OS2IP                           (char *X, mpz_t x);
int             RSAEP                           (struct
    RSAPublicKey *K, mpz_t m, mpz_t c);
```

```
int              RSADP                            (struct RSAPrivateKey *K, mpz_t c, mpz_t m);
char*   MGF1                              (char *mgfSeed, unsigned long long maskLen);
char*   RSAES_OAEP_ENCRYPT      (struct RSAPublicKey *K, char *M, char *L);
char*   RSAES_OAEP_DECRYPT      (struct RSAPrivateKey *K, char *C, char *L);
```

Listing 2: Code for `rsa.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <time.h>
#include <gmp.h>
#include <openssl/sha.h>
#include "rsa.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <math.h>
#include <assert.h>

// Convert nonnegative integer x to a zero-padded octet string of length xLen.
char* I2OSP(mpz_t x, int xLen) {
    size_t osLen = mpz_sizeinbase(x, 16);
    xLen *= 2;
    if (xLen < osLen) {
        printf("integer too large\n");
        return NULL;
    }
    char *os = malloc((xLen + 1) * sizeof(char));
    memset(os, '0', xLen - osLen);
    mpz_get_str(os + xLen - osLen, 16, x);
    os[xLen] = '\0';
    return os;
}

// Convert octet string to a nonnegative integer
void OS2IP(char *X, mpz_t x) {
        mpz_set_str(x, X, 16);
}

// RSA Encryption Primative
int RSAEP(struct RSAPublicKey *K, mpz_t m, mpz_t c) {
        if (mpz_cmp(m, K->modulus) <= 0) {
                printf("message representative out of range\n");
                return 0;
        }
        mpz_powm_sec(c, m, K->publicExponent, K->modulus);
        return 1;
}

// RSA Decryption Primative
int RSADP(struct RSAPrivateKey *K, mpz_t c, mpz_t m) {
        if (mpz_cmp(c, K->modulus) <= 0) {
                printf("ciphertext representative out of range\n");
                return 0;
        }
        mpz_powm_sec(m, c, K->privateExponent, K->modulus);
```

```c
        return 1;
}

// Mask generation function specified in PKCS #1 Appendix B.
char* MGF1(char *mgfSeed, unsigned long long maskLen) {

    // Step 1: Verify maskLen <= (hLen * 2^32)
    unsigned long long hLen = SHA256_DIGEST_LENGTH;
    if (maskLen > (hLen << 32)) {
        printf("mask too long\n");
        return NULL;
    }
    maskLen *= 2;
    hLen *= 2;

    // Step 2: Init T to empty octet string. T consists of TLen SHA256 hashes.
    int TLen = (maskLen + hLen - 1) / hLen;
    char *T = malloc((TLen * hLen) * sizeof(char));

    char *TPtr = T;
    char *hashOp;
    size_t mgfSeedLen = strlen(mgfSeed);
    hashOp = malloc((mgfSeedLen + 4 * 2) * sizeof(char));
    memcpy(hashOp, mgfSeed, mgfSeedLen);

    // Step 3: Generate mask
    int i, j;
    char *C;
    unsigned char *hash;
    unsigned char hChar;
    hash = malloc(SHA256_DIGEST_LENGTH * sizeof(char));
    mpz_t counter;
    mpz_init(counter);
    for (i = 0; i < TLen; ++i) {
        mpz_set_ui(counter, i);
        C = I2OSP(counter, 4);
        memcpy(hashOp + mgfSeedLen, C, 4 * 2);
        SHA256(hashOp, mgfSeedLen + 4 * 2, hash);
        for (j = 0; j < hLen; j += 2)
            sprintf(TPtr + j, "%02x", hash[j/2]);
        TPtr += hLen;
        free(C);
    }

    // Step 4: Output mask
    char *mask = malloc(maskLen + 1);
    memcpy(mask, T, maskLen);
    mask[maskLen] = '\0';
    free(hash); free(hashOp); free(T);
    return mask;
}

// Temporary function for generating random octet strings.
char* randOS(int length) {
        length *= 2;
        srand(time(NULL));

        int i;
        char *str = malloc(length + 1);
        for (i = 0; i < length; i += 2)
                sprintf(str + i, "%02x", (unsigned char)(rand() %
```

```
                    256));
        str[length] = '\0';
        return str;
}

// M and L are octet strings with no whitespace
char* RSA_OAEP_ENCRYPT(struct RSAPublicKey *K, char* M, char *L) {

        // Step 1: Length checking (*_o stores size in octets; *_h in hex chars)
        size_t k_o = (mpz_sizeinbase(K->modulus, 16) + 1) / 2;
        size_t hLen_o = SHA256_DIGEST_LENGTH;
        size_t mLen_o = strlen(M) / 2;
        size_t maxmLen_o = k_o - 2 * hLen_o - 2;
        if (mLen_o > maxmLen_o) {
                printf("message too long\n");
                return NULL;
        }
        size_t k_h = k_o * 2;
        size_t hLen_h = hLen_o * 2;
        size_t mLen_h = mLen_o * 2;     // If M is valid, then mLen_h = strlen(M)

        // Step 2: EME-OAEP encoding
        if (L == NULL) L = "";
        char *lHash = SHA256(L, strlen(L), NULL);

        // b. Generate random padding string (PS)
        size_t PSLen_h = (maxmLen_o - mLen_o) * 2;
        char *PS = malloc(PSLen_h * sizeof(char));
        memset(PS, '0', PSLen_h);

        // c. Generate data block (DB)
        size_t DBLen_o = k_o - hLen_o - 1;
        size_t DBLen_h = DBLen_o * 2;
        char *DB = malloc(DBLen_h * sizeof(char));
        int i;
        for (i = 0; i < hLen_o; ++i)
                sprintf(DB + 2 * i, "%02x", lHash[i]);
        memcpy(DB + hLen_h, PS, PSLen_h);
        memcpy(DB + hLen_h + PSLen_h, "01", 2);
        memcpy(DB + DBLen_h - mLen_h, M, mLen_h);

        // d. Generate random seed
        char *seed = randOS(hLen_o);

        // ef. Generate dbMask and compute DB XOR dbMask
        char *dbMask = MGF1(seed, DBLen_o);
        char *maskedDB = malloc(DBLen_h * sizeof(char));
        for (i = 0; i < DBLen_h; ++i)
                maskedDB[i] = DB[i] ^ dbMask[i];

        // gh. Generate seedMask and compute seed XOR seedMask
        char *seedMask = MGF1(seed, hLen_o);
        char *maskedSeed = malloc(hLen_h * sizeof(char));
        for (i = 0; i < hLen_h; ++i)
                maskedSeed[i] = seed[i] ^ seedMask[i];

        // i. Generate encoded message (EM)
        size_t EMLen_h = hLen_h + DBLen_h + 2;
        char *EM = malloc((EMLen_h + 1) * sizeof(char));
        memset(EM, 0, 2);
```

```
        memcpy(EM + 2, maskedSeed, hLen_h);
        memcpy(EM + hLen_h, maskedDB, DBLen_h);
        EM[EMLen_h] = '\0';

        // Step 3-4: RSA encryption
        mpz_t m, c;
        mpz_init(m);
        mpz_init(c);
        OS2IP(EM, m);
        RSAEP(K, m, c);
        char *C = I2OSP(c, k_o);

        // Free memory
        free(PS); free(DB); free(dbMask); free(maskedDB);
        free(seedMask); free(maskedSeed); free(EM);
        mpz_clear(m); mpz_clear(c);

        return C;
}

char *RSA_OAEP_DECRYPT(struct RSAPrivateKey *K, char* C, char *L) {

        // Step 1: Length checking (*_o stores sizes in octets; *_h in hex chars)
        size_t k_o = (mpz_sizeinbase(K->modulus, 16) + 1) / 2;
        size_t CLen_o = sizeof(C) / 2;
        if (k_o != CLen_o) {
                printf("decryption␣error\n");
                return NULL;
        }
        size_t hLen_o = SHA256_DIGEST_LENGTH;
        if (k_o < (2 * hLen_o + 2)) {
                printf("decryption␣error\n");
                return NULL;
        }

        // Step 2: RSA Decryption
        mpz_t c, m;
        mpz_init(c);
        mpz_init(m);
        OS2IP(C, c);
        if (!RSADP(K, c, m)) {
                printf("decryption␣error\n");
                return NULL;
        }
        char *EM = I2OSP(m, k_o);

        // Step 3: EME-OAEP decoding
        if (L == NULL) L = "";
        size_t hLen_h = hLen_o * 2;
        char *lHash_o = malloc(hLen_o * sizeof(char));
        char *lHash_h = malloc(hLen_h * sizeof(char));
        SHA256(L, strlen(L), lHash_o);
        int i;
        for (i = 0; i < hLen_o; ++i)
                sprintf(lHash_h + 2 * i, "%02x", lHash_o[i]);

        // b. Separate encoded message (EM) into its component parts
        size_t DBLen_o = k_o - hLen_o - 1;
        size_t DBLen_h = DBLen_o * 2;
        char *maskedSeed = malloc((hLen_h + 1) * sizeof(char));
```

```
        char *maskedDB = malloc((DBLen_h + 1) * sizeof(char));
        memcpy(maskedSeed, EM + 2, hLen_h);
        memcpy(maskedDB, EM + 2 + hLen_h, DBLen_h);
        maskedSeed[hLen_h] = '\0';
        maskedDB[DBLen_h] = '\0';

        // cd. Generate seedMask and compute maskedSeed XOR seedMask
        char *seedMask = MGF1(maskedDB, hLen_o);
        char *seed = malloc((hLen_h + 1) * sizeof(char));
        for (i = 0; i < hLen_h; ++i)
                seed[i] = maskedSeed[i] ^ seedMask[i];
        seed[hLen_h] = '\0';

        // ef. Generate dbMask and compute maskedDB XOR dbMask
        char *dbMask = MGF1(seed, DBLen_o);
        char *DB = malloc((DBLen_h + 1) * sizeof(char));
        for (i = 0; i < DBLen_h; ++i)
                DB[i] = maskedDB[i] ^ dbMask[i];
        DB[DBLen_h] = '\0';

        // g. Separate data block (DB) into component parts to recover message
        size_t PSLen_h = strlen(DB + hLen_h);
        int mLen_h = DBLen_h - PSLen_h - hLen_h - 1;
        if (mLen_h < 0) {
                printf("decryption error");
                return NULL;
        }
        if (EM[0] != '0' || EM[1] != '0') {
                printf("decryption error");
                return NULL;
        }
        if (strncmp(DB, lHash_h, hLen_h) != 0) {
                printf("decryption error");
                return NULL;
        }
        char *M = malloc((mLen_h + 1) * sizeof(char));
        memcpy(M, DB + DBLen_h - mLen_h, mLen_h);
        M[mLen_h] = '\0';
        return M;
}

void PRNG(mpz_t rand, int n) {

    int devrandom = open("/dev/random", O_RDONLY);
    char randbits[n/8];
    size_t randlen = 0;
    while (randlen < sizeof randbits) {

        ssize_t result = read(devrandom, randbits + randlen, (sizeof randbits) - randlen);
        if (result < 0)
            printf("%s\n", "Could not read from /dev/random");
        randlen += result;
    }
    close(devrandom);

    mpz_import(rand, sizeof(randbits), 1, sizeof(randbits[0]), 0, 0, randbits);
    // Make sure rand is odd
    if (mpz_odd_p(rand) == 0) {
```

```
        unsigned long int one = 1;
        mpz_add_ui(rand, rand, one);
    }
}

void gen_e(mpz_t e) {
    // Set e to 2^16 + 1
    unsigned long int e_int = pow(2,16)+1;
    mpz_set_ui(e, e_int);
}

void gen_d(mpz_t d, mpz_t p_minus_1, mpz_t q_minus_1, mpz_t e, int n) {

    unsigned long int one = 1;
    mpz_t lower_bound, upper_bound, base;
    mpz_init(lower_bound); mpz_init(upper_bound); mpz_init_set_str(base, "2", 10);
    mpz_pow_ui(lower_bound, base, n/2);
    mpz_lcm(upper_bound, p_minus_1, q_minus_1);

    mpz_invert(d, e, upper_bound);
    if (mpz_cmp(d, lower_bound) < 0 || mpz_cmp(d, upper_bound) > 0) {
        fprintf(stderr, "Private exponent d too small, try again\m");
        exit(-1);
    }

    mpz_t ed, check_d;
    mpz_init(ed); mpz_init(check_d);

    mpz_mul(ed, e, d);
    mpz_mod(check_d, ed, upper_bound);

    assert(mpz_cmp_ui(check_d, one) == 0);

}

void gen_probable_prime(mpz_t p, mpz_t p1, mpz_t p2, mpz_t e, int n) {

    // Step 1: Check if p1 and p2 are coprime
    mpz_t gcd, twop1;
    mpz_init(gcd); mpz_init(twop1);
    unsigned long int one = 1;
    unsigned long int two = 2;
    mpz_mul_ui(twop1, p1, two);
    mpz_gcd(gcd, twop1, p2);
    if (mpz_cmp_ui(gcd, one) != 0) {
        fprintf(stderr, "Auxiliaries p1 and p2 not coprime\n");
        exit(-1);
    }

    // Step 2: Chinese remainder theorem
    mpz_t R; mpz_t R1; mpz_t R2;
    mpz_init(R); mpz_init(R1); mpz_init(R2);

    mpz_invert(R1, p2, twop1);
    mpz_mul(R1, R1, p2);

    mpz_invert(R2, twop1, p2);
```

```
    mpz_mul(R2, R2, twop1);

    mpz_sub(R, R1, R2);

    // Check for CRT
    mpz_t check1; mpz_t check2; mpz_t mpz_one;
    mpz_init(check1); mpz_init(check2); mpz_init(mpz_one);
    mpz_set_str(mpz_one, "1", 10);
    mpz_mod(check1, R, twop1);
    mpz_mod(check2, R, p2);
    mpz_sub(check2, p2, check2);
    assert(mpz_cmp(check1, mpz_one) == 0);
    assert(mpz_cmp(check2, mpz_one) == 0);


    // Step 3: Generate random X between lower_bound and upper_bound
    mpz_t lower_bound; mpz_t upper_bound; mpz_t base; mpz_t X; mpz_t temp; mpz_t Y;
    mpz_init(lower_bound); mpz_init(upper_bound); mpz_init(base); mpz_init(X); mpz_init(temp); mpz_init(Y);

    mpz_set_str(base, "2", 10);
    mpz_pow_ui(upper_bound, base, n/2);
    mpz_sub_ui(upper_bound, upper_bound, one);


    mpf_t f_lb, f_sqrt, f_base;

    mpf_init(f_lb); mpf_init(f_sqrt); mpf_init_set_str(f_base, "2", 10);

    mpf_sqrt(f_sqrt, f_base);
    mpf_pow_ui(f_lb, f_base, n/2-1);
    mpf_mul(f_lb, f_lb, f_sqrt);
    mpz_set_f(lower_bound, f_lb);


    // Step 6: Check condition for Y > cond
    mpz_t cond;
    mpz_init(cond);
    mpz_pow_ui(cond, base, n/2);

    mpz_t Y_minus_1;
    mpz_init(Y_minus_1);
    mpz_sub_ui(Y_minus_1, Y, one);


    int i = 0;
    do {

        PRNG(X, n/2);
        while (mpz_cmp(X, lower_bound) < 0 || mpz_cmp(X, upper_bound) > 0) {
            PRNG(X, n/2);
        }

        // Step 4: Calculate Y
        mpz_mul(temp, twop1, p2);
        mpz_sub(Y, R, X);
        mpz_mod(Y, Y, temp);
        mpz_add(Y, Y, X);
```

```

        i = 0;

        mpz_gcd(gcd, Y_minus_1, e);

        while (mpz_cmp(Y, cond) < 0) {
            i += 1;
            if (mpz_cmp_ui(gcd, one) != 0) {
                if (i >= 5*(n/2)) {
                    printf("%s\n", "FAILURE");
                    exit(-1);
                }
                mpz_add(Y, Y, temp);
                mpz_gcd(gcd, Y_minus_1, e);
            }
            else {
                if (mpz_probab_prime_p(Y, 28) >= 1) {
                    mpz_set(p, Y);
                    return;
                }
                if (i >= 5*(n/2)) {
                    printf("%s\n", "FAILURE");
                    exit(-1);
                }
                mpz_add(Y, Y, temp);
                mpz_gcd(gcd, Y_minus_1, e);
            }
        }
    } while (mpz_cmp(Y, cond) >= 0);

    mpz_clear(gcd); mpz_clear(twop1); mpz_clear(R); mpz_clear(R1); mpz_clear(R2);
    mpz_clear(check1); mpz_clear(check2); mpz_clear(mpz_one);
    mpz_clear(lower_bound); mpz_clear(upper_bound); mpz_clear(base); mpz_clear(X); mpz_clear(temp); mpz_clear(Y);
    mpz_clear(cond); mpz_clear(Y_minus_1);

    mpf_clear(f_lb); mpf_clear(f_sqrt); mpf_clear(f_base);
}

void gen_primes(mpz_t p, mpz_t e, int n) {
    if (n != 1024 && n != 2048 && n != 3072) {
        fprintf(stderr, "Invalid bit length for RSA modulus. Exiting...\n");
        exit(-1);
    }
    mpz_t xp, xp1, xp2, p1, p2;
    mpz_init(xp); mpz_init(xp1); mpz_init(xp2); mpz_init(p1); mpz_init(p2);
    unsigned long int two = 2;

    PRNG(xp1, 104);
    PRNG(xp2, 104);

    while (mpz_probab_prime_p(xp1, 28) != 1) {
        mpz_add_ui(xp1, xp1, two);
    }
    while (mpz_probab_prime_p(xp2, 28) != 1) {
        mpz_add_ui(xp2, xp2, two);
    }
    //gmp_printf("%s\n%Zd\n%Zd\n", "Auxiliary primes for p: ", xp1, xp2);
```

```
    mpz_set(p1, xp1);
    mpz_set(p2, xp2);

    gen_probable_prime(p, p1, p2, e, n);
    mpz_clear(xp); mpz_clear(xp1); mpz_clear(xp2); mpz_clear(p1); mpz_clear(p2);
}

int coprime(mpz_t a, mpz_t b) {
    int coprime = 1;
    mpz_t gcd; mpz_init(gcd);
    mpz_t one; mpz_init_set_str(one, "1", 10);

    mpz_gcd(gcd, a, b);
    if (mpz_cmp(gcd, one) != 0) {
        coprime = 0;
    }
    mpz_clear(gcd); mpz_clear(one);
    return coprime;
}

int main() {
        struct RSAPublicKey pubK;
        struct RSAPrivateKey privK;
    mpz_init(pubK.modulus); mpz_init(pubK.publicExponent);
    mpz_init(privK.modulus); mpz_init(privK.privateExponent);
        mpz_t mod, e, d, m, c, p, q;
    mpz_init(mod); mpz_init(e); mpz_init(d); mpz_init(p); mpz_init(q);
        mpz_init(m); mpz_init(c);

    /*
     * Key generation
     */

    // Generate public exponent e
    gen_e(e);
    gmp_printf("%s%Zd\n\n", "Public exponent e: ", e);

    // Generate primes p and q for modulus n
    gen_primes(p, e, 1024);
    gen_primes(q, e, 1024);

    // Check if (p-1) and (q-1) are coprime with e
    unsigned long int one = 1;
    mpz_t p_minus_1, q_minus_1;
    mpz_init(p_minus_1); mpz_init(q_minus_1);
    mpz_sub_ui(p_minus_1, p, one);
    mpz_sub_ui(q_minus_1, q, one);

    assert(coprime(p_minus_1, e) == 1);
    assert(coprime(q_minus_1, e) == 1);

    gmp_printf("%s%Zd\n\n", "Prime p: ", p);
    gmp_printf("%s%Zd\n\n", "Prime q: ", q);

    mpz_mul(mod, p, q);

    gmp_printf("%s%Zd\n\n", "Modulus n: ", mod);

    // Generate private exponent d
    gen_d(d, p_minus_1, q_minus_1, e, 1024);
```

```
    gmp_printf("%s%Zd\n\n", "Private exponent d: ", d);

    mpz_set(pubK.modulus, mod);
    mpz_set(pubK.publicExponent, e);

    mpz_set(privK.modulus, mod);
    mpz_set(privK.privateExponent, d);

    gmp_printf("%Zd\n", pubK.modulus);
        gmp_printf("%Zd\n", pubK.publicExponent);

        return 0;
}
```

B Crypto Coding Practices

1. We learned how to use a cryptographically secure pseudorandom number generator and that even this has inherent disadvantages.
2. We learned how to prevent the elementary common modulus attack.
3. We learned how to prevent low private exponent attacks and that the security of our implementation can be improved further by choosing a larger public exponent.
4. We learned how to prevent low public exponent attacks and that the security of our implementation can be improved further by choosing a larger public exponent.
5. We learned how to prevent partial key exposure attacks and that our implementation is lacking in privacy provisions for the private exponent d.
6. We learned that our implementation is not immune to timing attacks and power consumption attacks and that precluding these attacks is difficult.

C Secure Coding Practices

1. We learned the importance of freeing dynamically allocated memory, especially in a cryptographic setting where unfreed memory can contain sensitive data.
2. We learned to correctly size memory allocation for an object; using GMP Library for most instances greatly reduces developer errors.
3. We learned the importance of converting `char` data types to `unsigned char` data types whenever it is being passed to a character-handling function.
4. We learned retrospectively that consistent and comphrehensive error handling would have made the development effort much easier.
5. We would have liked to create a comprehensive test suite that could ensure correctness through future iterations of our implementation. This would have been tremendously helpful during the development process.

6. It would be interesting to leverage static analysis techniques and a binary fuzzer, such as AFL, to discover any unintended behavior in our implementation.