

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 studied 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 focused on the number theory behind the algorithm, well-known attacks on the RSA cryptosystem, and secure coding practices associated with implementing cryptosystems more broadly. We implemented the RSA encryption and decryption algorithms according to cryptographic considerations for security and performance and according to the well-established specifications. This provided a better understanding of the nuances of cryptographic coding in practice.

2 Implementation

We first detail how we handle multiple precision numbers, then we detail our implementation of RSA key generation and encryption and decryption functions.

2.1 Handling Multiple Precision Numbers

Even before starting the implementation of PKCS #1 [2] itself, the first major challenge we faced was deciding how to store the numbers that would be used for encryption. Typical RSA integers are on the order of 1000 bits in size, which far exceeds the capacity of standard C data types. Thus, some custom BigInteger data type was necessary to store integers of arbitrary precision. Though less of a security concern, this was nonetheless a fundamental part of implementing the encryption scheme.

To gain experience working with arbitrary precision integers, we initially attempted to create the BigInteger library ourselves. Three primary design

decisions guided the process. First of all, to make memory usage efficient, we used dynamically-sized integers. This allowed integers to occupy only the memory they required, and freed up any they didn't. It also had the additional benefit of placing no limit on the capacity of a BigInteger. Secondly, intending to replicate the behavior of primitive C data types, we did not use in-place operations on BigIntegers. That is, the output of any BigInteger operation was a newly allocated BigInteger, and the operands were unchanged. Finally, we decided not to represent negative integers. This is sufficient for RSA, and had the advantage of simplicity.

The dynamic sizing ultimately proved to be very cumbersome to work with. For most operations, it wasn't possible to predict the number of bytes of storage that would be needed until after the result was computed. This resulted in excessive memory management (for example, reallocating memory after the operation to fit the size of the result) and significant performance overhead. It would have been better to assign a maximum size for a multi-precision integer, allocate a fixed block of that size, and let it grow or shrink as needed. Although this is a less efficient use of memory, the lack of overhead for managing memory would have cleaned up the code and increase performance significantly.

Likewise, avoiding in-place operations proved to be an inconvenience. On several occasions, it would have been more convenient to write back the result of an operation to one of the operands (for example, to use immediately afterward). But our library did not support this, so we were forced to allocate a new integer whether or not it wasn't necessary. This again resulted in unnecessary overhead due to memory management, and made the library more difficult to use.

In the end, our custom solution was quite inefficient, and fixing all its issues would have likely required a complete redesign. Thus, we decided instead to incorporate a preexisting library to handle multiple-precision integers. For this purpose, we settled on GMP (the GNU Multi-Precision library) [3].

2.2 Key Pair Generation

We follow the Digital Signature Standard (DSS) [4] issued by the National Institute of Standards and Technology (NIST) to generate key pairs.

2.2.1 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.

2.2.2 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 2.2.3) 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 .

2.2.3 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 desiderata 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 [4]. 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

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

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.

2.3 Encryption and Decryption

The PKCS #1 standard outlines two different schemes for RSA encryption: RSAES-OAEP and RSAES-PKCS1-v1_5. The former is required for new applications, and the latter is an older scheme kept around for backwards compatibility. For our project, we chose to implement the OAEP scheme. As indicated by its name, this scheme incorporates OAEP (Optimal Asymmetric Encryption Padding) which turns the otherwise deterministic RSA encryption into probabilistic encryption. This makes the scheme CPA-secure.

2.4 Data Primitives

As specified in the standard, there are two primary data types used for RSA encryption: octet strings and multiple-precision integers. Octet strings are used to represent messages (i.e. plaintext and ciphertext) and the multiple-precision integers are used to perform the basic mathematical operations of the RSA scheme (i.e. exponentiation). To convert between the two, the standard specifies two data conversion primitives: I2OSP (Integer to Octet String) and OS2IP (Octet String to Integer). To represent the multiple-precision integers, we already decided on the GMP library. However, there were a couple options for how to represent the octet strings.

1. Represent each octet as a single character: this is the most efficient way of representing octets, since each character can be any one of its possible 256 values. However, with this representation, string manipulation of octet strings became a challenge. Since '0' is a valid octet, a NULL character cannot be used to represent the end of an octet string. This would require storing the length of the octet string separately.
2. Represent each octet as two hex characters: this method is less efficient, as it requires two characters for each octet. But it allows for NULL-terminated octet strings, which is the standard way of representing strings in C. Furthermore, the GMP library has little support for base-256 octet strings, so this is the option we chose.

2.5 Cryptographic Primitives

The two cryptographic primitives are RSAEP, which is the encryption primitive, and RSADP, which is the decryption primitive. We implement these as prescribed in the specification, adapting the GMP Library.

2.6 RSAES-OAEP

RSAES-OAEP combines both of the cryptographic primitives aforementioned, and uses an encoding method based on Bellare and Rogaway's Optimal Asymmetric Encryption Scheme [5]. RSAES-OAEP is parameterized by a hash function and mask generation function that we describe further in Section 2.7. Both the RSAES-OAEP-Encryption and RSAES-OAEP-Decryption operations are implemented as prescribed in the PKCS specification.

To accomplish the OAEP padding, the RSA-OAEP scheme makes use of a Mask Generation Function, which in turn is based on a secure hash. The PKCS #1 standard recommends the use of a hashing algorithm from the SHA-2 hash family, but does not require any hash algorithm in particular. For our project, we decided to use the SHA-256 hash algorithm. Since the algorithm is not given in the standard, we used the implementation from the OpenSSL library.

The representation of octets as pairs of hex characters worked fairly well, but there were some issues it caused. In particular, the OpenSSL SHA256 hash function uses outputs a base-256 octet string, which was not compatible with our octet string representation. Consequently, we had to convert back and forth between these formats, decreasing the performance of our implementation. It would have been better to have a single common format for octet strings. The simplest way to do this would probably have been to store the length of an octet string along with the data and represent all octet strings in base-256.

2.7 Hash and Mask Generation

While there are numerous acceptable hash functions, SHA-512 and SHA-256 are recommended. Thus, erring for performance, we choose to use the OpenSSL SHA-256 implementation. The Mask Generation Function (MGF) is crucial for the security of the RSA encryption scheme as specified. The MGF takes an octet string of varying length, and then outputs a pseudorandom octet string of a desired length. This means that the output cannot be predicted, and the provable security of RSAES-OAEP relies on the MGF's randomness.

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 2.2.1. 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.

Guterman *et al.* perform an analysis of Linux's pseudorandom number generator (LRNG) and expose a number of security vulnerabilities [6]. 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 [7].

While the latter attacks are theoretical, and to our knowledge have not been successful in practice, Guterman also presents a denial of service attack that our implementation is susceptible to [6]. 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 exponentiation) by a factor of at least 10 for a 1024-bit modulus. However, Weiner shows that such a simplification is completely insecure [8]. Boneh and Durfee further improve the bounds of Weiner's attack, showing that $d < n^{0.292}$ is susceptible to attack [9]. 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 [10]. 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 [11]. 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 .

Another security concern we encountered when implementing the BigInteger library was a potential timing attack when performing modular exponentiation. The larger the encryption exponent, the longer a naïve implementation will take, and this leaks information about the length of the encryption exponent if an attacker can observe the time it takes to perform the exponentiation. For exponentiation using Montgomery multiplication, there is a modification called Montgomery’s Ladder which is used to make exponentiation run in constant time. We did not implement this modification, but it would be interesting to do so in the future.

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 [12].

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`, unsigned 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.

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

Taken as a whole, this project illuminated many of the intricacies involved in a real-world implementation of the RSA cryptosystem, and cryptosystems more broadly. Truly, what we learn as “textbook” RSA is a tremendous oversimplification to what RSA is in practice. As expected, the learning outcomes from this project were innumerable as we were confronted with both number theoretic attacks, as well as implementation attacks. In regards to secure coding practices, perhaps the most important learning outcome was realizing that the vast space of considerations makes cryptographic coding especially difficult, and mistakes devastating for the security of the cryptosystem. This project skimmed the surface of an RSA implementation, and it will be interesting future work to improve upon cryptographic coding practices, general coding practices, and performance.

References

- [1] Dan Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.
- [2] RSA Laboratories. PKCS #1 v2.2: RSA Cryptography Standard. 2012.
- [3] GNU GMP. Multiple precision arithmetic library.
- [4] PUB FIPS. 186-4. *Digital Signature Standard (DSS)*, 2013.
- [5] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption how to encrypt with rsa. In *Eurocrypt*, volume 94, pages 92–111, 1994.
- [6] 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.
- [7] Daniel Bernstein. Entropy attacks! 2014.
- [8] Michael J Wiener. Cryptanalysis of short rsa secret exponents. *IEEE Transactions on Information theory*, 36(3):553–558, 1990.

- [9] Dan Boneh and Glenn Durfee. New results on the cryptanalysis of low exponent rsa. *IEEE Transactions on Information Theory*, 46(4):1339–1349, 2000.
- [10] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997.
- [11] 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.
- [12] Robert C Seacord. *The CERT C secure coding standard*. Pearson Education, 2008.

A Code

Listing 1: Code for `rsa.h`.

```

1  /*
2   * Data Types
3   */
4  struct RSAPublicKey {
5      mpz_t modulus;
6      mpz_t publicExponent;
7  };
8
9  struct RSAPrivateKey {
10     mpz_t modulus;
11     mpz_t privateExponent;
12 };
13
14 /*
15  * Methods for Generating Key Pairs
16  */
17
18 void    gen_e            (mpz_t e);
19 void    gen_d            (mpz_t d, mpz_t p_minus_1, mpz_t
20     q_minus_1, mpz_t e, int n);
21 void    gen_probable_prime (mpz_t p, mpz_t p1, mpz_t p2, mpz_t e,
22     int n);
23 void    gen_primes       (mpz_t p, mpz_t e, int n);
24 int     coprime           (mpz_t a, mpz_t b);
25 void    PRNG              (mpz_t rand, int n);
26
27 /*
28  * Methods for Encryption and Decryption
29  */
30 char*   I2OSP              (mpz_t x, int xLen);
31 void    OS2IP              (char *X, mpz_t x);
32 int     RSAEP              (struct
33     RSAPublicKey *K, mpz_t m, mpz_t c);
34 int     RSADP              (struct
35     RSAPrivateKey *K, mpz_t c, mpz_t m);
36 char*   MGF1              (char *mgfSeed, unsigned
37     long long maskLen);
38 char*   RSAES_OAEP_ENCRYPT (struct RSAPublicKey *K, char *M,
39     char *L);
40 char*   RSAES_OAEP_DECRYPT (struct RSAPrivateKey *K, char *C,
41     char *L);

```

Listing 2: Code for `rsa.c`.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdarg.h>
4  #include <string.h>
5  #include <time.h>
6  #include <gmp.h>
7  #include <openssl/sha.h>
8  #include "rsa.h"
9  #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <fcntl.h>
12 #include <math.h>
13 #include <assert.h>
14
15 // Convert nonnegative integer x to a zero-padded octet string of
    length xLen.
16 char* I2OSP(mpz_t x, int xLen) {
17     size_t osLen = mpz_sizeinbase(x, 16);
18     xLen *= 2;
19     if (xLen < osLen) {
20         printf("integer too large\n");
21         return NULL;
22     }
23     char *os = malloc((xLen + 1) * sizeof(char));
24     memset(os, '0', xLen - osLen);
25     mpz_get_str(os + xLen - osLen, 16, x);
26     os[xLen] = '\0';
27     return os;
28 }
29
30 // Convert octet string to a nonnegative integer
31 void OS2IP(char *X, mpz_t x) {
32     mpz_set_str(x, X, 16);
33 }
34
35 // RSA Encryption Primitive
36 int RSAEP(struct RSAPublicKey *K, mpz_t m, mpz_t c) {
37     if (mpz_cmp(m, K->modulus) >= 0) {
38         printf("message representative out of range\n");
39         return 0;
40     }
41     mpz_powm_sec(c, m, K->publicExponent, K->modulus);
42     return 1;
43 }
44
45 // RSA Decryption Primitive
46 int RSADP(struct RSAPrivateKey *K, mpz_t c, mpz_t m) {
47     if (mpz_cmp(c, K->modulus) >= 0) {
48         printf("ciphertext representative out of range\n");
49         return 0;
50     }
51     mpz_powm_sec(m, c, K->privateExponent, K->modulus);
52     return 1;
53 }
54
55 // Mask generation function specified in PKCS #1 Appendix B.
56 char* MGF1(char *mgfSeed, unsigned long long maskLen) {
57     // Step 1: Verify maskLen <= (hLen * 2^32)
58     unsigned long long hLen = SHA256_DIGEST_LENGTH;
59     if (maskLen > (hLen << 32)) {
60

```

```

61         printf("mask too long\n");
62         return NULL;
63     }
64     maskLen *= 2;
65     hLen *= 2;
66
67     // Step 2: Init T to empty octet string. T consists of TLen
        SHA256 hashes.
68     int TLen = (maskLen + hLen - 1) / hLen;
69     char *T = malloc((TLen * hLen) * sizeof(char));
70
71     char *TPtr = T;
72     char *hashOp;
73     size_t mgfSeedLen = strlen(mgfSeed);
74     hashOp = malloc((mgfSeedLen + 4 * 2) * sizeof(char));
75     memcpy(hashOp, mgfSeed, mgfSeedLen);
76
77     // Step 3: Generate mask
78     int i, j;
79     char *C;
80     unsigned char *hash;
81     unsigned char hChar;
82     hash = malloc(SHA256_DIGEST_LENGTH * sizeof(char));
83     mpz_t counter;
84     mpz_init(counter);
85     for (i = 0; i < TLen; ++i) {
86         mpz_set_ui(counter, i);
87         C = I2OSP(counter, 4);
88         memcpy(hashOp + mgfSeedLen, C, 4 * 2);
89         SHA256(hashOp, mgfSeedLen + 4 * 2, hash);
90         for (j = 0; j < hLen; j += 2)
91             sprintf(TPtr + j, "%02x", hash[j/2]);
92         TPtr += hLen;
93         free(C);
94     }
95
96     // Step 4: Output mask
97     char *mask = malloc(maskLen + 1);
98     memcpy(mask, T, maskLen);
99     mask[maskLen] = '\0';
100    free(hash); free(hashOp); free(T);
101    return mask;
102 }
103
104 // RSA Encryption with OAEP. Section 7.1.1 in PKCS #1
105 char* RSAES_OAEP_ENCRYPT(struct RSAPublicKey *K, char* M, char *L)
106 {
107     // Step 1: Length checking (*_o stores size in octets; *_h
        in hex chars)
108     size_t k_o = (mpz_sizeinbase(K->modulus, 16) + 1) / 2;
109     size_t hLen_o = SHA256_DIGEST_LENGTH;
110     size_t mLen_o = strlen(M) / 2;
111     size_t maxmLen_o = k_o - 2 * hLen_o - 2;
112     if (mLen_o > maxmLen_o) {
113         printf("message too long\n");
114         return NULL;
115     }
116     size_t k_h = k_o * 2;
117     size_t hLen_h = hLen_o * 2;
118     size_t mLen_h = mLen_o * 2;    // If M is valid, then
        mLen_h = strlen(M)

```

```

119
120 // Step 2: EME-OAEP encoding
121 if (L == NULL) L = "";
122 char *lHash = SHA256(L, strlen(L), NULL);
123
124 // b. Generate random padding string (PS)
125 size_t PSlen_h = (maxLen_o - mLen_o) * 2;
126 char *PS = malloc(PSlen_h * sizeof(char));
127 memset(PS, '0', PSlen_h);
128
129 // c. Generate data block (DB)
130 size_t DBlen_o = k_o - hLen_o - 1;
131 size_t DBlen_h = DBlen_o * 2;
132 char *DB = malloc((DBlen_h + 1) * sizeof(char));
133 int i;
134 for (i = 0; i < hLen_o; ++i)
135     sprintf(DB + 2 * i, "%02x", (unsigned char)lHash[i]);
136 memcpy(DB + hLen_h, PS, PSlen_h);
137 memcpy(DB + hLen_h + PSlen_h, "01", 2);
138 memcpy(DB + DBlen_h - mLen_h, M, mLen_h);
139 DB[DBlen_h] = '\0';
140
141 // d. Generate random seed
142 mpz_t seed;
143 mpz_init(seed);
144 PRNG(seed, hLen_o * 8);
145 char *seedStr = I2OSP(seed, hLen_o);
146
147 // ef. Generate dbMask and compute DB XOR dbMask
148 char *dbMask = MGF1(seedStr, DBlen_o);
149 mpz_t op1, op2, rop;
150 mpz_init_set_str(op1, DB, 16);
151 mpz_init_set_str(op2, dbMask, 16);
152 mpz_init(rop);
153 mpz_xor(rop, op1, op2);
154 char *maskedDB = I2OSP(rop, DBlen_o);
155
156 // gh. Generate seedMask and compute seed XOR seedMask
157 char *seedMask = MGF1(maskedDB, hLen_o);
158 mpz_set_str(op1, seedStr, 16);
159 mpz_set_str(op2, seedMask, 16);
160 mpz_xor(rop, op1, op2);
161 char *maskedSeed = I2OSP(rop, hLen_o);
162
163 // i. Generate encoded message (EM)
164 size_t EMLen_h = hLen_h + DBlen_h + 2;
165 char *EM = malloc((EMLen_h + 1) * sizeof(char));
166 memset(EM, '0', 2);
167 memcpy(EM + 2, maskedSeed, hLen_h);
168 memcpy(EM + hLen_h + 2, maskedDB, DBlen_h);
169 EM[EMLen_h] = '\0';
170
171 // Step 3-4: RSA encryption
172 mpz_t m, c;
173 mpz_init(m);
174 mpz_init(c);
175 OS2IP(EM, m);
176 RSAEP(K, m, c);
177 char *C = I2OSP(c, k_o);
178
179 // Free memory

```

```

180     free(PS); free(DB); free(dbMask); free(maskedDB);
181     free(seedMask); free(maskedSeed); free(EM);
182     mpz_clear(op1); mpz_clear(op2); mpz_clear(rop);
183     mpz_clear(m); mpz_clear(c);
184
185     return C;
186 }
187
188 // RSA Decryption with OAEP. Section 7.1.2 in PKCS #1
189 char *RSAES_OAEP_DECRYPT(struct RSAPrivateKey *K, char* C, char *L)
190 {
191     // Step 1: Length checking (*_o stores sizes in octets; *_h
192     // in hex chars)
193     size_t k_o = (mpz_sizeinbase(K->modulus, 16) + 1) / 2;
194     size_t CLen_o = strlen(C) / 2;
195     if (k_o != CLen_o) {
196         printf("decryption_error\n");
197         return NULL;
198     }
199     size_t hLen_o = SHA256_DIGEST_LENGTH;
200     if (k_o < (2 * hLen_o + 2)) {
201         printf("decryption_error\n");
202         return NULL;
203     }
204     // Step 2: RSA Decryption
205     mpz_t c, m;
206     mpz_init(c);
207     mpz_init(m);
208     OS2IP(C, c);
209     if (!RSADP(K, c, m)) {
210         printf("decryption_error\n");
211         return NULL;
212     }
213     char *EM = I2OSP(m, k_o);
214
215     // Step 3: EME-OAEP decoding
216     if (L == NULL) L = "";
217     size_t hLen_h = hLen_o * 2;
218     char *lHash_o = malloc(hLen_o * sizeof(char));
219     char *lHash_h = malloc(hLen_h * sizeof(char));
220     SHA256(L, strlen(L), lHash_o);
221     int i;
222     for (i = 0; i < hLen_o; ++i)
223         sprintf(lHash_h + 2 * i, "%02x", (unsigned char)
224             lHash_o[i]);
225
226     // b. Separate encoded message (EM) into its component
227     // parts
228     size_t DBLen_o = k_o - hLen_o - 1;
229     size_t DBLen_h = DBLen_o * 2;
230     char *maskedSeed = malloc((hLen_h + 1) * sizeof(char));
231     char *maskedDB = malloc((DBLen_h + 1) * sizeof(char));
232     memcpy(maskedSeed, EM + 2, hLen_h);
233     memcpy(maskedDB, EM + 2 + hLen_h, DBLen_h);
234     maskedSeed[hLen_h] = '\0';
235     maskedDB[DBLen_h] = '\0';
236
237     // cd. Generate seedMask and compute maskedSeed XOR
238     // seedMask
239     char *seedMask = MGF1(maskedDB, hLen_o);

```

```

237     mpz_t op1, op2, rop;
238     mpz_init_set_str(op1, maskedSeed, 16);
239     mpz_init_set_str(op2, seedMask, 16);
240     mpz_init(rop);
241     mpz_xor(rop, op1, op2);
242     char *seed = I2OSP(rop, hLen_o);
243
244     // ef. Generate dbMask and compute maskedDB XOR dbMask
245     char *dbMask = MGF1(seed, DBLen_o);
246     mpz_set_str(op1, maskedDB, 16);
247     mpz_set_str(op2, dbMask, 16);
248     mpz_xor(rop, op1, op2);
249     char *DB = I2OSP(rop, DBLen_o);
250
251     // g. Separate data block (DB) into component parts to
252         recover message
253     size_t PSLen_h = strstr(DB + hLen_h, "01") - DB - hLen_h;
254     int mLen_h = DBLen_h - PSLen_h - hLen_h - 1;
255     int errCount = 0;
256     errCount += (mLen_h < 0);
257     errCount += !(EM[0] == '0' && EM[1] == '0');
258     errCount += (strncmp(DB, lHash_h, hLen_h) != 0);
259     if (errCount > 0) {
260         printf("decryption error\n");
261         return NULL;
262     }
263     char *M = malloc((mLen_h + 1) * sizeof(char));
264     memcpy(M, DB + DBLen_h - mLen_h + 1, mLen_h);
265     M[mLen_h] = '\0';
266
267     // Free memory
268     free(EM); free(lHash_o); free(lHash_h); free(maskedSeed);
269     free(maskedDB);
270     free(seedMask); free(seed); free(dbMask); free(DB);
271     mpz_clear(op1); mpz_clear(op2); mpz_clear(rop);
272     mpz_clear(m); mpz_clear(c);
273     return M;
274 }
275
276 // Generates pseudorandom n bits from /dev/random file
277 void PRNG(mpz_t rand, int n) {
278     int devrandom = open("/dev/random", O_RDONLY);
279     char randbits[n/8];
280     size_t randlen = 0;
281     while (randlen < sizeof randbits) {
282         ssize_t result = read(devrandom, randbits + randlen, (
283             sizeof randbits) - randlen);
284         if (result < 0)
285             printf("%s\n", "Could not read from /dev/random");
286         randlen += result;
287     }
288     close(devrandom);
289     mpz_import(rand, sizeof(randbits), 1, sizeof(randbits[0]), 0,
290         0, randbits);
291     // Make sure rand is odd
292     if (mpz_odd_p(rand) == 0) {
293         unsigned long int one = 1;
294         mpz_add_ui(rand, rand, one);
295     }

```



```

295 }
296
297 // Generate (constant) public exponent e
298 void gen_e(mpz_t e) {
299     // Set e to 2^16 + 1
300     unsigned long int e_int = pow(2,16)+1;
301     mpz_set_ui(e, e_int);
302 }
303
304 // Generate private exponent d
305 void gen_d(mpz_t d, mpz_t p_minus_1, mpz_t q_minus_1, mpz_t e, int
    n) {
306
307     unsigned long int one = 1;
308     mpz_t lower_bound, upper_bound, base;
309     mpz_init(lower_bound); mpz_init(upper_bound); mpz_init_set_str(
        base, "2", 10);
310     mpz_pow_ui(lower_bound, base, n/2);
311     mpz_lcm(upper_bound, p_minus_1, q_minus_1);
312
313     mpz_invert(d, e, upper_bound);
314     if (mpz_cmp(d, lower_bound) < 0 || mpz_cmp(d, upper_bound) > 0)
    {
315         fprintf(stderr, "Private exponent d too small, try again\n"
        );
316         exit(-1);
317     }
318
319     mpz_t ed, check_d;
320     mpz_init(ed); mpz_init(check_d);
321
322     mpz_mul(ed, e, d);
323     mpz_mod(check_d, ed, upper_bound);
324
325     assert(mpz_cmp_ui(check_d, one) == 0);
326
327 }
328
329 // Generate probable prime from auxiliary primes
330 void gen_probable_prime(mpz_t p, mpz_t p1, mpz_t p2, mpz_t e, int n
    ) {
331
332     // Step 1: Check if p1 and p2 are coprime
333     mpz_t gcd, twop1;
334     mpz_init(gcd); mpz_init(twop1);
335     unsigned long int one = 1;
336     unsigned long int two = 2;
337     mpz_mul_ui(twop1, p1, two);
338     mpz_gcd(gcd, twop1, p2);
339     if (mpz_cmp_ui(gcd, one) != 0) {
340         fprintf(stderr, "Auxiliaries p1 and p2 not coprime\n");
341         exit(-1);
342     }
343
344     // Step 2: Chinese remainder theorem
345     mpz_t R; mpz_t R1; mpz_t R2;
346     mpz_init(R); mpz_init(R1); mpz_init(R2);
347
348     mpz_invert(R1, p2, twop1);
349     mpz_mul(R1, R1, p2);
350
351     mpz_invert(R2, twop1, p2);

```

```

352     mpz_mul(R2, R2, twop1);
353
354     mpz_sub(R, R1, R2);
355
356     // Check for CRT
357     mpz_t check1; mpz_t check2; mpz_t mpz_one;
358     mpz_init(check1); mpz_init(check2); mpz_init(mpz_one);
359     mpz_set_str(mpz_one, "1", 10);
360     mpz_mod(check1, R, twop1);
361     mpz_mod(check2, R, p2);
362     mpz_sub(check2, p2, check2);
363     assert(mpz_cmp(check1, mpz_one) == 0);
364     assert(mpz_cmp(check2, mpz_one) == 0);
365
366
367     // Step 3: Generate random X between lower_bound and
368             upper_bound
369     mpz_t lower_bound; mpz_t upper_bound; mpz_t base; mpz_t X;
370             mpz_t temp; mpz_t Y;
371     mpz_init(lower_bound); mpz_init(upper_bound); mpz_init(base);
372             mpz_init(X); mpz_init(temp); mpz_init(Y);
373
374     mpz_set_str(base, "2", 10);
375     mpz_pow_ui(upper_bound, base, n/2);
376     mpz_sub_ui(upper_bound, upper_bound, one);
377
378     mpf_t f_lb, f_sqrt, f_base;
379
380     mpf_init(f_lb); mpf_init(f_sqrt); mpf_init_set_str(f_base, "2",
381             10);
382
383     mpf_sqrt(f_sqrt, f_base);
384     mpf_pow_ui(f_lb, f_base, n/2-1);
385     mpf_mul(f_lb, f_lb, f_sqrt);
386     mpz_set_f(lower_bound, f_lb);
387
388     mpz_t cond;
389     mpz_init(cond);
390     mpz_pow_ui(cond, base, n/2);
391
392     mpz_t Y_minus_1;
393     mpz_init(Y_minus_1);
394     mpz_sub_ui(Y_minus_1, Y, one);
395
396     int i = 0;
397     do {
398         PRNG(X, n/2);
399         while (mpz_cmp(X, lower_bound) < 0 || mpz_cmp(X,
400             upper_bound) > 0) {
401             PRNG(X, n/2);
402         }
403
404         // Step 4: Calculate Y
405         mpz_mul(temp, twop1, p2);
406         mpz_sub(Y, R, X);
407         mpz_mod(Y, Y, temp);
408         mpz_add(Y, Y, X);

```

```

409         // Step 5: i = 0
410         i = 0;
411
412         mpz_gcd(gcd, Y_minus_1, e);
413
414         // Step 11: Go to Step 6
415         while (mpz_cmp(Y, cond) < 0) {
416             i += 1;
417             if (mpz_cmp_ui(gcd, one) != 0) {
418                 if (i >= 5*(n/2)) {
419                     printf("%s\n", "FAILURE");
420                     exit(-1);
421                 }
422                 mpz_add(Y, Y, temp);
423                 mpz_gcd(gcd, Y_minus_1, e);
424             }
425             // Step 7: If GCD(Y-1, e) = 1
426             else {
427                 if (mpz_probab_prime_p(Y, 28) >= 1) {
428                     mpz_set(p, Y);
429                     return;
430                 }
431
432                 //Step 8: Check if failure
433                 if (i >= 5*(n/2)) {
434                     printf("%s\n", "FAILURE");
435                     exit(-1);
436                 }
437
438                 //Step 10: Update Y
439                 mpz_add(Y, Y, temp);
440                 mpz_gcd(gcd, Y_minus_1, e);
441             }
442         }
443         // Step 6: Check condition for Y > cond
444     } while (mpz_cmp(Y, cond) >= 0);
445
446     mpz_clear(gcd); mpz_clear(twop1); mpz_clear(R); mpz_clear(R1);
447     mpz_clear(R2);
448     mpz_clear(check1); mpz_clear(check2); mpz_clear(mpz_one);
449     mpz_clear(lower_bound); mpz_clear(upper_bound); mpz_clear(base);
450     ; mpz_clear(X); mpz_clear(temp); mpz_clear(Y);
451     mpz_clear(cond); mpz_clear(Y_minus_1);
452
453     mpf_clear(f_lb); mpf_clear(f_sqrt); mpf_clear(f_base);
454 }
455
456 // Generate auxiliary primes
457 void gen_auxiliary_primes(mpz_t p, mpz_t e, int n) {
458     if (n != 1024 && n != 2048 && n != 3072) {
459         fprintf(stderr, "Invalid bit length for RSA modulus.\n");
460         Exiting...\n");
461         exit(-1);
462     }
463     mpz_t xp, xp1, xp2, p1, p2;
464     mpz_init(xp); mpz_init(xp1); mpz_init(xp2); mpz_init(p1);
465     mpz_init(p2);
466     unsigned long int two = 2;
467
468     int len_aux = 0;
469     int mr_rounds = 0;

```

```

467     if (n == 1024) {
468         len_aux = 104;
469         mr_rounds = 28;
470     }
471     else if (n == 2048) {
472         len_aux = 144;
473         mr_rounds = 38;
474     }
475     else if (n == 3072) {
476         len_aux = 176;
477         mr_rounds = 41;
478     }
479
480     PRNG(xp1, len_aux);
481     PRNG(xp2, len_aux);
482
483     while (mpz_probab_prime_p(xp1, mr_rounds) != 1) {
484         mpz_add_ui(xp1, xp1, two);
485     }
486     while (mpz_probab_prime_p(xp2, mr_rounds) != 1) {
487         mpz_add_ui(xp2, xp2, two);
488     }
489     //gmp_printf("%s\n%Zd\n%Zd\n", "Auxiliary primes for p: ", xp1,
490         xp2);
491     mpz_set(p1, xp1);
492     mpz_set(p2, xp2);
493
494     gen_probable_prime(p, p1, p2, e, n);
495     mpz_clear(xp); mpz_clear(xp1); mpz_clear(xp2); mpz_clear(p1);
496     mpz_clear(p2);
497 }
498
499 // Check if gcd(a,b) = 1 (coprime)
500 int coprime(mpz_t a, mpz_t b) {
501     int coprime = 1;
502     mpz_t gcd; mpz_init(gcd);
503     mpz_t one; mpz_init_set_str(one, "1", 10);
504
505     mpz_gcd(gcd, a, b);
506     if (mpz_cmp(gcd, one) != 0) {
507         coprime = 0;
508     }
509     mpz_clear(gcd); mpz_clear(one);
510     return coprime;
511 }
512
513 /*
514 int main() {
515     struct RSAPublicKey pubK;
516     struct RSAPrivateKey privK;
517     mpz_init(pubK.modulus); mpz_init(pubK.publicExponent);
518     mpz_init(privK.modulus); mpz_init(privK.privateExponent);
519     mpz_t mod, e, d, p, q;
520     mpz_init(mod); mpz_init(e); mpz_init(d); mpz_init(p); mpz_init(
521         q);
522
523     //
524     * Key generation
525     *
526
527     // Generate public exponent e
528     gen_e(e);
529     gmp_printf("%s%Zd\n\n", "Public exponent e: ", e);

```

```

526
527 // Generate primes p and q for modulus n
528 gen_auxiliary_primes(p, e, 1024);
529 gen_auxiliary_primes(q, e, 1024);
530
531 // Check if (p-1) and (q-1) are coprime with e
532 unsigned long int one = 1;
533 mpz_t p_minus_1, q_minus_1;
534 mpz_init(p_minus_1); mpz_init(q_minus_1);
535 mpz_sub_ui(p_minus_1, p, one);
536 mpz_sub_ui(q_minus_1, q, one);
537
538 assert(coprime(p_minus_1, e) == 1);
539 assert(coprime(q_minus_1, e) == 1);
540
541 gmp_printf("%sZd\n\n", "Prime p: ", p);
542 gmp_printf("%sZd\n\n", "Prime q: ", q);
543
544 mpz_mul(mod, p, q);
545
546 gmp_printf("%sZd\n\n", "Modulus n: ", mod);
547
548 // Generate private exponent d
549 gen_d(d, p_minus_1, q_minus_1, e, 1024);
550
551 gmp_printf("%sZd\n\n", "Private exponent d: ", d);
552
553 mpz_set(pubK.modulus, mod);
554 mpz_set(pubK.publicExponent, e);
555
556 mpz_set(privK.modulus, mod);
557 mpz_set(privK.privateExponent, d);
558
559 gmp_printf("%Zd\n", pubK.modulus);
560 gmp_printf("%Zd\n", pubK.publicExponent);
561 return 0;
562 }*/

```

Listing 3: Code for test.c.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <gmp.h>
5 #include "rsa.h"
6
7 int Test1(struct RSAPublicKey*, struct RSAPrivateKey*);
8
9 int main() {
10     struct RSAPublicKey pubK;
11     struct RSAPrivateKey privK;
12     mpz_init(pubK.modulus);
13     mpz_init(pubK.publicExponent);
14     mpz_init(privK.modulus);
15     mpz_init(privK.privateExponent);
16
17     if (Test1(&pubK, &privK))
18         printf("Test1:␣Passed!\n");
19     else
20         printf("Test2:␣Failed!\n");
21
22     return 0;
23 }

```

```

24
25 int Test1(struct RSAPublicKey *pubK, struct RSAPrivateKey *privK) {
26     char *message = "6628194
        e12073db03ba94cda9ef9532397d50dba79b987004afefe34";
27     char *mStr = "a8b3b284af8eb50b387034a860f146
        c4919f318763cd6c5598c8ae4811a1e0ab4c
        c7e0b082d693a5e7fc6ed675cf4668512772c0
        cbc64a742c6c630f533c8cc72f62ae833c40b
        f25842e984bb78bd6bf97c0107d55bdb662f5
        c4e0fab9845cb5148ef7392dd3aa93ae1e6
        b667b3bd4247616d4f5ba10d4cfcd226de88d3
        9f16fb";
28     char *eStr = "010001";
29     char *dStr = "53339cfd7b9f8c846a655c7316ac8
        5c55fd8f6d898fdaf119517ef4f52e8fd8e
        258df93fdee180fa0e4ab29693cd83b152a553
        d4ac4d1812b8b9fa5af0e7f55fe7304df4157
        0926f3311f15c4d65a732c483116ee3d3d2d
        0af3549ad9bfb7c6fb78ad884f84d5b5eb04724
        dc7369b31def37d0cf539e9cfcd3de653729
        ea5d5d1";
30     mpz_set_str(pubK->modulus, mStr, 16);
31     mpz_set(privK->modulus, pubK->modulus);
32     mpz_set_str(pubK->publicExponent, eStr, 16);
33     mpz_set_str(privK->privateExponent, dStr, 16);
34
35     char *C = RSAES_OAEP_ENCRYPT(pubK, message, NULL);
36     char *M = RSAES_OAEP_DECRYPT(privK, C, NULL);
37     if (!M) return -1;
38
39     return (strcmp(M, message) == 0);
40 }

```

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 comprehensive 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.