3.1: Deriving the Private Key

The RSA algorithm falls under the category of public key algorithms that are based on the integer factorization problem. Although multiplying two large prime numbers is easy, factoring their product is difficult. The security of the RSA crypto scheme is closely related to the size of its operands and the principles of modular arithmetic. Due to its computational complexity, it is primarily used for encrypting small pieces of data or generating digital signatures.

There are two keys used in public-key cryptography: a public key for encrypting plaintext, and a private key for decrypting an encrypted message. Generally speaking, generating a private key `d` for RSA involves five steps. First, two large prime numbers `p` and `q` are chosen. The product `n = p × q` is referred to as the modulus. Next, Euler's totient function is calculated for `n`, representing the number of relatively prime integers in $\mathbb{Z}_n$. A public key `e` is chosen such that it is less than and relatively prime with $\Phi(n)$. Finally, the private key `d`, the modular multiplicative inverse of `e`, is computed using the formula $d \cdot e \equiv 1 \mod \Phi(n)$.

From page 176 of Understanding Cryptography (2010) by Christof Paar et al. :

RSA Key Generation
Output: public key: $k_{\text{pub}} = (n, e)$ and private key: $k_{\text{pr}} = (d)$

1. Choose two large primes p and q.
2. Compute $n = p \cdot q$.
3. Compute $\Phi(n) = (p-1)(q-1)$.
4. Select the public exponent $e \in \{1, 2, ..., \Phi(n) - 1\}$ such that $\gcd(e, \Phi(n)) = 1$.
5. Compute the private key d such that $d \cdot e \equiv 1 \pmod{\Phi(n)}$.

Given the values of `p` and `q`, the first step of the RSA key generation is already complete. However, storing and performing arithmetic operations on these numbers is not easy and requires a Big Number library like the BIGNUM API provided by OpenSSL. The `BN_mul` function can be used to multiply two BIGNUM objects `p` and `q` to obtain `n`, followed by `BN_num_bits`, which can be used to determine the number of significant bits a BIGNUM contains. Then `BN_sub_word` can be used to decrement `p` and `q`, followed by `BN_mul` to calculate $\Phi(n)$. The final step requires the function `BN_mod_inverse` to perform the modular exponentiation and derive the private key.

private_key_gen.c

```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(BIGNUM *a)
{
    char *number_str = BN_bn2hex(a); // binary to hex
    printf("0x%s", number_str);
    OPENSSL_free(number_str);
}

int main(int argc, char *argv[])
{
    /*
        Expected input: ./program p q e mode
```

```
    p and q are prime numbers and e is the public exponent in hex

    mode 1: print d
    mode 2: print n
    mode 3: print d, n and the number of bits in n
    mode 4: print d and n

    Example usage:
    cd a2
    gcc -o priv_key private_key_gen.c -lssl -lcrypto -ldl -I/usr/↩
       include/python3.10
    ./priv_key 0x879a5ee58ade33942040f  0x3bef5e448f18ae4ff08c65 0↩
       x10001 1
    ./priv_key 0x879a5ee58ade33942040f  0x3bef5e448f18ae4ff08c65 0↩
       x10001 2
    ./priv_key 0x879a5ee58ade33942040f  0x3bef5e448f18ae4ff08c65 0↩
       x10001 3
    ./priv_key 0x879a5ee58ade33942040f  0x3bef5e448f18ae4ff08c65 0↩
       x10001 4
*/
if (argc != 5) {
    fprintf(stderr, "Usage: %s <p> <q> <e> mode\n", argv[0]);
    return 1;
}

// Variables
BN_CTX *ctx = BN_CTX_new();
BIGNUM *p = BN_new(); // primes
BIGNUM *q = BN_new();
BIGNUM *e = BN_new(); // public and private exponents
BIGNUM *d = BN_new();
BIGNUM *n = BN_new(); // n and phi(n)
BIGNUM *phi_n = BN_new();

BN_hex2bn(&p, argv[1] + 2); // p = argv[1], ignore the '0x' prefix
BN_hex2bn(&q, argv[2] + 2); // q = argv[2]
BN_hex2bn(&e, argv[3] + 2); // e = argv[3]
int mode = atoi(argv[4]);

BN_mul(n, p, q, ctx); // n = p * q

BN_sub(p, p, BN_value_one()); // p = p - 1
BN_sub(q, q, BN_value_one()); // q = q - 1
BN_mul(phi_n, p , q, ctx); // phi(n) = (p - 1) * (q - 1)
BN_mod_inverse(d, e, phi_n, ctx); // d = e^-1 mod phi(n)

if (mode == 1){
    printBN(d);
    printf("\n");
}
else if (mode == 2) {
    printBN(n);
    printf("\n");
}
else if (mode == 3) {
    printf("d: ");
    printBN(d);
    printf("\nn: ");
```

```
        printBN(n);
        printf("\nbits in n : %d bits\n", BN_num_bits(n));
    }
    else {
        printBN(d);
        printf(" ");
        printBN(n);
        printf("\n");
    }

    // Clean up
    BN_CTX_free(ctx);
    BN_free(p);
    BN_free(q);
    BN_free(n);
    BN_free(e);
    BN_free(d);
    BN_free(phi_n);
    return 0;
}
```

The expected output is:

```
./priv_key 0x879a5ee58ade33942040f  0x3bef5e448f18ae4ff08c65 0x10001 3
d: 0x01A87C31CA14E9E34D1CD5B8816A148E3ACD85243B09
n: 0x01FBF5EEC7EF3A71C2754B2E0EE10767154C2053CDEB
bits in n : 169 bits
```

Therefore
(1) The bit length of the modulus n is 169 bits
(2) The private key d is `0x01A87C31CA14E9E34D1CD5B8816A148E3ACD85243B09`

3.2: Encrypting a Message

Before we encrypt the message, we must convert the ASCII string to hexadecimal, which can be done using the Python/C API and the binascii library. Then we convert the hexadecimal into a BIGNUM object using `BN_hex2bn()`. Since the RSA encryption function below is modular exponentiation, we can use the `BN_mod_exp()` function to compute $y = x^e \mod n$. It's also important to confirm that the value of x is less than or equal to the modulus $n - 1$ or else the encryption is not effective.

From page 174 of the textbook:

> **RSA Encryption:** Given the public key $(n, e) = k_{\text{pub}}$ and the plaintext x, the encryption function is:
>
> $$y = e_{k_{\text{pub}}}(x) \equiv x^e \mod n \quad (7.1)$$
>
> where $x, y \in \mathbb{Z}_n$.

The python script:

hex_ascii.py

```
import binascii
import sys

def ascii_to_hex(hex_str):
```

```
    try:
        return binascii.hexlify(hex_str.encode("utf-8")).decode("ascii")
    except (UnicodeDecodeError, binascii.Error):
        return "The message could not be converted to hexadecimal."

def hex_to_ascii(hex_str):
    try:
        return binascii.unhexlify(hex_str).decode("ascii")
    except (UnicodeDecodeError, binascii.Error):
        return "The bytes do not represent valid ASCII characters."
```

Header and code files that use the Python/C API:

hexify_dehexify.h

```
#ifndef HEXIFY_DEHEXIFY_H
#define HEXIFY_DEHEXIFY_H
#include <openssl/bn.h>

// Function to convert a hexadecimal string to ASCII
int dehexify(const char *script, const char *function, const char *↩
    hex_str, char **ascii_str);

// Function to convert an ASCII string to hexadecimal
int hexify(const char *script, const char *function, const char *↩
    ascii_str, BIGNUM *m);

#endif // HEXIFY_DEHEXIFY_H
```

hexify_dehexify.c

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "hexify_dehexify.h"
#include <stdio.h>

/*
    This code was adapted from the Python C API documentation
    https://docs.python.org/3/extending/embedding.html
    Section 1.3
    Changes made to reflect known argument number and types
    as well as conditional error handling and decref calls
*/
int dehexify(const char *script, const char *function, const char *↩
    hex_str, char **ascii_out) {
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;

    pName = PyUnicode_DecodeFSDefault(script);

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, function);

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(1);
            pValue = PyUnicode_FromString(hex_str);
```

```
            if (!pValue) {
                Py_DECREF(pArgs);
                Py_DECREF(pModule);
                fprintf(stderr, "Cannot convert hex string\n");
                return 1;
            }
            PyTuple_SetItem(pArgs, 0, pValue);

            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);

            if (pValue != NULL) {
                if (PyUnicode_Check(pValue)) {
                    const char *result = PyUnicode_AsUTF8(pValue);
                    *ascii_out = strdup(result); // Allocate and copy the↩
                        result string
                    if (!*ascii_out) { // strdup failed
                        fprintf(stderr, "Failed to allocate memory for ↩
                            ascii_out\n");
                        Py_DECREF(pValue);
                        return 1;
                    }
                }
                Py_DECREF(pValue);
            } else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr, "Call failed\n");
                return 1;
            }
            Py_DECREF(pFunc);
        } else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", function);
        }
        Py_DECREF(pModule);
    } else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", script);
        return 1;
    }
    return 0;
}

/*
    This code was adapted from the Python C API documentation
    https://docs.python.org/3/extending/embedding.html
    Section 1.3
    Changes made to reflect known argument number and types
    as well as conditional error handling and decref calls
*/
int hexify(const char *script, const char *function, const char *↩
    ascii_str, BIGNUM *m) {
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
```

```
    pName = PyUnicode_DecodeFSDefault(script);

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, function);

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(1);
            pValue = PyUnicode_FromString(ascii_str);

            if (!pValue) {
                Py_DECREF(pArgs);
                Py_DECREF(pModule);
                fprintf(stderr, "Cannot convert message to bytes\n");
                return 1;
            }
            PyTuple_SetItem(pArgs, 0, pValue);

            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);

            if (pValue != NULL) {
                const char *hex_str = PyUnicode_AsUTF8(pValue);
                BN_hex2bn(&m, hex_str);
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr, "Call failed\n");
                return 1;
            }
            Py_DECREF(pFunc);
        } else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", function);
        }
        Py_DECREF(pModule);
    } else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", script);
        return 1;
    }
    return 0;
}
```

The code which encrypts a given message:

encrypt_message.c

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "hexify_dehexify.h"
#include <stdio.h>
```

```c
#include <openssl/bn.h>
/*
    3.2
*/

void printBN(char *msg, BIGNUM *a)
{
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

int main(int argc, char *argv[])
{
    Py_Initialize();

    // Get the current working directory
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        // Convert current directory to Python object
        PyObject *sysPath = PySys_GetObject("path");
        PyObject *path = PyUnicode_FromString(cwd);
        // Append current directory to sys.path
        PyList_Append(sysPath, path);
        Py_DECREF(path);
    } else {
        perror("getcwd() error");
        return 1;
    }

    /*
        Expected input: ./program x n e
        x is the message in ASCII format
        p and q are the primes used to generate n in hex
        e is the public exponent in hex
        Example usage:
        cd path/to/current/directory
        gcc -o encrypt encrypt_message.c hexify_dehexify.c $(python3-↩
            config --cflags) -L/usr/lib -lpython3.10 $(python3-config --↩
            ldflags) -lssl -lcrypto -ldl
        ./encrypt 'i<3crypto' $(./priv_key 0x879a5ee58ade33942040f  0↩
            x3bef5e448f18ae4ff08c65 0x10001 2) 0x10001
    */

    if (argc != 4) {
        fprintf(stderr, "Usage: %s x n e\n", argv[0]);
        Py_Finalize();
        return 1;
    }

    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *x = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *y = BN_new();

    char *x_str = argv[1]; // x = argv[1]
    BN_hex2bn(&n, argv[2] + 2); // n = argv[2] ignore the 0x
```

```
    BN_hex2bn(&e, argv[3] + 2); // e = argv[3] ignore the 0x

    /*
        The hexify function will call the python script to convert the ↩
           message to hex
        and then store it in Bignum x
        parameters: script = "hex_ascii.py", function = "ascii_to_hex", ↩
           message = x_str, m = x
    */
    if (hexify("hex_ascii", "ascii_to_hex", x_str, x) != 0) {
        fprintf(stderr, "Failed to hexify message\n");
        Py_Finalize();
        return 1;
    }

    if (BN_cmp(x, n) >= 0)
    {
        printf("x is greater than or equal to n\n");
        Py_Finalize();
        return 1;
    }

    BN_mod_exp(y, x, e, n, ctx); // y = x^e mod n
    printBN("y = ", y); // Print the encrypted message

    // Free memory
    BN_CTX_free(ctx);
    BN_free(x);
    BN_free(n);
    BN_free(e);
    BN_free(y);
    Py_Finalize();
    return 0;
}
```

The expected output is:

```
./encrypt 'i<3crypto' $(./priv_key 0x879a5ee58ade33942040f
0x3bef5e448f18ae4ff08c65 0x10001 2) 0x10001
y =  01F88D6DC82F2D6A72A5633CF78A02CE26C236AC758A
```

Therefore
(1) The encrypted message is `0x01F88D6DC82F2D6A72A5633CF78A02CE26C236AC758A`

3.2: Decrypting a Message

Decryption in RSA is the inverse of encryption, so we will continue to use `BN_mod_exp` for modular exponentiation, only this time it will be applied to the ciphertext and the private key (d, n) replaces the public key used in encryption. Before we get to that point, we need to take the ciphertext C and convert it to a BIGNUM object using `BN_hex2bn`. Finally, after applying the decryption function below, we can use the above python script to convert the retrieved hexadecimal plaintext into ASCII.

From page 175 in the textbook:

RSA Decryption: Given the private key $(d, n) = k_{\text{pr}}$ and the ciphertext y, the decryption function is:

$$x = d_{k_{\text{pr}}}(y) \equiv y^d \mod n \quad (7.2)$$

where $x, y \in \mathbb{Z}_n$.

The code which decrypts a given encrypted message:

decrypt_message.c

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "hexify_dehexify.h"
#include <stdio.h>
#include <openssl/bn.h>
/*
    3.3
*/

void printBN(char *msg, BIGNUM *a)
{
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

int main(int argc, char *argv[])
{
    Py_Initialize();

    // Get the current working directory
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        // Convert current directory to Python object
        PyObject *sysPath = PySys_GetObject("path");
        PyObject *path = PyUnicode_FromString(cwd);
        // Append current directory to sys.path
        PyList_Append(sysPath, path);
        Py_DECREF(path);
    } else {
        perror("getcwd() error");
        return 1;
    }
    /*
        Expected input: ./program d n y
        d is the private key, n is the modulus, y is the encrypted ↩
            message (in hex)
        Example usage:
        cd path/to/current/directory
        gcc -o decrypt decrypt_message.c hexify_dehexify.c $(python3-↩
            config --cflags) -L/usr/lib -lpython3.10 $(python3-config --↩
            ldflags) -lssl -lcrypto -ldl
        ./decrypt $(./priv_key 0x879a5ee58ade33942040f  0↩
            x3bef5e448f18ae4ff08c65 0x10001 4) 0↩
            x0182c38e75c5a4889ec3c8da3602114b42e1d2cc9e58
    */
```

```
    if (argc != 4) {
        fprintf(stderr, "Usage: %s d n y\n", argv[0]);
        return 1;
    }

    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *x = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *y = BN_new();

    BN_hex2bn(&d, argv[1] + 2); // d = argv[1], ignore the 0x
    BN_hex2bn(&n, argv[2] + 2); // n = argv[2]
    BN_hex2bn(&y, argv[3] + 2); // y = argv[3]

    BN_mod_exp(x, y, d, n, ctx); // x = y^d mod n
    printBN("x = ", x); // Print the decrypted message in hex

    char *ascii_out = malloc(256);  // Allocate memory for the output ↩
        string
    if (ascii_out == NULL) {
        fprintf(stderr, "Failed to allocate memory\n");
        Py_Finalize();
        return 1;
    }

    char *hex_str = BN_bn2hex(x);
    if (dehexify("hex_ascii", "hex_to_ascii", hex_str, &ascii_out) != 0) ↩
        {
        fprintf(stderr, "Failed to dehexify message\n");
        free(ascii_out);  // Free the memory if dehexify fails
        Py_Finalize();
        return 1;
    }

    printf("Decrypted message: %s\n", ascii_out); // Print the decrypted ↩
        message in ASCII

    // Free memory
    free(ascii_out);
    BN_CTX_free(ctx);
    BN_free(x);
    BN_free(n);
    BN_free(y);
    Py_Finalize();
    return 0;
}
```

The expected output is:

```
./decrypt $(./priv_key 0x879a5ee58ade33942040f
0x3bef5e448f18ae4ff08c65 0x10001 4)
0x0182c38e75c5a4889ec3c8da3602114b42e1d2cc9e58
x =  6F6D6567616C756C
Decrypted message: omegalul
```

Therefore
(1) The decrypted message is `omegalul`

3.4: Signing a Message

Digital signatures use public-key cryptography, where the signer uses their private key (d, n) to create a signature for a message x, and appends it to the message. The receiver applies the corresponding public key (e, n) to the signature and verifies it by comparing it to the message. The message can be transmitted in clear text or encrypted (using AES, for example) as the signature protocol is primarily concerned with data authentication and integrity, not confidentiality.

Typically, a message is first compressed using a hashing function and then signed with the private key, addressing both security concerns and RSA's computational limitations for large messages. If the signature was created by signing a hash of the message, then the receiver will hash the message and compare this hash to the result obtained after applying the public key operation to the signature. In our case, we can simply apply the public key operation to the signature and compare that to the message for verification.

The code which generates a signature for a message:

message_signature_gen.c

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "hexify_dehexify.h"
#include <stdio.h>
#include <openssl/bn.h>
/*
    3.4
*/

void printBN(BIGNUM *a)
{
    char *number_str = BN_bn2hex(a); // binary to hex
    printf("0x%s", number_str);
    OPENSSL_free(number_str);
}

int main(int argc, char *argv[])
{
    Py_Initialize();

    // Get the current working directory
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        // Convert current directory to Python object
        PyObject *sysPath = PySys_GetObject("path");
        PyObject *path = PyUnicode_FromString(cwd);
        // Append current directory to sys.path
        PyList_Append(sysPath, path);
        Py_DECREF(path);
    } else {
        perror("getcwd() error");
        return 1;
    }

    /*
        Expected input: ./program d n x
        d is the private key in hex
        n is the modulus in hex
```

```
        x is the message in ascii
        Example usage:
        cd path/to/current/directory
        gcc -o sign message_signature_gen.c hexify_dehexify.c $(python3-↩
            config --cflags) -L/usr/lib -lpython3.10 $(python3-config --↩
            ldflags) -lssl -lcrypto -ldl
        ./sign $(./priv_key 0x879a5ee58ade33942040f  0↩
            x3bef5e448f18ae4ff08c65 0x10001 4) 'I owe you $100'
    */

    if (argc != 4) {
        fprintf(stderr, "Usage: %s d n x\n", argv[0]);
        Py_Finalize();
        return 1;
    }

    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *d = BN_new(); // private key
    BIGNUM *n = BN_new(); // modulus
    BIGNUM *x = BN_new(); // message
    BIGNUM *s = BN_new(); // signature

    BN_hex2bn(&d, argv[1] + 2);
    BN_hex2bn(&n, argv[2] + 2);
    char *x_str = argv[3];

    /*
        The hexify function will call the python script to convert the ↩
            message to hex
        and then store it in Bignum x
        parameters: script = "hex_ascii.py", function = "ascii_to_hex", ↩
            message = x_str, m = x
    */
    if (hexify("hex_ascii", "ascii_to_hex", x_str, x) != 0) {
        fprintf(stderr, "Failed to hexify message\n");
        Py_Finalize();
        return 1;
    }

    if (BN_cmp(x, n) >= 0)
    {
        printf("x is greater than or equal to n\n");
        Py_Finalize();
        return 1;
    }

    BN_mod_exp(s, x, d, n, ctx); // s = x^d mod n
    printBN(s); // Print the signature in hex
    printf("\n");

    BN_CTX_free(ctx);
    BN_free(d);
    BN_free(n);
    BN_free(x);
    BN_free(s);
    Py_Finalize();
    return 0;
}
```

The expected output is:

```
./sign $(./priv_key 0x879a5ee58ade33942040f
0x3bef5e448f18ae4ff08c65 0x10001 4) 'I owe you $100'
0x0101E8B9CF2C2F0FAFF042981804D38157594FD54479
```

Therefore,

(1) The signature generated for the message using the private key from step 1 is `0x0101E8B9CF2C2F0FAFF042981804D38157594FD54479`

(2) The private key or exponent was used to generate the signature.

(3) I changed the message to `'I owe you $100'` to `'I owe you $200'`

The expected output is:

```
./sign $(./priv_key 0x879a5ee58ade33942040f
0x3bef5e448f18ae4ff08c65 0x10001 4) 'I owe you $200'
0x47D8A1AAC753B4AFC113FC693202AFD89F537D3424
```

The two signatures are drastically different despite only changing one letter, with no apparent pattern as the changes are diffused throughout the signature. This is expected due to the design principles of cryptographic protocols.

3.5: Verifying a Signature

The verification of a signature should only return true if the signature, after applying the public key operation, is equal to the message it signed. Otherwise, it should return false.

The code that verifies a signature for a message:

verify_signature.c

```c
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "hexify_dehexify.h"
#include <stdio.h>

void printBN(BIGNUM *a)
{
    char *number_str = BN_bn2hex(a); // binary to hex
    printf("0x%s", number_str);
    OPENSSL_free(number_str);
}

void flip_bitBN(BIGNUM *a, int bit) {
    // if you wanted to flip the 5th bit, you would call flip_bitBN(a, 4)
    // or run the code again etc.
    BN_is_bit_set(a, bit) ? BN_clear_bit(a, bit) : BN_set_bit(a, bit);
}

int verify(BIGNUM *x, BIGNUM *s, BIGNUM *e, BIGNUM *n)
{
    // take the signature and apply modular exponentiation
    // and compare that to the message
```

```
    BIGNUM *result = BN_new();
    BN_CTX *ctx = BN_CTX_new();
    BN_mod_exp(result, s, e, n, ctx); // result = s^e mod n

    int comparison = BN_cmp(result, x);

    // Free memory
    BN_CTX_free(ctx);
    BN_free(result);

    return comparison == 0;
}

int main(int argc, char *argv[])
{
    Py_Initialize();
    // Get the current working directory
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        // Convert current directory to Python object
        PyObject *sysPath = PySys_GetObject("path");
        PyObject *path = PyUnicode_FromString(cwd);
        // Append current directory to sys.path
        PyList_Append(sysPath, path);
        Py_DECREF(path);
    } else {
        perror("getcwd() error");
        return 1;
    }
    /*
        Expected input: ./program x s e n
        x: message, s: signature, e: public key, n: modulus
        Example usage:
        cd a2
        gcc -o verify verify_signature.c hexify_dehexify.c $(python3-↩
            config --cflags) -L/usr/lib -lpython3.10 $(python3-config --↩
            ldflags) -lssl -lcrypto -ldl
        ./verify 'Launch a missile.'  0↩
            x643d6f34902d9c7ec90cb0b2bca36c47fa37165c0005cab026c0542cbdb6802f↩
              0x010001  0↩
            xae1cd4dc432798d933779fbd46c6e1247f0cf1233595113aa51b450f18116115↩

        ./verify 'Launch a missile.'  0↩
            x643d6f34902d9c7ec90cb0b2bca36c47fa37165c0005cab026c0542cbdb6803f↩
              0x010001  0↩
            xae1cd4dc432798d933779fbd46c6e1247f0cf1233595113aa51b450f18116115↩

    */
    if (argc != 5)
    {
        printf("Usage: %s x s e n\n", argv[0]);
        return 1;
    }

    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *x = BN_new();
    BIGNUM *s = BN_new();
    BIGNUM *e = BN_new();
```

```
    BIGNUM *n = BN_new();

    char *x_str = argv[1];
    BN_hex2bn(&s, argv[2]+2); // s = argv[2] + 2 to skip the 0x
    BN_hex2bn(&e, argv[3]+2); // e = argv[3] + 2
    BN_hex2bn(&n, argv[4]+2); // n = argv[4] + 2

    /*
        The hexify function will call the python script to convert the ↩
           message to hex
        and then store it in Bignum x
        parameters: script = "hex_ascii.py", function = "ascii_to_hex", ↩
           message = x_str, m = x
    */
    if (hexify("hex_ascii", "ascii_to_hex", x_str, x) != 0) {
        fprintf(stderr, "Failed to hexify message\n");
        Py_Finalize();
        return 1;
    }

    if (BN_cmp(x, n) >= 0)
    {
        printf("x is greater than or equal to n\n");
        Py_Finalize();
        return 1;
    }

    verify(x, s, e, n) ? printf("True\n") : printf("False\n");
    // Free memory
    BN_CTX_free(ctx);
    BN_free(x);
    BN_free(s);
    BN_free(e);
    BN_free(n);
    Py_Finalize();
    return 0;
}
```

The expected output is:

```
./verify 'Launch a missile.'
0x643d6f34902d9c7ec90cb0b2bca36c47fa37165c0005cab026c0542cbdb6802f
0x010001
0xae1cd4dc432798d933779fbd46c6e1247f0cf1233595113aa51b450f18116115
True
```

Therefore,

(1) The verification of the signature for the given message returned true

(2) Changing one bit (for example, 2F to 3F in the last byte) will corrupt the signature, so it should return false.

The expected output is:

```
./verify 'Launch a missile.'
0x643d6f34902d9c7ec90cb0b2bca36c47fa37165c0005cab026c0542cbdb6803f
```

```
0x010001
0xae1cd4dc432798d933779fbd46c6e1247f0cf1233595113aa51b450f18116115
False
```

3.6: Cracking an RSA Private Key

Most, if not all, of the libraries available for integer factorization require the integer to be expressed in decimal. A simple script to accomplish this is given here (you will need to install the command bc):

hex_to_dec.sh

```
#!/bin/bash

# Check for correct number of arguments
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <hex_number>"
    exit 1
fi

HEX_NUMBER=$1

# Remove the '0x' prefix if present
HEX_NUMBER=${HEX_NUMBER^^} # Convert to uppercase to ensure consistency
HEX_NUMBER=${HEX_NUMBER#0X} # Remove '0X' prefix if exists

# Convert hexadecimal num to decimal
DECIMAL_NUMBER=$(echo "ibase=16; $HEX_NUMBER" | bc)

# Print the decimal equivalent
echo "Decimal equivalent: $DECIMAL_NUMBER"
```

This will be used alongside the integer factorization library and private_key_gen.c from 3.1 to crack the corresponding RSA private key for each of the following public keys.

(1) The modulus `n = 0x7c5cfe617c286a27ffc10ecf88a8d35ebbf1e30320af` is converted to decimal:

```
./hex_to_dec.sh 0x7c5cfe617c286a27ffc10ecf88a8d35ebbf1e30320af
Decimal equivalent:
46529818383710374672711659156505490446378563486818479
```

The two prime factors of the modulus are found using the YAFU (Yet Another Factoring Utility) library:

```
YAFU Version 2.11
Built with GCC 11
Using GMP-ECM 7.0.6-dev, Powered by GMP 6.3.0
Detected Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
Detected L1 = 32768 bytes, L2 = 6291456 bytes, CL = 64 bytes
CPU features enabled:
Using 1 random witness for Rabin-Miller PRP checks
Cached 664579 primes; max prime is 9999991
Parsed yafu.ini from /Math/yafu

===============================================================
======= Welcome to YAFU (Yet Another Factoring Utility) =======
```

```
=======              bbuhrow@gmail.com                    =======
=======     Type help at any time, or quit to quit       =======
================================================================

>> factor(46529818383710374672711659156505490446378563486818479)
fac: factoring 46529818383710374672711659156505490446378563486818479
fac: using pretesting plan: normal
fac: using specified qs/gnfs crossover of 100 digits
fac: using specified qs/snfs crossover of 75 digits
div: primes less than 10000
fmt: 1000000 iterations
rho: x^2 + 3, starting 1000 iterations on C53
rho: x^2 + 2, starting 1000 iterations on C53
rho: x^2 + 1, starting 1000 iterations on C53
pm1: starting B1 = 150K, B2 = gmp-ecm default on C53
ecm: 30/30 curves on C53, B1=2k, B2=gmp-ecm default
ecm: 18/18 curves on C53, B1=11k, B2=gmp-ecm default

starting SIQS on c53: 46529818383710374672711659156505490446378563486818479

==== sieving in progress (  2 threads):    1952 relations needed ====
====             Press ctrl-c to abort and save state            ====
1979 rels found: 921 full + 1058 from 9813 partial, (30301.32 rels/sec)

building matrix with 1979 columns
SIQS elapsed time = 0.4030 seconds.
Total factoring time = 1.1992 seconds


***factors found***

P27 = 293906828378037345736457113
P27 = 158314859986381280032123783

ans = 1
```

The corresponding RSA private key can be found now, via private_key_gen.c:

```
./priv_key $(echo "obase=16; ibase=10;
293906828378037345736457113" | bc |
awk '{print "0x" $0}') $(echo "obase=16; ibase=10;
158314859986381280032123783" | bc | awk '{print "0x" $0}') 0x10001 1
0x39C1E2AE99C017554E557582B85DCA21843C2BFCDDD1
```

(2) The modulus `n = 0x1e6f1558fe63761406be065a9e07b060f8360b2725b09f4071` is converted to decimal:

```
./hex_to_dec.sh 0x1e6f1558fe63761406be065a9e07b060f8360b2725b09f4071
Decimal equivalent: 191036808839073752998059970737347860342917676432172452364401
```

The two prime factors of the modulus are found using the YAFU (Yet Another Factoring Utility) library:

```
YAFU Version 2.11
Built with GCC 11
Using GMP-ECM 7.0.6-dev, Powered by GMP 6.3.0
Detected Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
Detected L1 = 32768 bytes, L2 = 6291456 bytes, CL = 64 bytes
CPU features enabled:
Using 1 random witness for Rabin-Miller PRP checks
Cached 664579 primes; max prime is 9999991
Parsed yafu.ini from /Math/yafu

===============================================================
======= Welcome to YAFU (Yet Another Factoring Utility) =======
=======             bbuhrow@gmail.com                   =======
=======     Type help at any time, or quit to quit      =======
===============================================================

>> factor(191036808839073752998059970737347860342917676432172452364401)
fac: factoring 191036808839073752998059970737347860342917676432172452364401
fac: using pretesting plan: normal
fac: using specified qs/gnfs crossover of 100 digits
fac: using specified qs/snfs crossover of 75 digits
div: primes less than 10000
fmt: 1000000 iterations
rho: x^2 + 3, starting 1000 iterations on C60
rho: x^2 + 2, starting 1000 iterations on C60
rho: x^2 + 1, starting 1000 iterations on C60
pm1: starting B1 = 150K, B2 = gmp-ecm default on C60
ecm: 30/30 curves on C60, B1=2k, B2=gmp-ecm default
ecm: 49/49 curves on C60, B1=11k, B2=gmp-ecm default

starting SIQS on c60: 191036808839073752998059970737347860342917676432172452364

==== sieving in progress (  2 threads):    3504 relations needed ====
====             Press ctrl-c to abort and save state            ====
3535 rels found: 1743 full + 1792 from 17230 partial, (16901.64 rels/sec)

building matrix with 3535 columns
SIQS elapsed time = 1.2130 seconds.
Total factoring time = 2.6347 seconds


***factors found***

P30 = 335403766841212825761913864577
P30 = 569572639682113597061986473713

ans = 1
```

The corresponding RSA private key can be found now, via private_key_gen.c:

```
./priv_key $(echo "obase=16; ibase=10;
335403766841212825761913864577" | bc |
```

```
awk '{print "0x" $0}') $(echo "obase=16; ibase=10;
569572639682113597061986473713" | bc | awk '{print "0x" $0}') 0x10001 1
0x06C5DFF15B43D54E65A5BA9243985CA4E0F3C03FFCCA3E2801
```

(3) The modulus `n = 0x3bb8210adb66a6a26e0bcb4837a1dd1ddd605e4295f592631c0 6358734e627261718681cbdf7` is converted to decimal:

```
./hex_to_dec.sh 0x3bb8210adb66a6a26e0bcb4837a1dd1ddd605e4295f592631c063
58734e627261718681cbdf7
Decimal equivalent: 760314182869541909192306679884601127330067391488593
704283924637776785854420138490920624075 9
```

The two prime factors of the modulus are found using the CADO-NFS library:

```
/Math/cado-nfs# ./cado-nfs.py 760314182869541909192306679884601127330067
391488593704283924637776785854420138490920624075 9
Info:root: Using default parameter file ./parameters/factor/params.c90
Info:root: No database exists yet
Info:root: Created temporary directory /tmp/cado.jzg09pb_
Info:Database: Opened connection to database /tmp/cado.jzg09pb_/c90.db
Info:root: Set tasks.linalg.bwc.threads=8 based on detected physical cores
Info:root: Set tasks.threads=8 based on detected logical cpus
Info:root: tasks.threads = 8 [via tasks.threads]
Info:root: tasks.polyselect.threads = 2 [via tasks.polyselect.threads]
Info:root: tasks.sieve.las.threads = 2 [via tasks.sieve.las.threads]
Info:root: tasks.linalg.bwc.threads = 8 [via tasks.linalg.bwc.threads]
Info:root: tasks.sqrt.threads = 8 [via tasks.threads]

...
Info:Complete Factorization / Discrete logarithm: Total cpu/elapsed time
for entire Complete Factorization 8469.75/1964.84
Info:root: Cleaning up computation data in /tmp/cado.jzg09pb_
490380731495195857998205096862830195023394555 7
155045688795990248701223094788463727054136898 7
```

The corresponding RSA private key can be found now, via private_key_gen.c:

```
./priv_key $(echo "obase=16; ibase=10; 490380731495195857998205096862830
1950233945557" | bc | awk '{print "0x" $0}') $(echo "obase=16; ibase=10;
155045688795990248701223094788463727054136898 7" | bc | awk '{print "0x" $0}')
0x10001 1
0x0B49CED5228BD0CBBB49776F7B34997B5FB12B696D89F0C8BE10DE9C5CCF0BDA2DD29F4E7921
```

(4) I started off with a tool called YAFU, which appeared to be the most popular integer-factorization tool. I was able to follow the instructions here: https://ftp.mersenneforum.org/showthread.php?t=23087 regarding installion which involved several repositories for different factorization methods.
It was very fast for 3.6.1 and 3.6.2, with times of 1.1992 seconds for the first modulus and 2.6347 seconds for the second one.

However, I soon ran into problems when trying to use YAFU on my virtual machine, as it took significantly longer to factor the final modulus. As it would freeze near the end, I decided to find a new method. I settled on CADO-NFS, which is another integer-factorization tool/library which has been used to solve several RSA factorization challenges. I again followed the installation instructions on the same site, https://www.mersenneforum.org/showthread.php?t=23089. This took much less time than it

would take using YAFU, as it was using the NFS (Number Field Sieve) Algorithm which is the fastest algorithm and suited towards larger integers. Nevertheless, it took 1964.84 seconds.
My computer's specifications are a 1.4 GHz Quad-Core Intel Core i5 processor and 8 GB of RAM. It appears that the cracking time (if we are to compare across different tools/factorization methods) does not increase linearly with integer size. It is likely exponential, since there is no known polynomial-time algorithm for integer factorization.