RSA Encryption Tool

Nebhrajani, Aditya V.

Acknowledgements

No project can ever be completed without the graceful contribution of many, many people.

This entire project was written on open source software, using open source programming and markup languages. I am hence deeply indebted to the open source community at large. I would like to separately acknowledge the GNU Emacs[1], LaTeX[2], Python[3] and the StackExchange[4] communities for their powerful – and open source – tools.

Lastly, I would like to thank my parents for their love and guidance without which nothing can be accomplished.

[1]The Emacs text editor/eLISP REPL: https://www.gnu.org/software/emacs/
[2]LaTeX document creation system: https://www.latex-project.org/
[3]Python programming language: https://www.python.org/
[4]StackExchange forums: https://stackexchange.com/

Contents

List of Figures

Source Code Listings

1 Introduction

Democratic governments the world over have a similar system of guaranteeing rights to people. However, one right which is still not guaranteed by most is the right to privacy. Prior to the advent of computer technology, privacy was easy to ensure: one had but to close the door or seal a letter. Things have changed since then: humans communicate more and more over the Internet. This makes our communications less private, since the medium of communication is not secure.

In any case, there are tremendous vested interests in invading people's privacy, from showing more personalised ads to societal control and conditioning. After whistleblower Edward Snowden leaked the United States' National Security Agency's highly classified information, it is public domain knowledge that not only companies but also governments invade our privacy to their own ends.

It is difficult to overstate how morally incorrect and dangerous such a practice is. Snowden put the situation excellently:

> "Arguing that you don't care about privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say."

The irony of the situation is that most people are aware that companies such as Google, Microsoft and Facebook steal their data[5], but they do nothing about it. This is best explained through consumers preferring convenience over privacy. Google is easier to use than DuckDuckGo, and is better developed. To expect anybody other than a few highly privacy-conscious individuals to stop using Instagram and Windows 10 is not an ideal solution. Instead, the convenience of encryption needs to be available to the general public in a user-friendly interface. Ideas like PGP have tried and fallen into obscurity, PAKE didn't gain traction[6], and people are trusting, or trying to trust, WhatsApp's 'end-to-end-encryption'. There is an opening for a system that lets people secure their privacy while also being accessible and easy to use.[4][7]

This project is an attempt to remedy this situation by building a framework for quick and easy to use encryption. It's written in Python, which makes it easy to port or include in other applications. The interactive CLI and Pandas database are merely for demonstration purposes. Ideally, this project should be used as a module or as part of a larger communication application.

[5] Reference: Ask anyone.

2 Cryptography

2.1 In History

Since time immemorial, kings, queens, and armies have relied on the privacy of their communications to communicate battle plans to units spread over a large area. Other people reliant on privacy were secret lovers, those inciting rebellion, and even Mary Queen of Scots.

Privacy was ensured by somehow obscuring the original message using methods such as steganography or cryptography. Steganography is of less interest to us, as it involves physically hiding a message, for instance, by engraving it in a hidden place. The other method, cryptography, involved taking the original message, the 'plaintext', and converting it via an algorithm to an obscure mess of letters, the 'ciphertext'. The receiver then performed the reverse algorithm and retrieved the plaintext. Thus, even if the message were intercepted, it could not be read and its information exploited.[6]

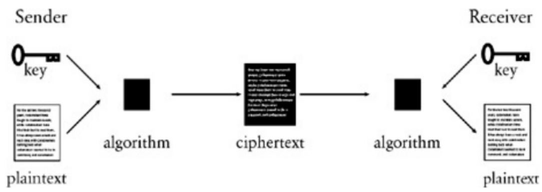

Figure 1: Principle of encryption

In cryptography, the security of the whole system depends not on the security of the communication medium but rather on the algorithm used. An example is the very simple Caesar cipher[2]:

> The action of a Caesar cipher is to replace each plaintext letter with a different one a fixed number of places down the alphabet. The cipher illustrated here uses a left shift of three, so that (for example) each occurrence of E in the plaintext becomes B in the ciphertext.

In this case, the 'key' would be the number of places to shift each letter by, or the 'shift'. The Caesar cipher, like many others of its time, was broken by frequency analysis: given a long enough sample of the ciphertext, a cryptanalyst could replace the most frequently occurring letter by the one used most in the English alphabet ('e'), or the second most used ('t'), and so on. By trial and error, the original message is soon revealed. In fact, the cipher of Mary Queen of Scots was broken by frequency analysis by a cryptanalyst called Thomas Phelippes. Once the cipher was broken, the Babington plot was revealed, and she was executed, with the cracked cipher being the central evidence in the case against Mary Queen of Scots.

Much later, during the Second World War, the Polish and British forces joined hands to crack the German Enigma cipher, which used an Enigma machine to encrypt the plaintext. The Polish designed machines to crack the Enigma code (called 'bombes' for the loud noises they made), which were further refined by Alan Turing, who insisted on spending time designing mechanical machines that could break the cipher rather than spend time cracking the ciphers by hand. This was one of the first instances of machines being used to encrypt plaintext, and with Turing, machines being used to break ciphers.

2.2 Modern Methods and Some Mathematics

The primary problems in cryptography are key distribution and the security of the ciphertext. While the security of the ciphertext can, in theory, be ensured by using complicated algorithms implemented by machines, the problem of secure key distribution remains. If the keys are distributed through an insecure communication medium, this renders the algorithm a mere computational hurdle at best.

[6] Image credit: The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography, Simon Singh. OCLC: 59459928

2.2.1 Diffie-Hellman Key Exchange

To solve the problem of secure key-distribution, Diffie, Hellman, and Merkle created a method for generating keys even on an insecure communication medium. [3] The method is shown in the figure below.[7]

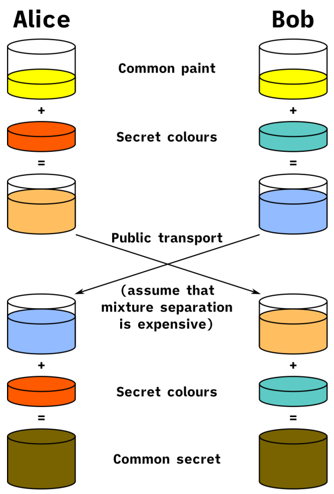

Figure 2: Diffie-Hellman key exchange

As evident from the figure, a man-in-the-middle, despite having knowledge of the starting color Alice and Bob agree on (yellow), and the intermediaries that they exchange (orange-tan and light blue), he cannot find the final key since Alice and Bob do not exchange information about the secret colors they use (blue and orange). In practice, of course, Alice and Bob use large numbers and the modulo function (modulus from clock math) to eventually get the same secret key, which is a large integer. This algorithm, proposed in 1976, is used to generate secret keys throughout the Internet. A simple generalisation of the principle used (using finite cyclic groups) is as follows (a cyclic group is generated by a single element, and any element in the group can be obtained by repeated group operations. A clock with an arithmetic of addition defined is a cyclic group of order 12.):

1. Alice and Bob agree on a finite cyclic group G (multiplicative) of order n and a generating element g in G. (g is assumed to be known by all attackers.)
2. Alice picks a random natural number a ($1 < a < n$) and sends g^a to Bob.
3. Bob picks a random natural number b ($1 < b < n$) and sends g^b to Alice.
4. Alice finds $(g^b)^a$.
5. Bob finds $(g^a)^b$.
6. Since $(g^b)^a = (g^a)^b = g^{ab}$, both Alice and Bob have the same group element, which serves as the secret key.

We note, however, that it is likely organisations with large budgets can crack Diffie-Hellman, especially ones that use keys of 1024-bit lengths or less. Moreover, other vulnerabilities have been found in Diffie-Hellman, as described in the paper *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice.* [1] According to Adrian et al. around two-thirds of popular HTTPS sites allow TLS using Diffie-Hellman and 1024-bit-keys. Some even allow legacy 512-bit keys. Two-thirds of the HTTPS servers analyzed also use common groups for generation, meaning some primes are more 'popular' than others. These vulnerabilities were exploited by the paper and the Logjam attack was used to cryptanalyze Diffie-Hellman.

Diffie-Hellman inspired a much more powerful algorithm, the RSA asymmetric key encryption.

[7] Image credit: Diffie-Hellman key exchange - Wikipedia/Wikimedia Commons

2.2.2 RSA Encryption

RSA encryption[8], named after Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman, uses the concept of each user having two keys: one public key and one private key. Assume these keys are pre-generated. Say Bob wants to send a message M to Alice. Alice's public key is a tuple of integers (n, e) and her private key is an integer (d). The public key is available to everyone, maybe Alice even keeps it in her Instagram bio.

1. Bob converts his message M into an integer m using a pre-decided scheme (specified by a standard such as PKCS#1).
2. Bob gets Alice's public key, (n, e) and computes

$$m^e \equiv c \pmod n$$

   He then sends Alice c over the communication medium.
3. Alice recovers m from c by using her private key d, thus:

$$c^d \equiv (m^e)^d \equiv m \pmod n$$

4. Alice then reverses the message-to-integer scheme and gets M from m.

This method results in only two pieces of information passing through the communication channel: Alice's public key, and the encrypted message, as shown in figure.[8]

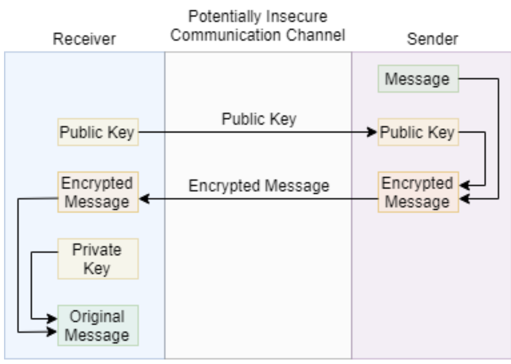

Figure 3: RSA encryption

Key Generation Keys are generated using two large primes, and this is what offers RSA its high level of security once encrypted. The exact procedure is as follows:

1. Choose two distinct primes p and q, at random. These are kept secret.
2. Compute $n = pq$. n is used as the modulus, and its length expressed in bits is defined as the "key length". n is part of the public key tuple.
3. Compute $\lambda(n)$, where λ is Carmichael's totient function. Since $n = pq$,

$$\lambda(n) = \mathrm{LCM}(p - 1, q - 1)$$

   $\lambda(n)$ is kept secret.
4. Choose integer e such that e and $\lambda(n)$ are coprimes. e is usually set to $2^{16} + 1 = 65537$, since it has a low Hamming weight (its binary representation has a lot of zeros). e is called the public exponent.

[8]Image credit: Jin Kyu Lim, Medium https://medium.com/@jinkyulim96/algorithms-explained-rsa-encryption-9a37083aaa62

5. Find d by solving

$$d \cdot e \equiv 1 \pmod{\lambda(n)}$$

that is, taking the modular multiplicative inverse of $e \pmod{\lambda(n)}$. d is the private key and is kept secret.

We hence have the public key tuple (n, e) and the private key d. The above operations are usually carried out using various optimized algorithms, which will be discussed while implementing the Python code.

2.2.3 RSA Proof and Example

Fermat's Little Theorem Fermat's Little Theorem states that $a^{p-1} \equiv 1 \pmod{p}$ for any integer a and prime p not dividing a.

RSA Proof We are to show,

$$(m^e)^d \equiv m \pmod{pq} \forall m$$

where p and q are distinct primes, and e and d are positive integers satisfying

$$ed \equiv 1 \pmod{\lambda(pq)}$$

Since $\lambda(n) = \text{LCM}(p-1, q-1)$, by construction, divisible by both $p-1$ and $q-1$:

$$ed - 1 = h(p-1) = k(q-1)$$

for some natural numbers h and k.

From the Chinese Remainder Theorem,

> To check whether two numbers, like m^{ed} and m, are congruent mod pq, it suffices (and in fact is equivalent) to check that they are congruent mod p and mod q separately.

To show $m^{ed} \equiv m \pmod{p}$, we consider two cases:

1. If $m \equiv 0 \pmod{p}$, m is a multiple of p (multiplicative cyclic group). Thus, m^{ed} is a multiple of p. Thus, $m^{ed} \equiv 0 \equiv m \pmod{p}$.
2. If $m \not\equiv 0 \pmod{p}$,

$$m^{ed} = m^{ed-1}m = m^{h(p-1)}m \equiv 1^h m \equiv m \pmod{p}$$

where $m^{h(p-1)} \equiv 1^h m \pmod{p}$ by Fermat's Little Theorem.

We proceed similarly for q, which completes the proof.

RSA Example The following is a simple proof of concept of RSA.

1. Let $p = 11$, $q = 3$.
2. $n = 11 \times 3 = 33$
3. $\phi = (p-1)(q-1) = 20$. Note that we're not taking the LCM. [9]
4. Choose $e = 3$.
5. Find d such that $ed \equiv 1 \pmod{\phi}$. By trial and error, $d = 7$.
6. Hence, the public key is $(33, 3)$ and the private key is 7.
7. Let us encrypt $m = 7$.

$$c \equiv m^e \equiv 7^3 \equiv 343 \pmod{33} = 13$$

The ciphertext is hence 13.

8. Now, decrypting,

$$m \equiv c^d \equiv 13^7 \pmod{33} = 7$$

We hence retrieve the original message.

[9] We are using Euler's rather than Carmichael's totient function. Any d satisfying the former satisfies the latter, however, sometimes, Euler's yields a d larger than required. In this case, the distinction is irrelevant. This doesn't matter for implements that use the Chinese Remainder Theorem, as d is not used directly.

3 Python Implementation

We now focus on building a complete RSA system in Python (3.6) that is capable of generating keys, maintaining a database of users and their public/private keys, and encrypting and decrypting ASCII text files of arbitrary length in reasonable time.

Wherever further mathematics is required for implementing a certain primitive or algorithm, it is discussed prior its implementation in code.

3.1 Installation

The source code for this project is available on GitHub. You should either download it from `https://github.com/nebhrajani-a/rsa-py` or clone it into a new directory.

```
$ git clone https://github.com/nebhrajani-a/rsa-py ./rsa-py
```

3.1.1 Dependencies

The major non-standard dependancy is `progress`, which draws the progress bars during encryptions and decryptions.

```
$ pip install progress
```

Other (standard) dependancies are Python 3 (note that this program will **not** run on Python 2), math (for the ceiling function), Secrets (secure random number generation), getpass (secure password entry) and Pandas (database management).

3.1.2 Compatibility

This program is written to be platform independent, using Python implementations everywhere rather than (possibly simpler) alternatives like command line calls. It's been tested on a machine running Linux Mint 19.3 (XFCE 64-bit) and in two Microsoft Windows virtual machines (7 and 10), and behaves as expected. If you think you are experiencing bugs, please write an email to aditya.v.nebhrajani@gmail.com.

3.1.3 Executing

To use the program, `cd` to `/src` and execute

```
$ python main.py
```

Note: If you use Windows, Python may be aliased to `py` instead. Make sure `python` runs Python 3 and not Python 2. On some Linux systems Python 3 needs to be called using `python3`.

3.2 Key Generation

To generate keys, we must have: a large prime generator for p and q, and math functions for taking the LCM and modular multiplicative inverse of two numbers.

3.2.1 Large Prime Generation

Large prime generation is a mathematics problem as there is no evident pattern to finding primes. $\pi(n)$ is the prime counting function for primes $\leq n$. According to the prime number theorem[10]

$$\pi(n) \sim \frac{n}{\ln n}$$

or the probability of a random n being prime is $1/\ln n$. For instance, the probability of a random 2048 bit number being prime is $1/\ln 2^{2048} \approx 1/1419$.

[10] Which describes the asymptotic distribution of primes in the natural numbers. In fact, it is more explicitly stated as $\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1$.

Immediate Improvements We can improve the previously calculated probability by noting that no prime other than two is a multiple of two, meaning we have to check only half of the otherwise required 1419 numbers, so ~ 710 numbers must be checked.

Now, to check if a number is prime, we need to divide it by every divisor d such that $1 < d < n$. There's another improvement here: we only have to check till $\sqrt{n}$ since if n is composite and equal to pq, either $p \leq \sqrt{n}$ or $q \leq \sqrt{n}$.

Or, in Python:

```
def prime_generator(n):
    if n % 2 == 0:
        return False
    for i in range(3, sqrt(n), 2):
        if n % d == 0:
            return False
    return True
```

Listing 1: Basic Prime Number Generator

This function certainly works, but has a time complexity $\mathcal{O}(\sqrt{n})$. This will need to be faster for generating primes as large as 2048 bits. We hence turn to the probability based Miller-Rabin primality test.

Non-Trivial Square Roots We first define trivial and non-trivial square roots. The trivial square roots of $1 \mod p$ are 1 and -1, as they always return 1 on being squared modulo p, where p is a prime satisfying $p > 2$.

We define a non-trivial square root of $1 \mod p$ as any root other than 1 and -1. We prove now that if such a root exists, p cannot be prime (a special case of the result that, in a field, a polynomial has as many zeros as its degree).

$$x^2 \equiv 1 \pmod p$$
$$(x-1)(x+1) \equiv 0 \pmod p$$

Which means $x \equiv 1$ or $-1 \pmod p$.

For example, consider $3^2 = 9 \equiv 1 \pmod 8$, meaning 3 is a non-trivial square root of $1 \mod 8$. Clearly, 8 is composite.

Miller-Rabin Primality Test The Miller-Rabin test relies on finding nontrivial square roots of $1 \mod n$.

Let n be a prime and $n > 2$. $n-1$ must be even, and we write it as $2^s \times d$, where s and d are positive integers and d is odd. For each a in $(Z/nZ)\times$ (in range $[2, n-2]$), either

$$a^d \equiv 1 \pmod n$$

or

$$a^{2^r \cdot d} \equiv -1 \pmod n$$

for some $0 \leq r \leq s-1$. To prove the claim, check that by Fermat's Little Theorem, $a^{n-1} \equiv 1 \pmod n$. Hence, if we repeatedly take the square roots of a^{n-1}, we will get either 1 or -1.

The Miller-Rabin test is the contrapositive of the above, that if we can find an a such that both of the above hold false, n must be composite. a is called a witness to the fact that n is non-prime. If some a does not satisfy the negation of both the above, a is called a strong liar, and n is a strong *probable* prime for base a (that is, it might yet be composite and still hold the contrapositive).

Every odd composite number n has many witnesses a, however, if we're unlucky, a will be a strong liar for n and we'll categorize it as prime. There's no known way of generating an a which isn't a liar. We instead make the test probability based: we choose a random non-zero a and test it. Since it might be a strong liar, we repeat the test for a different value of a. The greater the number of iterations we check, the higher is the probability of generating a prime n.

The time complexity of the Miller-Rabin test is $\mathcal{O}(k \log^3 n)$, which means it is a polynomial time algorithm, a vast improvement over the simple method discussed earlier.

Final Implementation We first generate a random number n using Python's `secrets` module. [11] We first implement a low level checker: we divide the randomly generated n by every element in a static list of the first 2000 primes. This eliminates obvious composite numbers early on, speeding up our work. Also, this prevents some strong liars in Miller-Rabin. This is implemented as follows:

```
import first_primes as fp
import secrets

def gen_random(l):
    '''Generate a random number l bits long.'''
    randgenerator = secrets.SystemRandom()
    return randgenerator.randrange(2**(l-1)+1, 2**l - 1)
```

Listing 2: l-bit Random Generator

Note that the random number generated is in the range $[2^{l-1}+1, 2^l-1]$ to ensure the correct bit-length and to ensure that no extra even numbers are checked. Also note that `first_primes` is just a file with one list declaration. Files can be imported as modules in Python 3 without an `__init.py__` file.

```
def low_level_checker(l):
    '''Check that the random number isn't divisible by the first few primes.'''
    while True:
        x = gen_random(l)
        for divisor in first_primes:
            if x % divisor == 0 and divisor**2 <= x:
                break
        else: return x
```

Listing 3: Low Level Primality Test

n is then divided by every number in the first 2000 primes which is less than $\sqrt{n}$. Miller Rabin is then implemented exactly as described mathematically:

```
def miller_rabin_checker(mrc):
    '''Run 40 iterations of the Miller-Rabin Primality Test.'''
    randgenerator = secrets.SystemRandom()
    max_divisions_by_two = 0
    y = mrc-1
    while y % 2 == 0:
        y >>= 1
        max_divisions_by_two += 1
    assert(2**max_divisions_by_two * y == mrc-1)

    def trial_composite(round_tester):
        if pow(round_tester, y, mrc) == 1:
            return False
        for i in range(max_divisions_by_two):
            if pow(round_tester, 2**i * y, mrc) == mrc-1:
                return False
        return True
    number_of_rabin_trials = 40
    for i in range(number_of_rabin_trials):
        round_tester = randgenerator.randrange(2, mrc)
        if trial_composite(round_tester):
```

[11] The `random` module is not suitable for cryptographic applications. Secrets instead calls the system's most secure source of randomness, such as /dev/random on Linux.

```
            return False
    return True
```

Listing 4: Miller-Rabin Primality Test

Note that the third argument of Python's `pow()` function takes an optional argument for the modulus.

Now that there is an implementation for both a low level and a high level checker (Miller-Rabin), the last requirement is a driver function to actually generate the prime.

```
def driver(l):
    while True:
        prime_candidate = low_level_checker(l)
        if not miller_rabin_checker(prime_candidate):
            continue
        else:
            return prime_candidate
```

Listing 5: Prime Generation Driver

This driver function keeps generating random numbers until a number passes both the low level checker and 40 iterations of Miller-Rabin[12], which means it has an acceptably high probability of being prime.

3.2.2 Mathematics Functions

Now that there is a framework for generating the random primes p and q, a framework for taking the LCM (lowest common multiple) and the modular multiplicative inverse is required.

LCM For any integers a and b, $ab = \text{LCM}(a, b) \times \text{GCD}(a, b)$. Calculating the GCD (greatest common divisor or highest common factor) of two numbers is a simple recursive algorithm called the (Standard) Euclidean Algorithm. It involves a succession of Euclidean divisions (division with remainder).

First, take the larger of a and b, the two numbers whose GCD we are to find. Divide the larger (say a), by the smaller (say b). Then, divide b by the remainder of this division. Continue to take successive remainders as divisors for the next step and divisors for the former as dividends for the latter. Continue this process until a remainder of zero is obtained. The divisor for this operation is the GCD. The proof is trivial and well-known, so we skip to the Python implementation.

```
import sys
sys.setrecursionlimit(10**6)

def gcd(a,b):
    if a == 0:
        return b
    return gcd(b % a, a)
```

Listing 6: Recursive GCD Using Euclidean Algorithm

Note that the system recursion limit is increased to 10^6 to prevent Python's stack overflow inhibitor from kicking in for large numbers. While an iterative GCD function could be used, recursive functions are funner and simpler to write. In addition, Python uses enough system resources that this increase is not too resource-intensive. We do note, however, that Python is not a very fast language and an iterative implementation is a better choice for real world applications.

The LCM is now trivial to calculate by simple division:

[12] 40 iterations were chosen from the reasoning in `https://stackoverflow.com/a/6330138`

```python
def lcm(a,b):
    return (a*b) // gcd(a,b)
```

Listing 7: LCM Finder Using GCD

'Floor' division is used as opposed to 'true' division to prevent integer to float conversion, which results in rounding and loss of LSB information.

Modular Multiplicative Inverse The modular multiplicative inverse is calculated using the Extended Euclidean Algorithm, which is similar to the Standard Euclidean Algorithm used for calculating the GCD, except that the we use two sequences rather than one. In the Standard Euclidean Algorithm the first divisors are represented by $r_0 = a$ and $r_1 = b$ with the sequence following through $r_{i+1} = r_{i-1} - q_i r_i$ till $r_{k+1} = 0$ and r_k is the GCD.

In the Extended Euclidean Algorithm, the two sequences start with three values each, the first being $r_0 = a$, $s_0 = 1$, $t_0 = 0$ and the other $r_1 = b$, $s_1 = 0$, $t_1 = 1$. The sequence is followed through $r_{i+1} = r_{i-1} - q_i r_i$, $s_{i+1} = s_{i-1} - q_i s_i$, $t_{i+1} = t_{i-1} - q_i t_i$, until we find $r_{k+1} = 0$ and r_k as GCD.

However, s_k and t_k are the Bézout coefficients of a and b[13]. Bézout's identity is defined as:[14]

> Bézout's identity — Let a and b be integers with greatest common divisor d. Then, there exist integers x and y such that $ax + by = d$. More generally, the integers of the form $ax + by$ are exactly the multiples of d.

The Bézout's coefficients can be used to efficiently calculate the modular inverse, thus: $ax + by = \mathrm{GCD}(a, b) = 1$ or, $ax - 1 = (-y)b$ or,

$$ax \equiv 1 \pmod b$$

which completes our task. This algorithm runs in $\mathcal{O}(\log b^2)$ time. The Python implementation of the above two algorithms is as follows:

```python
def eea(a, b):
    '''Extended Euclidean algorithm'''
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = eea(b % a, a)
        return (g, x - (b // a) * y, y)
```

Listing 8: Extended Euclidean Algorithm

which is very similar to the recursive GCD algorithm, albeit with different sequences.

```python
def modinv(a, m):
    '''Find modular multiplicative inverse using EEA'''
    g, x, y = eea(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist.')
    else:
        return x % m
```

Listing 9: Modular Inverse Using Extended Euclidean Algorithm

which simply calls `eea()` and takes the modulus.

[13] The proof for this claim is not within the scope of this paper.

[14] From Wikipedia, `https://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity`

3.2.3 Putting It Together

We are now set to generate keys, having all the required functions to do so. This is done as outlined in Section 2.2.2. The code is self-explanatory.

```
import prime_generator as pg
import math_functions as mf

def gen_keys(bits):
    p = pg.driver(bits//2)
    q = pg.driver(bits//2)
    n = p*q
    lambda = mf.lcm((p-1), (q-1))
    e = 65537
    d = mf.modinv(e, lambda)
    return [p, q, n, e, d, bits]
```

Listing 10: Key Generator

`gen_keys()` takes an argument for the bit-length of the resulting public key n, and returns a list of the prime factors of n, n, the public and private keys, and the bit-length of the public key. This data will be added to the user database eventually.

3.3 Primitive RSA Encryption and Decryption

Now that the keys have been generated, encryption and decryption is merely an exponentiation in modulo n as described in Section 2.2.2. The functions can hence be written as:

```
def enc(m, e, n):
    c = pow(m, e, n)
    return c
```

Listing 11: RSA Encryption

and,

```
def dec(c, d, n):
    m = pow(c, d, n)
    return m
```

Listing 12: Primitive RSA Decryption

The current implementation can be improved by using the Chinese Remainder Algorithm to speed up the decryption process.

3.3.1 Chinese Remainder Algorithm for Fast Decryption

The Chinese Remainder Algorithm is a generalisation of the method used to solve problems such as:

> There are certain things whose number is unknown. If we count them by threes, we have two left over; by fives, we have three left over; and by sevens, two are left over. How many things are there?

We use the following optimisation to decrypt, based on the general Chinese Remainder Theorem. The following values are precomputed and stored as part of the private key:

1. p and q: the primes from generation.
2. $d_P = d \pmod{p-1}$

3. $d_Q = d \pmod{p-1}$

4. $q_{inv} = q^{-1} \pmod{p}$

We then use these four values to calculate the $c^d \pmod{pq}$ thus:

1. $m_1 = c^{d_P} \pmod{p}$

2. $m_2 = c^{d_Q} \pmod{q}$

3. If $m_1 > m_2$:
$$h = q_{inv}(m_1 - m_2) \pmod{p}$$

4. Else:
$$h = q_{inv}[(m_1 + \left\lceil \frac{q}{p} \right\rceil p) - m_2] \pmod{p}$$

5. $m = m_2 + hq \pmod{pq}$

This algorithm derives its efficiency from calculating two small modular exponentiations rather than one large one. The Python implementation is as follows:

```
from math import ceil
def cra(c, p, q, d_p, d_q, q_inv):
    '''Chinese Remainder Algorithm'''
    m_1 = pow(c, d_p, p)
    m_2 = pow(c, d_q, q)
    if m_1 >= m_2:
        h = q_inv*(m_1-m_2) % p
    else:
        h = q_inv*((m_1 + (ceil(q/p))*p) - m_2) % p
    return pow((m_2 + h*q), 1, p*q)
```

Listing 13: Chinese Remainder Algorithm for Fast Decryption

The decryption function then changes to:

```
def dec(c, p, q, d_p, d_q, q_inv):
    m = mf.cra(c, p, q, d_p, d_q, q_inv)
    return m
```

Listing 14: Decryption Using CRA

3.4 ASCII to Integer via Octet Strings and Back

Given all the previous algorithms and their implementations, we are ready to convert an integer message m to ciphertext c and back. However, one typically wishes to encrypt text, not an integer. We hence define a few primitives to convert text to a long integer in a predictable and reversible way.

3.4.1 PT2OS

The first step is to convert individual letters to integers. Luckily for us, this is exactly how letters are represented in computers, using ASCII octet strings. We hence write a Python function to convert a string into a list of ASCII values (PlainText 2 (to) Octet String).

```
def nochunk_PT2OS(text):
    return [ord(i) for i in text]
```

Listing 15: Plain Text to Octet String

3.4.2 OS2IP

We must now convert the resulting octet string to an integer, which is done using the Octet String 2 (to) Integer Primitive. The PKCS#1 standard defines OS2IP as follows [5]:

Input: X, octet string to be converted

Output: x corresponding nonnegative integer

Steps:

1. Let $X_1\ X_2 \ldots X_n$ be the octets of X from first to last, and let X_i have the integer value x_{l-i} for $1 \leq i \leq l$.
2. Let $x = x_{l-1}256^{l-1} + x_{l-2}256^{l-2} + ... + x_1 256 + x_0$.
3. Output x.

The Python implementation is trivial:

```
def OS2IP(X):
    l = len(X)
    i = 1
    sum_ = 0
    for X_i in X:
        sum_ = X_i*(256**(l-i)) + sum_
        i += 1
    return sum_
```

Listing 16: OS2IP

Note that this is effectively a conversion to base 256. This integer will then be encrypted.

3.4.3 I2OSP

After decryption, we receive the base-256 integer again, and we must now restore it to an octet string. Or, as the standard states (Integer 2 (to) Octet String Primitive):

Input: x, nonnegative integer to be converted l, intended length of the resulting octet string

Output: X, corresponding octet string of length l

Error: "Integer too large"

Steps:

1. If $x \geq 256^l$, output "integer too large" and stop.
2. Write the integer x in its unique l-digit representation in base 256:

$$x = x_{l-1}256^{l-1} + x_{l-2}256^{l-2} + \ldots + x_1 256 + x_0$$

where $0 \leq x_i < 256$ (note that one or more leading digits will be zero if x is less than 256^{l-1}).

3. Let the octet X_i have the integer value x_{l-i} for $1 \leq i \leq l$. Output the octet string $X = X_1 X_2 \ldots X_l$.

The Python implementation of this is again simple, we merely divide x and collect successive remainders, recreating the octet string list.

```
def I2OSP(x, l):
    if x >= 256**l:
        raise ValueError("Int too large.")
    i = 0
```

```
    X = []
    while i != (l-1):
        r = x % 256
        X.append(r)
        x = x // 256
        i += 1
    X.append(x)
    return X[::-1]
```

Listing 17: I2OSP

3.4.4 OS2PT

We now are left with converting the octet string back to plaintext (Octet String 2 (to) PlainText), which is another trivial operation in Python:

```
def OS2PT(stream):
    return ''.join(chr(i) for i in stream)
```

Listing 18: Octet String to Plain Text

3.4.5 Putting It Together

The order in which we apply the primitives is hence:

1. PT2OS
2. OS2IP
3. RSA Encrypt
4. RSA Decrypt (CRA)
5. I2OSP
6. OS2PT

3.5 File Chunking

We can hence encrypt any plaintext whose integral value is less than the number of bits of n.[15] However, very often, the requirement is to encrypt strings of far greater length called 'files'. To encrypt a file of arbitrary length, it must be split or 'chunked' into multiple parts, each individually encrypted, each individually decrypted, then woven back together.

To fulfill such a requirement, we first note that the file must be broken into equally sized chunks. However, a file may not necessarily be perfectly divisible into chunks, that is, there will be a remainder block which has insufficient bits. Hence, we pad the last block with a newline and ASCII nulls, which will be removed later. This prevents corruption of the last block. To do so, we rewrite PT2OS in the following way:

```
def PT2OS(text, l):
    ret = [ord(i) for i in text]
    l_2 = len(ret)
    if l_2 < l:
        newline = [10]
        spaces = [0] * (l - l_2 -1)
        ret.extend(newline)
        ret.extend(spaces)
    return ret
```

[15] To do this, select the `string` option when running the main program.

Listing 19: Chunking PT2OS

Note that 10 is an ASCII newline, which will be later removed. Now that the file is 'chunkable', we write a generator function which returns chunks of a specified size that we can later operate on.

```
def read_in_chunks(file_object, chunk_size):
    '''Lazy function (generator) to read a file piece by piece.'''
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break
        yield data
```

Listing 20: Read File in Chunks Generator

And now, all that's left to do is remove the nulls from the last chunk after decryption.[16]

```
def kill_nulls(filename):
    f = open(filename, 'r')
    lines = f.readlines()
    f.close()
    lines.pop()
    f = open(filename, 'w')
    for line in lines:
        f.write(line)
    f.close()
    f = open(filename, 'r')
    string = f.read()
    f.close()
    string = string[:-1]
    f = open(filename, 'w')
    f.write(string)
    f.close()
```

Listing 21: Remove Extraneous Nulls

This function first reads the decrypted file line by line into a list, then deletes the last element of the list, and writes the new list back to the file. It then reads the file into a string and strips the newline from the end of the file, restoring the file to its pre-encryption state.

3.6 User Data Pandas DataFrame

This is the simpler part of our operation, we merely need to write functions to create a DataFrame to act as a database file, and functions to add, remove, and search for data. This is the DataFrame which maintains a list of users, their passwords, and their keys.

DataFrame Creation and Storage The DataFrame, once created, must be persistent across sessions. For this reason we save the DataFrame as a Pickle (`.pkl`) file.

```
import pandas as pd

def save_table(usr_data):
    usr_data.to_pickle("usr_data.pkl")
```

[16] While the Pythonic way to write this function would be using `with open(filename):`, we use `f.open()` and `f.close()` for the sake of explicitness. They are operationally the same.

Listing 22: Pickle DataFrame

to create the DataFrame and,

```
def read_table():
    return pd.read_pickle("usr_data.pkl")
```

Listing 23: Read DataFrame Pickle

to read the created DataFrame.

Additionally, at level 0 indent, we must create the DataFrame if `usr_data.pkl` doesn't exist.

```
from os import path
if not path.exists("usr_data.pkl"):
    usr_data = pd.DataFrame(columns = ['username', 'password', 'p', 'q', 'n', 'e',
    ↪  'd_p', 'd_q', 'q_inv', 'bits'])
    usr_data.loc[0] = ['root', 0, 0, 0, 0, 0, 0, 0, 0, 0]
    save_table(usr_data)
```

Listing 24: Create DataFrame

Note that the DataFrame stores the Chinese Remainder Algorithm parameters instead of the private key d.

Add And Remove Data

```
def add_data(data):
    usr_data = read_table()
    usr_data.loc[usr_data.index.max() + 1, :] = data
    save_table(usr_data)
```

Listing 25: Add Data to DataFrame

```
def drop_row(key):
    usr_data = read_table()
    index_ = get_index(key, 'user')
    usr_data.drop(index=index_, inplace = True)
    save_table(usr_data)
```

Listing 26: Remove Data from DataFrame

Search For Data

```
def search_column(column, value):
    usr_data = read_table()
    result = (usr_data[usr_data[column] == value])
    if result.empty:
        return False
    else:
        return True
```

Listing 27: Search DataFrame

Getting Fields And Indexes

```
def get_field(key, column):
    usr_data = read_table()
    x = usr_data.username[usr_data.username.str.contains('|'.join(key.split('
    ↪  ')))]
    index = x.index[0]
    return usr_data.loc[index][column]
```

Listing 28: Retrieve Field from DataFrame

Note that the actual integral value of the index is the first index of the Pandas Series datatype that is assigned to x.

```
def get_index(key, column):
    usr_data = read_table()
    x = usr_data.username[usr_data.username.str.contains('|'.join(key.split('
    ↪  ')))]
    index = x.index[0]
    return index
```

Listing 29: Retrieve Index from DataFrame

3.7 Interaction Functions

These are the functions that 'talk' to the user and ask for inputs. They are fairly self explanatory. While the ideal way to take inputs is a single command line which takes arguments, these functions are written for the sake of providing a user-friendly front end to perhaps later implement as a GUI.

3.7.1 Users and Passwords

First, prompt the user for account creation or signup:

```
dimport usr_data as ud
import data_conversion_primitives as dcp
from getpass import getpass
import os.path

def login_or_signup():
    while True:
        try:
            state = int(input("[1] Create account\n[2] Sign in\n> "))
            if state not in (1, 2):
                print("ERR: Enter a valid choice.")
            else:
                return state
        except (TypeError, ValueError):
            print("ERR: Enter a valid choice.")
```

Listing 30: Login/Signup Function

If the user wants to signup:

```
def signup_core():
    while True:
        try:
```

```python
            user = str(input("Enter a username:\n> "))
            if ud.search_column('username', user):
                print("Username taken, try another.")
            else:
                break
        except (TypeError, ValueError):
            print("Enter a valid username.")
    while True:
        try:
            passphrase_a = getpass(prompt = "Passphrase: ")
            passphrase_b = getpass(prompt = "Repeat passphrase: ")
            if passphrase_a == passphrase_b:
                passphrase_a = dcp.OS2IP(dcp.nochunk_PT2OS(passphrase_a))
                passphrase_a = pow(passphrase_a, 65537,
                ↪  206013970136021274755909796996044923643)
                break
            else:
                print("Passphrases do not match. Try again.")
        except (ValueError, TypeError):
            print("Enter a valid passphrase.")
    return [user, passphrase_a]

def signup_pqned(data, pqned):
    data.extend(pqned)
    ud.add_data(data)
```

Listing 31: Signup Function

The first function returns a list of the user's selected username and password, and the second appends the key data to it after generation. This is then added to the DataFrame and pickled. Note that passwords are **not** stored as plaintext, instead, they are actually RSA encrypted using a static (n, e). Note that this is done as an alternative to hashing the password.

If the user chooses to login, cross check data with the DataFrame and grant access.

```python
def login():
    while True:
        try:
            user = str(input("Enter your username:\n> "))
            if ud.search_column('username', user):
                break
            else:
                print("Username doesn't exist.")
        except (TypeError, ValueError):
            print("Enter a valid username.")
    while True:
        try:
            passphrase = getpass(prompt = "Passphrase: ")
            passphrase = dcp.OS2IP(dcp.nochunk_PT2OS(passphrase))
            passphrase = pow(passphrase, 65537,
            ↪  206013970136021274755909796996044923643)
            if passphrase == ud.get_field(user, 'password'):
                break
            else:
                print("Incorrect passphrase. Try again.")
        except (TypeError, ValueError):
            print("Enter a valid passphrase.")
    return user
```

Listing 32: Login Function

This concludes the login procedure.

3.7.2 Get Functions

To figure out what the user wants to do:

```
def get_operation():
    while True:
        try:
            location = int(input("\n[1] File\n[2] String\n[3] Add path\n[4] Delete
            ↪  account\n[5] Exit\n> "))
            if location not in (1, 2, 3, 4, 5):
                print("ERR: Enter a valid choice.")
            else:
                return location
        except (TypeError, ValueError):
            print("ERR: Enter a valid choice.")
```

Listing 33: Main Loop Operation Selector

If the user wants to operate on files, get the input file and the output file:

```
def get_input_file():
    while True:
        filename = str(input("Input file?\n> "))
        try:
            with open(filename):
                return filename
        except FileNotFoundError:
            print("Incorrect file or path.")

def get_output_file():
    filename = str(input("Output file?\n> "))
    return filename
```

Listing 34: File Management Functions

Choose whether to encrypt or decrypt:

```
def encrpyt_or_decrypt():
    while True:
        try:
            state = int(input("[1] Encrypt file\n[2] Decrypt file\n> "))
            if state not in (1, 2):
                print("ERR: Enter a valid choice.")
            else:
                return state
        except (TypeError, ValueError):
            print("ERR: Enter a valid choice.")
```

Listing 35: Encryption/Decryption Choice Function

Get encryption and decryption data:

```python
def get_encrypt_list(user):
    bits = ud.get_field(user, 'bits')
    e = ud.get_field(user, 'e')
    n = ud.get_field(user, 'n')
    return [bits, e, n]
def get_decrypt_list(user):
    bits = ud.get_field(user, 'bits')
    p = ud.get_field(user, 'p')
    q = ud.get_field(user, 'q')
    d_p = ud.get_field(user, 'd_p')
    d_q = ud.get_field(user, 'd_q')
    q_inv = ud.get_field(user, 'q_inv')
    return [bits, p, q, d_p, d_q, q_inv]
```

Listing 36: Key Information Functions

When encrypting a file, select which user's public key to use:

```python
def get_reciever():
    while True:
        try:
            user = str(input("Enter recipient username:\n> "))
            if ud.search_column('username', user):
                break
            else:
                print("Username doesn't exist.")
        except (TypeError, ValueError):
            print("Enter a valid username.")
    return user
```

Listing 37: Receiver Information Function

If the user wants to add a path to `sys.path`:

```python
def get_path():
    while True:
        path = str(input("Path?\n> "))
        if os.path.isdir(path):
            return path
        else:
            print("Not a valid path.")
```

Listing 38: Path Validity Checker Function

Get the number of bits:

```python
def get_bits():
    print("Generating keys (one-time procedure).")
    while True:
        try:
            bits = int(input("Key length (bits)?\n> "))
            if bits >= 128 and mf.poweroftwocheck(bits):
                print("Key length okay...generating keys.")
                return bits
            else:
                print("ERR: Key length must be a power of two and greater than or
                ↪  equal to 128.")
```

```python
        except (TypeError, ValueError):
            print("ERR: Enter a valid key length.")
```

Listing 39: Interactive Bit-Length Finder

Since we need to check if the key input is a power of two, we write the following function:

```python
def poweroftwocheck(n):
    if (n == 0):
        return False
    while (n != 1):
        if (n % 2 != 0):
            return False
        n = n // 2
    return True
```

Listing 40: Power of Two Checker

3.7.3 Miscellaneous Functions

If the user wants to delete their account:

```python
def del_user():
    user = login()
    ud.drop_row(user)
```

Listing 41: User Deletion Interactive

3.8 Putting It All Together

Having created a library of functions, all that is left is assembling these correctly into a working system. This is done as follows.

3.8.1 Imported Modules

```python
import rsa_primitive as rsa
import math_functions as mf
import data_conversion_primitives as dcp
import interaction_functions as intfunc
from progress.bar import IncrementalBar
import sys
```

Listing 42: Main Module Imports

`progress` is used to draw the progress bars for encryption and decryption. `sys` will be used to append paths.

3.8.2 Signup And Login

When the program is run, we call the login-related interaction functions.

```python
state = intfunc.login_or_signup()
if state == 1:
    data = intfunc.signup_core()
    bits = rsa.get_bits()
    pqned = rsa.gen_keys(bits)
```

```
    print("Keys successfully generated.")
    intfunc.signup_pqned(data, pqned)

user = intfunc.login()
```

Listing 43: Main Signup and Login

The key generation process is done once for each user and stored in the database. After the signup is completed, the user is asked to login to continue using the program. In case the user exists, go directly to the login screen.

3.8.3 Main Loop

All the following code is wrapped in a forever loop. First, get the operation:

```
operation = intfunc.get_operation()
```

Listing 44: Main Operation Selection

File Encryption and Decryption Choose whether to encrypt or decrypt a file.

```
if operation == 1:
    enc_or_dec = intfunc.encrpyt_or_decrypt()
```

Listing 45: Main File Operation Selection

If encryption:

```
if enc_or_dec == 1
    recpt = intfunc.get_reciever()
    input_file = intfunc.get_input_file()
    output_file = intfunc.get_output_file()
    enc_data = intfunc.get_encrypt_list(recpt)
    bits = enc_data[0]
    e = enc_data[1]
    n = enc_data[2]
    chunk_size = (bits//8) - 1
    open(output_file, 'w+').close()
    list_of_chunks = []
    fobj = open(input_file, 'r')
    l = sum(1 for _ in (dcp.read_in_chunks(fobj, chunk_size)))
    with open(input_file) as fin:
        print("All okay.")
        for data in
        IncrementalBar('Encrypting...', max=l).iter(dcp.read_in_chunks(fin,
         ↪  chunk_size)):
            data = dcp.PT2OS(data, chunk_size)
            data = dcp.OS2IP(data)
            data = rsa.enc(data, e, n)
            list_of_chunks.append(int(data))
    with open(output_file, 'a') as fout:
        print(list_of_chunks, end='', file=fout)
    print("Encrypted. Output written to", fout.name)
```

Listing 46: Main Encryption Algorithm

The user first selects which file to encrypt (input) and which file to write to encrypted output to. Note that these cannot be the same file. When encrypting a file, the public key of the recipient must be retrieved from the database. The username of the recipient is requested, and the recipient's public key retrieved from the database and read into variables e and n. The output file is then restored to an empty file state. The input file is read chunk by chunk, (the chunk size being defined as one less than the byte length of n) and each chunk encrypted. The last chunk has nulls added by PT2OS. Each chunk is written as an element of a list, which is written directly to the file. The encrypted file is hence the output of printing a list to the output file.

Note that the `bar` variable creates the progress bar. For more details, see the documentation of `progress`.

Else if decryption:

```
else:
    input_file = intfunc.get_input_file()
    output_file = intfunc.get_output_file()
    dec_data = intfunc.get_decrypt_list(user)
    bits = dec_data[0]
    p = dec_data[1]
    q = dec_data[2]
    d_p = dec_data[3]
    d_q = dec_data[4]
    q_inv = dec_data[5]
    chunk_size = (bits//8) - 1
    open(output_file, 'w+').close()
    with open(input_file) as fin:
        list_of_chunks = fin.read()
    list_of_chunks = list_of_chunks.strip('][').split(', ')
    print("All okay.")
    f =  open(output_file, 'a')
    bar = IncrementalBar('Decrypting...', max=len(list_of_chunks))
    for data in list_of_chunks:
        data = rsa.dec(int(data), p, q, d_p, d_q, q_inv)
        data = dcp.I2OSP(data, chunk_size)
        data = dcp.OS2PT(data)
        print(data, end='', file=f)
        bar.next()
    f.close()
    dcp.kill_nulls(output_file)
    bar.finish()
    print("Decrypted. Output written to", f.name)
```

Listing 47: Main Decryption Algorithm

When decrypting, first retrieve the private-key information of the user from the database. Ask for the input and output files, and clear the output file. Then, read the input file into a list. Decrypt each element in the list, add it to the output file, and close the output file. Then, kill the nulls in the last chunk and exit.

String Encryption To encrypt a string using the user's own public key and decrypt it:

```
elif operation == 2:
    enc_data = intfunc.get_encrypt_list(user)
    bits = enc_data[0]
    e = enc_data[1]
    n = enc_data[2]
    chunk_size = (bits//8) - 1
    dec_data = intfunc.get_decrypt_list(user)
```

```
    bits = dec_data[0]
    p = dec_data[1]
    q = dec_data[2]
    d_p = dec_data[3]
    d_q = dec_data[4]
    q_inv = dec_data[5]

    m = str(input("Enter a string: "))
    m = dcp.pt2os(m, chunk_size)
    m = dcp.os2ip(m)

    c = rsa.enc(m, e, n)
    print("Encrypted string is:", m)
    m = rsa.dec(c, p, q, d_p, d_q, q_inv)

    m = dcp.i2osp(m, chunk_size)
    m = dcp.os2pt(m)
    print("Decrypted string is:", m)
```

Listing 48: Main String Encryption and Decryption

String encryption merely demonstrates basic encryption and decryption using the user's public and private keys. The algorithm is self-explanatory.

Appending Path To append a path for file encryption:

```
elif operation == 3:
    path = intfunc.get_path()
    sys.path.append(path)
    print(sys.path[-1], "successfully appended to sys.path.")
```

Listing 49: Main Append Path

The program can only access files to encrypt and decrypt which are in the `sys.path` list. To add a path to that list, append it.

Account Deletion When the user wants to delete their account, confirm if they want to delete, ask for a login, then delete.

```
elif operation == 4:
    while True:
        try:
            sure = str(input("[ACCOUNT DELETION]: Proceed? Irreversible. (y or n):
            ↪  "))
            if sure not in ('y', 'n'):
                print("[ACCOUNT DELETION]: Please answer y or n.")
            else:
                if sure == "y":
                    intfunc.del_user()
                    print("All okay. Account successfully deleted.")
                    exit()
                else:
                    break
        except (TypeError, ValueError):
            print("[ACCOUNT DELETION]: Please answer y or n.")
```

Listing 50: Main Account Deletion

3.8.4 Exit

To exit:

```
else:
    exit()
```

Listing 51: Main Exit

4 Results

The results in this section are sandbox tested on an Anaconda 3 install of Python on Microsoft Windows 7 and 10, and on the standard Python 3 install on modern Linux systems.

4.1 File Encryption Demonstration

```
aditya@ganga:~$ python3 main.py
[1] Create account
[2] Sign in
>
```

Create two accounts: one for the reciever and one for the transmitter. Also, test the passphrases should match check.

```
[1] Create account
[2] Sign in
> 1
Enter a username:
> Rx
Passphrase:
Repeat passphrase:
Passphrases do not match. Try again.
Passphrase:
Repeat passphrase:
Generating keys (one-time procedure).
Key length (bits)?
> 2048
Key length okay...generating keys.
Keys successfully generated.
```

Create transmitter account:

```
[1] Create account
[2] Sign in
> 1
Enter a username:
> Tx
Passphrase:
Repeat passphrase:
Generating keys (one-time procedure).
Key length (bits)?
> 2048
Key length okay...generating keys.
Keys successfully generated.
```

Login as `Tx`, and encrypt a file using `Rx`'s public key. The standard test file provided with the program is `file.txt`, a 3.7MB plaintext file that contains all Sherlock Holmes stories, copyright-free. Note that 3.7MB of plaintext is on the larger side for plaintext files.

```
[1] Create account
[2] Sign in
> 2
Enter your username:
> Tx
Passphrase:

[1] File
[2] String
```

```
[3] Add path
[4] Delete account
[5] Exit
> 1
[1] Encrypt file
[2] Decrypt file
> 1
Enter recipient username:
> Rx
Input file?
> file.txt
Output file?
> encrypted_file.txt
All okay.
Encrypting... |--------------------------------| 15168/15168
Encrypted. Output written to encrypted_file.txt

[1] File
[2] String
[3] Add path
[4] Delete account
[5] Exit
> 5
aditya@ganga:~$
```

Now, login as `Rx` and decrypt `encrypted_file.txt`. Also, test if the input file validity checker works. Note that decryption takes time on slower machines, so be patient.

```
[1] Create account
[2] Sign in
> 2
Enter your username:
> Rx
Passphrase:

[1] File
[2] String
[3] Add path
[4] Delete account
[5] Exit
> 1
[1] Encrypt file
[2] Decrypt file
> 2
Input file?
> not_a_real_file
Incorrect file or path.
Input file?
> encrypted_file.txt
Output file?
> decrypted_file.txt
All okay.
Decrypting... |--------------------------------| 15168/15168
Decrypted. Output written to decrypted_file.txt

[1] File
[2] String
[3] Add path
```

```
[4] Delete account
[5] Exit
> 5
aditya@ganga:~$
```

Now, take a `diff` of the original `file.txt` and the decrypted `decrypted_file.txt`.

```
aditya@ganga:~$ diff -s file.txt decrypted_file.txt
Files file.txt and decrypted_file.txt are identical
```

Everything is working as expected, and there is no file corruption during transmission.

4.2 String Encryption Demonstration

Encrypt a string as user `Rx`:

```
[1] Create account
[2] Sign in
> 2
Enter your username:
> Rx
Passphrase:

[1] File
[2] String
[3] Add path
[4] Delete account
[5] Exit
> 2
Enter a string: Vi has two modes: beep repeatedly and break everything.
Encrypted string is: 426106800080092545292628332367505162029833888494867760168
2440247683671895829464017836521322062591784104878381529237227270526846099329333
0920944134397163136618631774465159752315562746213131147556459241713843242932955
3847737622580247545187072936075922268115841110862574335229638209510946539872744
3888154885873934224921389787771729051747115616256273647230280936829219170859955
3418784527464525472498025325225055040372144033162783330423773773904293890152622
2418618780950612539850693488049273682512103892577942655140660962807774155933099
3656699892709300234224546156706379640061258308995783552591443692755997095331666
74813558784
Decrypted string is: Vi has two modes: beep repeatedly and break everything.
```

4.3 Account Deletion Demonstration

Delete account `Tx`.

```
[1] Create account
[2] Sign in
> 2
Enter your username:
> Tx
Passphrase:

[1] File
[2] String
[3] Add path
[4] Delete account
[5] Exit
> 4
[ACCOUNT DELETION]: Proceed? Irreversible. (y or n): maybe
```

```
[ACCOUNT DELETION]: Please answer y or n.
[ACCOUNT DELETION]: Proceed? Irreversible. (y or n): y
Enter your username:
> Tx
Passphrase:
All okay. Account successfully deleted.
```

Try to login with the deleted account:

```
[1] Create account
[2] Sign in
> 2
Enter your username:
> Tx
Username doesn't exist.
Enter your username:
>
```

That concludes our demonstrations. To check the path addition and exit functions is trivial and hence omitted.

4.4 Encryption Time Test

To test the time taken for encryption, we remove all user inputs and measure the time for purely the encryption of `file.txt`. We run:

```
aditya@ganga:~$ for i in {1..10}; do time python3 enc_test.py; done
```

The encryption tests take times between 10.2 and 10.8 seconds, as shown in the graph.

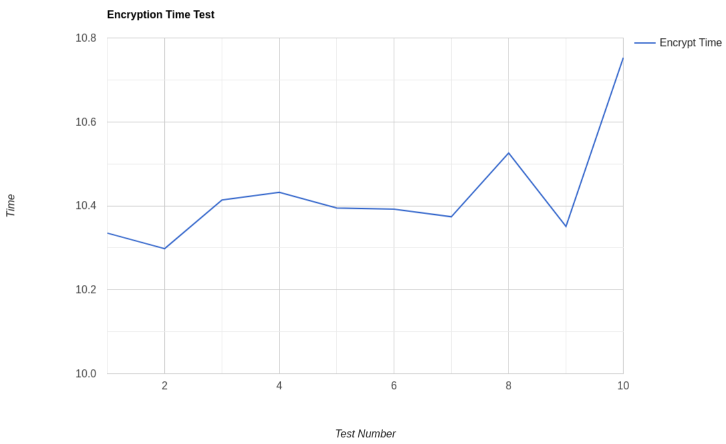

Figure 4: Encryption Time Graph

4.5 Decryption Time Test

Decryption takes longer than encryption, around 2 minutes 38 seconds on average. While this value is a lot more than encryption, using the Chinese Remainder Algorithm speeds up the process greatly. Using only Python's `pow()` function to decrypt took 8m19.523s.

```
aditya@ganga:~$ for i in {1..5}; do time python3 dec_test.py; done
```

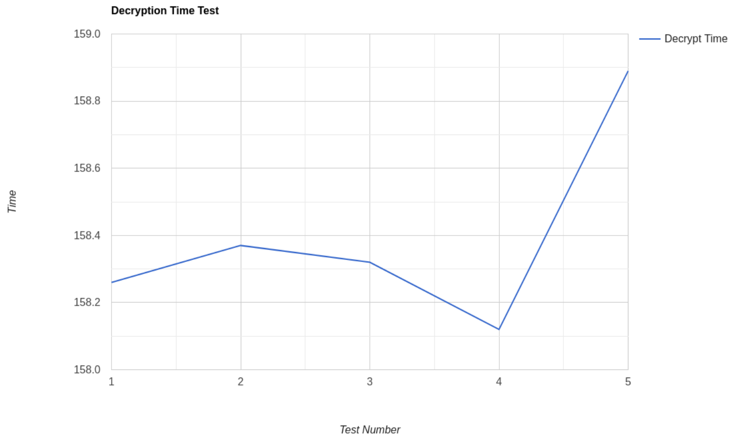

Figure 5: Decryption Time Graph

We have now completely tested the system created. In case of any discrepancies or improvements, write an email to aditya.v.nebhrajani@gmail.com.

5 Author's Note

5.1 Public Key Cryptography Standard

This tool implements the greater part of the PKCS (Public Key Cryptography Standard), with the notable exception of OEAP padding. This part of the standard wasn't implemented due to time constraints and other academic commitments. Padding is considered vital for short plaintexts, such as those the string encryption implemented. However, for the length of plaintexts this tool expects encryption for, padding is not so much a requirement as yet another compute-intensive task. Given more time, I hope to eventually implement OAEP as well, complying with the standard specified by PKCS.

5.2 Tools

I am deeply indebted to the Python community and the creator of `progress`. While Python is not very fast and too high level for an application like this one, the fact that it could do it this well, acting as both front end and back end (Pandas), makes it an excellent and powerful tool. I hope to eventually implement a much better encryption/decryption tool in C to improve performance. [17]

The complete list of tools is:

- Foxconn Core i7 NanoPC.
- Linux Mint 19.3 XFCE 64-bit.
- GNU Emacs 25.2.2 (x86_64-pc-linux-gnu, GTK+ Version 3.22.21), modified by Debian.
- Python 3.6.9 from Debian repositories.
- LaTeX for documentation. pdfTeX 3.14159265-2.6-1.40.18, from TeX Live 2017/Debian.
- LaTeX package `minted` for code highlighting.
- Emacs `org-mode` 9.3.6, GNU ELPA.
- GNU CLISP 2.49.60+ for some scripts.
- VirtualBox 5.2.42 for sandbox.
- Evince 2.4.4.

[17] Personally, I prefer statically typed, compiled languages with delimiters. In case the source code of this project can be more Pythonic, please do let me know. There are many places where I opted to write functions that could've been imported from various modules, since I believe the only way to truly understand that code one writes is building it from scratch. Also, it increases the translatablity of the code to languages with different libraries.

6 Source Code by File

This section directly highlights and prints from the /src directory. Other code snippets in this document are static. If there is a change in source code, expect it to show up here first and elsewhere in this document later.

```
ip_project/
|-- docs
|-- README.md
|-- src
```

6.1 prime_generator.py

```python
import first_primes as fp
import secrets

def gen_random(l):
    '''Generate a random number l bits long.'''
    randgenerator = secrets.SystemRandom()
    return randgenerator.randrange(2**(l-1)+1, 2**l - 1)

def low_level_checker(l):
    '''Check that the random number isn't divisible by the first few primes.'''
    while True:
        x = gen_random(l)
        for divisor in fp.first_primes:
            if x % divisor == 0 and divisor**2 <= x:
                break
        else: return x

def miller_rabin_checker(mrc):
    '''Run 40 iterations of the Miller-Rabin Primality Test.'''
    randgenerator = secrets.SystemRandom()
    max_divisions_by_two = 0
    y = mrc-1
    while y % 2 == 0:
        y >>= 1
        max_divisions_by_two += 1
    assert(2**max_divisions_by_two * y == mrc-1)

    def trial_composite(round_tester):
        if pow(round_tester, y, mrc) == 1:
            return False
        for i in range(max_divisions_by_two):
            if pow(round_tester, 2**i * y, mrc) == mrc-1:
                return False
        return True
    number_of_rabin_trials = 40
    for i in range(number_of_rabin_trials):
        round_tester = randgenerator.randrange(2, mrc)
        if trial_composite(round_tester):
            return False
    return True

def driver(l):
    while True:
        prime_candidate = low_level_checker(l)
        if not miller_rabin_checker(prime_candidate):
```

```
            continue
        else:
            return prime_candidate
```

Listing 52: File: Prime Generator

6.2 math_functions.py

```
import sys
sys.setrecursionlimit(10**6)
from math import ceil

def poweroftwocheck(n):
    if (n == 0):
        return False
    while (n != 1):
        if (n % 2 != 0):
            return False
        n = n // 2
    return True

def gcd(a,b):
    if a == 0:
        return b
    return gcd(b % a, a)
def lcm(a,b):
    return (a*b) // gcd(a,b)

def eea(a, b):
    '''Extended Euclidean algorithm'''
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = eea(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    '''Find modular multiplicative inverse using EEA'''
    g, x, y = eea(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist.')
    else:
        return x % m
def cra(c, p, q, d_p, d_q, q_inv):
    '''Chinese Remainder Algorithm'''
    m_1 = pow(c, d_p, p)
    m_2 = pow(c, d_q, q)
    if m_1 >= m_2:
        h = q_inv*(m_1-m_2) % p
    else:
        h = q_inv*((m_1 + (ceil(q/p))*p) - m_2) % p
    return pow((m_2 + h*q), 1, p*q)
```

Listing 53: File: Math Functions

6.3 rsa_primitive.py

```
import prime_generator as pg
import math_functions as mf

def get_bits():
    print("Generating keys (one-time procedure).")
    while True:
        try:
            bits = int(input("Key length (bits)?\n> "))
            if bits >= 128 and mf.poweroftwocheck(bits):
                print("Key length okay...generating keys.")
                return bits
            else:
                print("ERR: Key length must be a power of two and greater than or
                ↪  equal to 128.")
        except (TypeError, ValueError):
            print("ERR: Enter a valid key length.")

def gen_keys(bits):
    p = pg.driver(bits//2)
    q = pg.driver(bits//2)
    n = p*q
    lambda_ = mf.lcm((p-1), (q-1))
    e = 65537
    d = mf.modinv(e, lambda_)
    d_p = d % (p-1)
    d_q = d % (q-1)
    q_inv = mf.modinv(q, p)
    return [p, q, n, e, d_p, d_q, q_inv, bits]

def enc(m, e, n):
    c = pow(m, e, n)
    return c

def dec(c, p, q, d_p, d_q, q_inv):
    m = mf.cra(c, p, q, d_p, d_q, q_inv)
    return m
```

Listing 54: File: RSA Primitive

6.4 data_conversion_primitives.py

```
def nochunk_PT2OS(text):
    return [ord(i) for i in text]

def PT2OS(text, l):
    ret = [ord(i) for i in text]
    l_2 = len(ret)
    if l_2 < l:
        newline = [10]
        spaces = [0] * (l - l_2 -1)
        ret.extend(newline)
        ret.extend(spaces)
    return ret
```

```
def OS2PT(stream):
    return ''.join(chr(i) for i in stream)

def OS2IP(X):
    l = len(X)
    i = 1
    sum_ = 0
    for X_i in X:
        sum_ = X_i*(256**(l-i)) + sum_
        i += 1
    return sum_

def I2OSP(x, l):
    if x >= 256**l:
        raise ValueError("Int too large.")
    i = 0
    X = []
    while i != (l-1):
        r = x % 256
        X.append(r)
        x = x // 256
        i += 1
    X.append(x)
    return X[::-1]
def read_in_chunks(file_object, chunk_size):
    '''Lazy function (generator) to read a file piece by piece.'''
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break
        yield data

def kill_nulls(filename):
    f = open(filename, 'r')
    lines = f.readlines()
    f.close()
    lines.pop()
    f = open(filename, 'w')
    for line in lines:
        f.write(line)
    f.close()
    f = open(filename, 'r')
    string = f.read()
    f.close()
    string = string[:-1]
    f = open(filename, 'w')
    f.write(string)
    f.close()
```

Listing 55: File: Data Conversion Primitives

6.5 interaction_functions.py

```
import usr_data as ud
import data_conversion_primitives as dcp
from getpass import getpass
```

```python
import os.path

def login_or_signup():
    while True:
        try:
            state = int(input("[1] Create account\n[2] Sign in\n> "))
            if state not in (1, 2):
                print("ERR: Enter a valid choice.")
            else:
                return state
        except (TypeError, ValueError):
            print("ERR: Enter a valid choice.")

def signup_core():
    while True:
        try:
            user = str(input("Enter a username:\n> "))
            if ud.search_column('username', user):
                print("Username taken, try another.")
            else:
                break
        except (TypeError, ValueError):
            print("Enter a valid username.")
    while True:
        try:
            passphrase_a = getpass(prompt = "Passphrase: ")
            passphrase_b = getpass(prompt = "Repeat passphrase: ")
            if passphrase_a == passphrase_b:
                passphrase_a = dcp.OS2IP(dcp.nochunk_PT2OS(passphrase_a))
                passphrase_a = pow(passphrase_a, 65537,
                ↪  206013970136021274755909796996044923643)
                break
            else:
                print("Passphrases do not match. Try again.")
        except (ValueError, TypeError):
            print("Enter a valid passphrase.")
    return [user, passphrase_a]

def signup_pqned(data, pqned):
    data.extend(pqned)
    ud.add_data(data)

def login():
    while True:
        try:
            user = str(input("Enter your username:\n> "))
            if ud.search_column('username', user):
                break
            else:
                print("Username doesn't exist.")
        except (TypeError, ValueError):
            print("Enter a valid username.")
    while True:
        try:
            passphrase = getpass(prompt = "Passphrase: ")
            passphrase = dcp.OS2IP(dcp.nochunk_PT2OS(passphrase))
```

```
            passphrase = pow(passphrase, 65537,
            ↪  206013970136021274755909796996044923643)
            if passphrase == ud.get_field(user, 'password'):
                break
            else:
                print("Incorrect passphrase. Try again.")
        except (TypeError, ValueError):
            print("Enter a valid passphrase.")
    return user

def get_operation():
    while True:
        try:
            location = int(input("\n[1] File\n[2] String\n[3] Add path\n[4] Delete
            ↪  account\n[5] Exit\n> "))
            if location not in (1, 2, 3, 4, 5):
                print("ERR: Enter a valid choice.")
            else:
                return location
        except (TypeError, ValueError):
            print("ERR: Enter a valid choice.")


def get_input_file():
    while True:
        filename = str(input("Input file?\n> "))
        try:
            with open(filename):
                return filename
        except FileNotFoundError:
            print("Incorrect file or path.")

def get_output_file():
    filename = str(input("Output file?\n> "))
    return filename
def encrpyt_or_decrypt():
    while True:
        try:
            state = int(input("[1] Encrypt file\n[2] Decrypt file\n> "))
            if state not in (1, 2):
                print("ERR: Enter a valid choice.")
            else:
                return state
        except (TypeError, ValueError):
            print("ERR: Enter a valid choice.")


def get_encrypt_list(user):
    bits = ud.get_field(user, 'bits')
    e = ud.get_field(user, 'e')
    n = ud.get_field(user, 'n')
    return [bits, e, n]
def get_decrypt_list(user):
    bits = ud.get_field(user, 'bits')
    p = ud.get_field(user, 'p')
    q = ud.get_field(user, 'q')
    d_p = ud.get_field(user, 'd_p')
```

```
    d_q = ud.get_field(user, 'd_q')
    q_inv = ud.get_field(user, 'q_inv')
    return [bits, p, q, d_p, d_q, q_inv]
def del_user():
    user = login()
    ud.drop_row(user)

def get_reciever():
    while True:
        try:
            user = str(input("Enter recipient username:\n> "))
            if ud.search_column('username', user):
                break
            else:
                print("Username doesn't exist.")
        except (TypeError, ValueError):
            print("Enter a valid username.")
    return user

def get_path():
    while True:
        path = str(input("Path?\n> "))
        if os.path.isdir(path):
            return path
        else:
            print("Not a valid path.")
```

Listing 56: File: Interaction Functions

6.6 usr_data.py

```
import pandas as pd
from os import path

def save_table(usr_data):
    usr_data.to_pickle("usr_data.pkl")

def read_table():
    return pd.read_pickle("usr_data.pkl")
if not path.exists("usr_data.pkl"):
    usr_data = pd.DataFrame(columns = ['username', 'password', 'p', 'q', 'n', 'e',
    ↪  'd_p', 'd_q', 'q_inv', 'bits'])
    usr_data.loc[0] = ['root', 0, 0, 0, 0, 0, 0, 0, 0, 0]
    save_table(usr_data)


def add_data(data):
    usr_data = read_table()
    usr_data.loc[usr_data.index.max() + 1, :] = data
    save_table(usr_data)

def search_column(column, value):
    usr_data = read_table()
    result = (usr_data[usr_data[column] == value])
    if result.empty:
        return False
```

```
    else:
        return True
def get_field(key, column):
    usr_data = read_table()
    x = usr_data.username[usr_data.username.str.contains('|'.join(key.split('
    ↪  ')))]
    index = x.index[0]
    return usr_data.loc[index][column]
def get_index(key, column):
    usr_data = read_table()
    x = usr_data.username[usr_data.username.str.contains('|'.join(key.split('
    ↪  ')))]
    index = x.index[0]
    return index

def drop_row(key):
    usr_data = read_table()
    index_ = get_index(key, 'user')
    usr_data.drop(index=index_, inplace = True)
    save_table(usr_data)
```

Listing 57: File: User Data Management

6.7 main.py

```
import rsa_primitive as rsa
import math_functions as mf
import data_conversion_primitives as dcp
import interaction_functions as intfunc
from progress.bar import IncrementalBar
import sys


state = intfunc.login_or_signup()
if state == 1:
    data = intfunc.signup_core()
    bits = rsa.get_bits()
    pqned = rsa.gen_keys(bits)
    print("Keys successfully generated.")
    intfunc.signup_pqned(data, pqned)

user = intfunc.login()

while True:
    operation = intfunc.get_operation()
    if operation == 1:
        enc_or_dec = intfunc.encrpyt_or_decrypt()
        if enc_or_dec == 1:
            recpt = intfunc.get_reciever()
            input_file = intfunc.get_input_file()
            output_file = intfunc.get_output_file()
            enc_data = intfunc.get_encrypt_list(recpt)
            bits = enc_data[0]
            e = enc_data[1]
            n = enc_data[2]
            chunk_size = (bits//8) - 1
```

```
        open(output_file, 'w+').close()
        list_of_chunks = []
        fobj = open(input_file, 'r')
        l = sum(1 for _ in (dcp.read_in_chunks(fobj, chunk_size)))
        with open(input_file) as fin:
            print("All okay.")
            for data in IncrementalBar('Encrypting...',
            ↪  max=l).iter(dcp.read_in_chunks(fin, chunk_size)):
                data = dcp.PT2OS(data, chunk_size)
                data = dcp.OS2IP(data)
                data = rsa.enc(data, e, n)
                list_of_chunks.append(int(data))
        with open(output_file, 'a') as fout:
            print(list_of_chunks, end='', file=fout)
        print("Encrypted. Output written to", fout.name)

    else:
        input_file = intfunc.get_input_file()
        output_file = intfunc.get_output_file()
        dec_data = intfunc.get_decrypt_list(user)
        bits = dec_data[0]
        p = dec_data[1]
        q = dec_data[2]
        d_p = dec_data[3]
        d_q = dec_data[4]
        q_inv = dec_data[5]
        chunk_size = (bits//8) - 1
        open(output_file, 'w+').close()
        with open(input_file) as fin:
            list_of_chunks = fin.read()
        list_of_chunks = list_of_chunks.strip('][').split(', ')
        print("All okay.")
        f =  open(output_file, 'a')
        bar = IncrementalBar('Decrypting...', max=len(list_of_chunks))
        for data in list_of_chunks:
            data = rsa.dec(int(data), p, q, d_p, d_q, q_inv)
            data = dcp.I2OSP(data, chunk_size)
            data = dcp.OS2PT(data)
            print(data, end='', file=f)
            bar.next()
        f.close()
        dcp.kill_nulls(output_file)
        bar.finish()
        print("Decrypted. Output written to", f.name)

elif operation == 2:
    enc_data = intfunc.get_encrypt_list(user)
    bits = enc_data[0]
    e = enc_data[1]
    n = enc_data[2]
    chunk_size = (bits//8) - 1
    dec_data = intfunc.get_decrypt_list(user)
    bits = dec_data[0]
    p = dec_data[1]
    q = dec_data[2]
    d_p = dec_data[3]
    d_q = dec_data[4]
```

```
        q_inv = dec_data[5]

        m = str(input("Enter a string: "))
        m = dcp.PT2OS(m, chunk_size)
        m = dcp.OS2IP(m)

        c = rsa.enc(m, e, n)
        print("Encrypted string is:", m)
        m = rsa.dec(c, p, q, d_p, d_q, q_inv)

        m = dcp.I2OSP(m, chunk_size)
        m = dcp.OS2PT(m)
        print("Decrypted string is:", m)

    elif operation == 3:
        path = intfunc.get_path()
        sys.path.append(path)
        print(sys.path[-1], "successfully appended to sys.path.")


    elif operation == 4:
        while True:
            try:
                sure = str(input("[ACCOUNT DELETION]: Proceed? Irreversible. (y or n): "))
                if sure not in ('y', 'n'):
                    print("[ACCOUNT DELETION]: Please answer y or n.")
                else:
                    if sure == "y":
                        intfunc.del_user()
                        print("All okay. Account successfully deleted.")
                        exit()
                    else:
                        break
            except (TypeError, ValueError):
                print("[ACCOUNT DELETION]: Please answer y or n.")

    else:
        exit()
```

Listing 58: File: Main

References

[1] David Adrian et al. "Imperfect forward secrecy: How Diffie-Hellman fails in practice". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 5–17.

[2] *Caesar cipher - Wikipedia*. URL: `https://en.wikipedia.org/wiki/Caesar_cipher`.

[3] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[4] *HackerNews Discussion on Encryption Tools*. URL: `https://news.ycombinator.com/item?id=19173326`.

[5] Jakob Jonsson and Burt Kaliski. *Public-key cryptography standards (PKCS)# 1: RSA cryptography specifications version 2.1*. Tech. rep. RFC 3447, February, 2003.

[6] *Let's talk about PAKE*. URL: `https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/`.

[7] *Modern Alternatives to PGP*. URL: `https://blog.gtank.cc/modern-alternatives-to-pgp/`.

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