Attacks on RSA

Shabab Ahmed

January 2019

1 Introduction

The advent of the internet has made communication simpler and more accessible to people all around the globe. However, the internet also raises important questions about data privacy and security. How do we know that the data we are sending over the Internet will not fall into the wrong hands? This is a particularly important question given that we use the Internet for day-to-day activities such as sending important e-mail conversations and credit card payments. It would be a disaster if one's credit card information fell into the wrong hands and even worse if national secrets could be easily intercepted and read. Thankfully, RSA takes care of that problem and ensures that our communication over the internet is secure. RSA, a cryptosystem named after L. Adleman, R. Rivest and A. Shamir, has become one of the most widely used methods of secure data transmission. Developed in 1977, it was one of the first public key cryptosystems. It is the fundamental part of various security protocols for communication over the internet.

RSA was developed in search of an asymmetric key cryptosystem. Symmetric key cryptography requires the use of the same key for encryption and decryption. The problem with asymmetric key cryptosystem is that that people need to communicate their key first in order to communicate. Asymmetric cryptosystems have a public key and a private key for a given person and the public key is known to everyone who wants to communicate with that person. The beauty of RSA is that it uses a one way function. These functions are easy to compute in one direction but extremely difficult to invert that computation.

1.1 RSA Basics:

So, how does RSA really work? To demonstrate this we will assume that Alice and Bob want to securely communicate with each other. For simplicity's sake, let us suppose that Bob wants to send a message to Alice. RSA requires the following[1]:

- Alice generating two large primes, p and q at random
- Alice computing $N = pq$ which will be the modulus and computing $(p-1)(q-1)$ to get $\phi(N)$ [1]
- Selecting a number e satisfying $1 < e < \phi(N)$ and $\gcd(e, \phi(N)) = 1$
- Computing $d = e^{-1} \bmod(\phi(N))$ [2] so that $ed \equiv 1 \bmod(\phi(N))$
- Forming $\langle N, e\rangle$ to be the public key and making it available to Bob
- Form $\langle N, d\rangle$ to be the private key and keep it secure
- Bob computing $C \equiv \mathrm{M}^e \bmod (N)$ for a message $M \in \mathbb{Z}*_N$ and sending C to Alice
- Upon receiving C, Alice computing $C^d \bmod N$, that is, $M^{ed} \bmod (N)$ to recover M

[1] $\phi(N)$ is Euler's totient function which is the number of integers strictly smaller than N that have no common divisors with N. It can also be thought of as the order of the multiplicative group $\mathbb{Z}_N^*$

[2] This computation of the inverse can be easily done using the Extended Euclidean algorithm. This algorithm for computing modular inverse is included in Appendix A.

This works because $ed \equiv 1 \bmod \phi(N)$ and so $ed = 1 + k(p-1)(q-1)$ for some $k \in \mathbb{Z}$. Thus, $C^d \equiv M.(M^{(p-1)(q-1)k}) \bmod N$. Euler's Theorem, derived from Fermat's Little Theorem gives us that $M^{(p-1)(q-1)} \equiv 1 \bmod N$ and modular arithmetic properties give us that $M^{(p-1)(q-1)k} \equiv 1 \bmod N$. Hence, $C^d \equiv M \bmod N$.

It is important to note that if Alice wanted to send a message to Bob, the roles would be reversed and Bob would have to go through the same process of creating his public and private key. The process mentioned above only describes sending messages to Alice and would not be sufficient to securely send messages to Bob.

The following example demonstrates the whole process:

Example: Suppose Alice uses $p = 7$ and $q = 11$ and hence, $N = 77$. We will further assume that Alice picks $e = 13$ since 13 is less than $\phi(N) = 60$ and $\gcd(13, 60) = 1$. Then, $d = 37$ since $13 * 37 = 481 \equiv 1 \bmod 60$. Therefore, Alice's public key is $\langle 77, 13\rangle$ and her private key is $\langle 77, 37\rangle$. Now, assume that Bob wants to send $M = 17$ to Alice (notice that $17 \in \mathbb{Z}^*_{77}$). So, Bob computes $17^{13} \bmod (77)$ to get $C = 73$ which he sends to Alice. Then, Alice computes $73^{37} \bmod (77)$ to recover $M = 17$. An immediate question is what happens when Bob wants to send a message that is larger than N or not relatively prime to N. If the message is larger than N, Bob can break the message into parts so that each part is smaller than N. Also, if the message is not relatively prime to N Bob could pad the message with extra characters to ensure that the message is relatively prime to N.

The implementation of RSA in the example is vulnerable to attacks because the primes used are too small. The use of small primes mean that the resulting N will be small and it is easy for an attacker to guess the two prime factors of N given N. The safety of RSA depends on the problem of factoring large integers. Choosing small primes makes it easier to factor the product of the two primes. An interceptor would need to have access to d in order to recover the plaintext from the ciphertext. However, the interceptor would need to know $\phi(N)$ to compute d given their knowledge of $\langle N, e\rangle$. But, computing $\phi(N)$ requires the ability to factor N into its two distinct primes which is an extremely difficult problem. It can be shown that factoring N is equivalent to the knowledge of d.

Fact 1: Given the private exponent d, it is possible to factorize N. Conversely, given the factorization of N, it is possible to figure out d.

The idea of the proof is as follows [2]: For the forward direction, suppose that we know d. By default, we also know e and N. We need to compute $k = de - 1$ and since k will be a multiple of $\phi(N)$ because $ed \equiv \text{mod}(\phi(N))$. Given that $\phi(N)$ is even, $k = 2^t r$ with r odd and $t \geq 1$. Also, $g^k \equiv 1 \bmod N$ by Euler's theorem and so $g^{k/2}$ is a square root of unity modulo N. The Chinese Remainder Theorem[3] guarantees four square roots modulo N. Using one of the non trivial square roots, x, we can factorize N using $\gcd(x-1, N)$. If $x > 1$ and $\gcd(x-1, N) > 1$, the gcd and $\frac{N}{gcd}$ gives the two prime factors. It can be shown that if g is chosen at random from $\mathbb{Z}^*_N$, there is a 50% probability that one of the elements in the sequence $g^{k/2}, g^{k/4}, ..., g^{k/2^t} \bmod N$ is a square root of unity that can be used to factorize N.

For the other direction, suppose we know the factorization of N. This means we know p and q and so we can compute $\phi(N)$. However, we also know e and d is the inverse of $e \bmod (\phi(N))$. Therefore, d can be found using the extended Euclidean algorithm.

The algorithm to implement Fact 1 in Python is included in Appendix B.

Obviously, with any cryptosystem there are concerns of security. The cryptosystem must be robust to attacks so that even if someone intercepts the message they are not able to recover information from it. The RSA cryptosystem has been analyzed since its inception to detect possible vulnerabilities. Several attacks have been designed based on mathematical theory and we discuss some of these attacks in this paper. The attacks described in this paper try to expand on the attacks mentioned in [2] and can be visited to learn about other attacks.

[3]This link provides more information about the Chinese Remainder Theorem.

2 Time complexity

Time complexity of an algorithm is an indicator of the difficulty of running an algorithm. It tells us whether it is reasonable for an algorithm to run or not. This is an important in our discussion because a multitude of attacks can be theoretically feasible but maybe not computationally. Time complexity can be thought of as the number of times a statement may execute in a computer program. This depends on the input and not on the number of processors and represents the worst case scenario. It is, however, possible that the running time of an algorithm might be shorter than the time complexity. The big O notation is used to denote the time complexity. If the time complexity is $O(f(n))$, then the running time is at most $cf(n)$ for $c \in \mathbb{Z}^+$. The following are the most common time complexities based on the input:

Name	**Notation**	**Algorithms**
Constant	$O(1)$	Finding median values in lists of sorted numbers
Linear	$O(N)$	Finding the minimum or maximum in an unsorted array
Logarithmic	$O(\log N)$	Algorithms involving iterative halving of data sets (Binary search)
Quadratic	$O(N^2)$	Algorithms involving nested iterations (Bubble sort)
Polynomial	$2^{O(\log N)}$	AKS primality Test
Sub-exponential	$2^{o(N)}$	Best-known algorithm for integer factorization(general number field sieve)
Exponential	2^N	Solving the traveling salesman problem

Table 1: Common time complexities[4][3]

We discuss attacks which have algorithms that can at least run in polynomial time. Factoring Large integers is the most obvious attack and that is why we mention it in the paper, even though it has a greater time complexity.

3 Factoring Large Integers

A direct way to break RSA is to factor N. Factoring N means that we are able to compute $\phi(N) = (p-1)(q-1)$. We can then compute $d = e^{-1} \bmod (\phi(N))$. However, as mentioned above factoring large integers is a hard task specially when the integer is a multiple of two randomly generated large prime numbers. The current fastest algorithm is the general number field sieve which still has a sub-exponential running time. This algorithm took about two years to factor a 232-digit number and a 1024-bit RSA modulus (N) is estimated to take thousand times longer[4].

There is no algorithm that can currently factor all integers in polynomial time. This means that RSA is secure as long as large enough modulus (N) is used. However, Peter Shor came up with a algorithm that runs on a quantum computer which can factorize efficiently[5]. Since his algorithm relies on quantum computing, this algorithm is not feasible until quantum computing becomes scalable.

Note: If $p-1$ for $N = pq$ is a product of prime factors less than B, then N can be factored in time less than B^3. However, this attack is not interesting and will not be discussed in the paper. Additionally, some

[4]We do not discuss the little o notation for the sub-exponential time complexity in this paper. This link can be used to learn more about it.

implementations of RSA will reject p if $p-1$ is a product of small number of primes making the attack redundant[2, Section 1.1, p.2].

4 Elementary attacks

Elementary attacks are possible due to the misuse of RSA. This might involve using small private or public exponents for individual use. This may also happen when when the same modulus N is used for a number of users. As we shall see, the attacks can be avoided by being careful in the appropriate use of RSA. Four such attacks are mentioned below:

4.1 Common Modulus

Suppose that there is a group of people communicating with each other. It is possible that all these people use the same modulus, N, to make things easier. This can be done through a trusted central authority who gives each individual an unique pair $(e_i,\, d_i)$ to form their public key $\langle N, e_i \rangle$ and private key $\langle N, d_i \rangle$.

This seems safe because none of the users know the private key of other users. However, it is possible for users to decrypt messages intended for other uses. Suppose, Bob wants to decrypt a message, M, intended for Alice. He can do so by factoring the modulus N using his own exponents e_b, d_b which is possible due to Fact 1. Then, he can recover Alice's private key d_a using her public key e_a by an application of the extended Euclidean algorithm. Once Bob has d_a, he can recover messages meant for Alice by simply intercepting the ciphertext and raising it to d_a.

Example: We will assume that there are two people, Alice and Bob. Bob wants to intercept a message intended for Alice. He needs to recover Alice's private exponent, d_a. Suppose $N = 21$ and that $e_a = 5$, $d_a = 5$, $e_b = 11$ and $d_b = 11$. Bob can use $d_b = 11$ and Fact 1 to recover that $p = 3$ and $q = 7$. Then, he can compute $\phi(N) = (3-1)*(7-1) = 12$ and use e_a to recover $d_a = 5$. Now, we suppose that the message, 16 was intended for Alice. So, the ciphertext, 4 is sent to Alice. Bob knows d_a and manages to intercept 4, then, he can use d_a to recover 16.

Appendix C contains the algorithm for the common modulus attack implemented in Python.

4.2 Low Private Exponent

It might be convenient to use a small value of d instead of a randomly generated d to reduce decryption time. However, Wiener devised an attack to break RSA when d is small. Wiener's attack is able to recover d if d is smaller than $\frac{1}{3}N^{0.25}$[2, Section 3, p.4]. We will have to introduce continued fractions before we discuss the idea behind the attack.

Continued Fractions:[5] A continued fraction is an expression of the form $a_1 + \frac{b_1}{a_2 + \frac{b_2}{a_3 + \frac{b_3}{a_4 + \ldots}}}$. It is used to represent both rational and irrational numbers but we will only be concerned with rational numbers. We will also only deal with simple continued fractions so that all the b_is are equal to 1. A simple continued fraction is of the form:

$$a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4 + \ldots}}}$$

The a_is can form an infinite sequence such that a_1 is non-negative and all the other a_is are positive. However, we will only consider finite simple continued fractions. These kind of continued fractions can also be represented by a k-tuple $[c_1, c_2, ..., c_k]$ where $c_i = a_i$ for all k $\in 1, 2, ..., k$. We can consider subsets of the k-tuple in order to form convergents of the continued fraction. For all $1 \leq j \leq k$, $[c_1, c_2, ..., c_j]$ forms the j^{th} convergent of the continued fraction.

[5] Read more about continued fractions here.

It is simple to compute the k-tuple representation of the continued fraction of $\frac{a}{b}$ given that a and b are relatively prime using the Euclidean algorithm. For example, consider $\frac{13}{35}$ and apply the Euclidean algorithm to 13 and 35:

$$\begin{aligned} 13 &= \mathbf{0}.35 + 13 \\ 35 &= \mathbf{2}.13 + 9 \\ 13 &= \mathbf{1}.9 + 4 \\ 9 &= \mathbf{2}.4 + 1 \\ 4 &= \mathbf{4}.1 + 0 \end{aligned}$$

Then, the continued fraction of $\frac{13}{35}$ is given by $[0, 2, 1, 2, 4]$.

Now, that we have given a brief description of continued fractions we will give an important result. The following theorem is extremely important in Wiener's attack and we provide it without any proof:

Theorem[6]**:** Let $a, b, c, d \in \mathbb{N}$. Assume that $(a, b) = (c, d) = 1$ and $|\frac{a}{b} - \frac{c}{d}| \leq \frac{1}{2d^2}$. Then $\frac{c}{d}$ equals one of the convergents of the continued fraction expansion of $\frac{a}{b}$.

We can now move onto Wiener's attack. The idea behind the attack is as follows: We will assume that $d < \frac{1}{3}N^{0.25}$ and that $q < p < 2q$ and we will recover d. We know that there exists an integer k such that $ed - k\phi(N) = 1(1)$. We can divide the expression by d and $\phi(N)$ to get $|\frac{e}{\phi(N)} - \frac{k}{d}| = \frac{1}{d\phi(N)}$. Therefore, $\frac{k}{d}$ is an approximation of $\frac{e}{\phi(N)}$. Using the assumptions and the fact that $N - \phi(N) = p + q - 1$ it is possible to show that $|\frac{e}{N} - \frac{k}{d}| < \frac{1}{3d^2} < \frac{1}{2d^2}$. The $\gcd(k, d) = 1$ by assumption and $\gcd(e, N) = 1$ because otherwise we could factor N. Then, by the theorem above one of the convergents of the continued fraction expansion of $\frac{e}{N}$ is equal to $\frac{k}{d}$. Since $\frac{e}{N}$ is public information we can easily compute the convergents of the continued fraction expansion. Then, it's a matter of finding out which one is the correct convergent and manipulating (1) to extract $\phi(N)$. Once we have $\phi(N)$ we just need to calculate the inverse of e mod $(\phi(N))$ to get d. The implementation of this algorithm runs in linear time.[2]

Note: Figuring out the correct convergent requires a bit of work. We need to figure out which values of convergent gives us solutions to $x^2 - (p + q)x + pq$. This is because p and q are solutions to the equation $x^2 - (p+q)x + pq$ and $pq = N$ and $p+q = N+1-\phi(N)$. If we have chosen the correct convergent, that is, the correct $\phi(N)$ we will be able to find the correct prime factors. Otherwise, we would not have any solutions. Also, we are only concerned with finite continued fraction expansions because we are dealing with rational numbers given e and n are both integers. Finding the continued fraction expansion of rational numbers is equivalent to finding the gcd of the numerator and denominator using the Euclidean algorithm which we know terminates.

The algorithm for Wiener's attack is included in Appendix C. A more formal proof of the attack can be found in [6].

Example[?] Suppose $e = 47318087$ and $N = 60477719$. Then the continued fraction expansion of $\frac{e}{N}$ is $[0, 1, 3, 1, 1, 2, 8, 1, 9, 4, 1, 4, 1, 1, 4, 2, 1, 1, 2, 2, 3]$. The first few convergents are $\frac{0}{1}, \frac{1}{1}, \frac{3}{4}, \frac{4}{5}, \frac{7}{9}, \frac{18}{23}$ and so on. For $\frac{18}{23}$, we get that the prime factors are $p = 6719$ and $q = 9001$. So, $\phi(N) = 60462000$ and by the extended euclidean algorithm, $d = 23$.

4.3 Low Public exponent

Low public exponents might be used to reduce encryption time. However, there are two mentionable attacks which make it possible for an interceptor to recover the plaintext from the ciphertext when a small public exponent is used. These attacks require at least two messages to be intercepted unlike the case of the low private exponent. The attacks are described below:

4.3.1 Hastad's Broadcast Attack

Suppose Alice wants to send a message, M, to a number of people $P_1, P_2, ..., P_k$. The people will have different N_is but the same public exponents, e. It is important to notice that M has to be less than all of the N_is. Now, Alice sends the message M to each person using their public keys. Bob wanting to recover the message can do so by possibly intercepting all the ciphertexts. Bob can recover the message as long as the number of messages being intercepted is at least as big as the public exponent, that is, $k \geq e$.

Idea: we will assume all public exponents are equal to 3, that is, $e_i = 3$ for $i \in 1, 2, .., k$. Bob has to at least intercept 3 ciphertexts, say, C_1, C_2 and C_3. Bob knows that $C_1 = M^3 \bmod N_1$, $C_2 = M^3 \bmod N_2$ and $C_3 = M^3 \bmod N_3$. We will also assume that $\gcd(N_i, N_j) = 1$ for all $i \neq j$ since otherwise Bob can factor some of the N_is. We can apply the Chinese Remainder Theorem to C_1, C_2, C_3 to get $C' \in \mathbb{Z}^*_{N_1 N_2 N_3}$ such that $C' = M^3 \bmod N_1 N_2 N_3$. Bob can recover M by computing $(C')^{\frac{1}{3}}$ since $M^3 < N_1 N_2 N_3$.

This is the simplest form of the Hastad's broadcast attack. This is based on Coppersmith's theorem which we will not discuss in this paper. The proof uses LLL lattice basis reduction algorithm and both theorem and its proof can be found in [2, Section4, p.6]. Hastad went onto generalize this attack by showing that any sort of padding is still insecure. The algorithm for the simple broadcast attack is included in Appendix C.

Example: For simplicity's sake we will assume that all public exponents are equal to 3. we suppose that Alice wants to send the message $M = 10$ to three people, P_1, P_2 and P_3. Let $N_1 = 87$, $N_2 = 115$ and $N_3 = 187$. We will also assume that Bob has intercepted the 3 ciphertexts, $C_1 = 43, C_2 = 80$ and $C_3 = 65$. Bob can then use the Chinese Remainder Theorem to calculate $C' = 1000 \bmod (87 * 115 * 187)$. All Bob needs to do right now is to compute $1000^{\frac{1}{3}}$ to get back the message $M = 10$.

4.3.2 Franklin-Reiter Related Message Attack

Suppose that Bob wants to send Alice two encrypted messages using the same modulus. If $\langle N, e \rangle$ is Alice's public key, Bob sends his messages $M_1, M_2 \in \mathbb{Z}^*_N$ by computing $M_i^e \bmod (N)$ for $i = 1, 2$. However, we assume that $M_1 = f(M_2)$ for some publicly known polynomial $f \in \mathbb{Z}_N[x]$. Franklin and Reiter showed that an attacker can recover M_1, M_2 given $(N, e, C_1, C{+}2, f)$ where $C_i \equiv M_i \bmod (N)$ for $i = 1, 2$. For $e = 3$, the time complexity is quadratic in $\log N$ and for $e > 3$, the complexity changes to quadractic in e[2, Section 4.3, p.9].

The idea of the attack is as follows: We consider $g_1(x) = (f(x))^e - C_1 \in \mathbb{Z}_N[x]$. We notice that M_2 is a root of this polynomial since $f(M_2) = M_1$. Similarly, M_2 is a root of the polynomial $g_2(x) = x^e - C_2$. The linear factor $x - M_2$ divides both polynomial. We can then use the Euclidean algorithm for polynomials to compute the greatest common factor of g_1 and g_2. If the common factor is linear we have M_2.

Appendix C contains the algorithm for this attack. Note: For $e = 3$, the greatest common factor must be linear. The proof can be found here. For $e > 3$, the greatest common factor is most likely to be linear. However, the attack fails if it is not linear[2].

Example: For simplicity's sake we will assume that $e = 3$. We suppose that Alice's public key is $15, 3\rangle$ and so her private key is $15, 3\rangle$. Bob wants to send $M_1 = 4$ and $M_2 = 7$. The publicly known poylnomial, f, is $x - 3$. However, $f \in \mathbb{Z}_N[x]$ so $f(x) = x + 12$. The attacker can intercept the two ciphertexts, $C_1 = 4$ and $C_2 = 13$. Then the attacker computes $g_1(x) = (x - 3)^3 - 4$ and $g_2(x) = x^3 - 13$. With the knowledge that M_2 will be a root of the two polynomials the attacker finds out that the only root of g_1 and g_2 is 7. So, $M_2 = 7$ and then the attacker knows that $M_1 = f(M_2) = 7 - 3 = 4$. Therefore, both the messages are recovered by the attacker.

5 Digital Signatures

Another important application of the RSA cryptosystem is to provide digital signatures. Digital signatures ensure the authenticity of electronic legal documents and are used to sign electronic checks. To sign a message $M \in \mathbb{Z}^*_N$, we just apply $\langle N, d \rangle$, our private key, to it. This gives us $S = M^d \bmod N$. The authorities have access to our public key $\langle N, e \rangle$ and can use that to verify if $S^d = M \bmod N$. This is the same phenomenon used in the encryption and decryption process in communicating except the exponents are reversed. Just like for secure communication, it is important for digital signatures to be safe from possible attacks. Several attacks have been devised to test the vulnerability of digital signatures and we discuss one of the attacks below:

5.1 Blinding

This attack can be considered when Bob wants Alice to sign a particular message, M. However, Alice may be wary of this fact and decide not to sign it. Bob may be able to trick Alice to sign a different message, m instead by making it seem legit and trustworthy and try to recover the signature, S, on M. So, what should m be? It turns out that multiplying M by a random $r \in \mathbb{Z}_N^*$ raised to e does the trick. If Bob manages to get Alice to sign m, he receives $s = m^d \bmod (N)$, he could compute $\frac{s}{r}$ to get $S = M^d$[2, Section 2.2, p.4].

$$S^e = \frac{(s)^e}{r^e} = \frac{m^{ed}}{r^e} \equiv \frac{m}{r^e} = M \bmod (N)$$

Appendix C contains an algorithm for the blinding attack.

Example: Suppose Alice has the public key, $\langle 15, 3 \rangle$. So, her private key is $\langle 15, 3 \rangle$. Now, let us assume that Bob wants Alice to sign the message 2. He will first blind this message using a random $r \in \mathbb{Z}_N^*$. We will suppose that $r = 7$. So, the blinded message is $m = 11$. Then, he will send it to Alice who will sign the blinded message. Bob receives $s = 11$ and then computes $\frac{s}{r} \bmod (N)$ to get the original signature, $S = 8$. Notice: $2^3 = 8 \bmod 15$ which would have been Alice's signature on the original message, $M = 2$.

6 Conclusion

It is easy to observe that a proper implementation of RSA will make most attacks in this paper ineffective. A smart implementation would make sure that the exponents are high enough; for example, a public exponent greater than $2^{16} + 1$ will break Hastad's attack. It would also avoid using the same modulus when sending multiple messages to multiple people. Still, these are clever attacks based on mathematical theory and should be discussed and studied. It is also important to note that once quantum computing becomes scalable RSA would break. This is because Shor's algorithm would be able to factor large integers efficiently which would make RSA insecure[5].

Appendix A: Basic number theory algorithms in Python

These contains the euclidean algorithm and the extended euclidean algorithm. The euclidean algorithm computes the gcd of two positive integers and the extended euclidean algorithm can be used to return the modular inverse of a positive integer. Both these algorithms are extremely important and useful in implementing RSA and breaking RSA. These are efficient algorithms and involve simple computations once all the inputs are provided.

```
def gcd(a, b):
    # Returns the GCD of positive integers a and b using the Euclidean Algorithm.
    x, y = a, b
    while y != 0:
        r = x % y
        x = y
        y = r
    return x

def extendedGCD(a,b):
    # Returns integers u, v such that au + bv = gcd(a,b).
    x, y = a, b
    u1, v1 = 1, 0
    u2, v2 = 0, 1
    while y != 0:
        r = x % y
        q = (x - r) // y
        u, v = u1 - q*u2, v1 - q*v2
        x = y
        y = r
        u1, v1 = u2, v2
        u2, v2 = u, v
    return (u1, v1)

def findModInverse(a, m):
    # Returns the inverse of a modulo m, if it exists.
    if gcd(a,m) != 1:
        return None
    u, v = extendedGCD(a,m)
    return u % m
```

Appendix B: Implementation of Fact 1 in Python

These are the algorithms to recover d from the factorization of N and vice versa. The first function uses the two prime factors of N to recover d with the help of the extended euclidean algorithm from Appendix A. The second function gives the factorization of N using d with the help of the proof of Fact 1. Since, a random g has a $\frac{1}{2}$ probability of giving the factorization of N it is possible that the function has to be run several times to make sure the function chooses the g that will give the factorization. Also, $x = (\text{pow}(g, t))\%N$ may return an overflow error due to the constraints of the computer. In this case, we can just run it again so that the function can pick another g to avoid the problem.

```
import Cryptomath, random

def Ntod(p,q, public):
    #Returns private exponent, d, using the public key and the factors of N
    N = public[0]
    e = public[1]
    phi = (p-1)*(q-1)
    d = Cryptomath.findModInverse(e, phi)
    return d


def dtoN(d, public):
    #Factorizes N to return [p, q] given <N, e> and d
    N = public[0]
    e = public[1]
    k = d*e - 1
    g = random.randint(2, N)
    t = k
    x = 0
    gcd = 0
    foundfactor = False
    while not foundfactor:
        t = t/2
        x = (pow(g,t))%N
        gcd = Cryptomath.gcd(x-1, N)
        if x > 1 and gcd >1:
            foundfactor = True
    p = int(gcd)
    q = int(N/gcd)

    return [p,q]
```

Appendix C: Implementation of RSA attacks

C.1 Common Modulus Attack: The attack uses the two functions mentioned in Appendix B. The function takes in our public and private key and the public key of Alice, the person the message was intended for. The function also assumes that we have the ciphertext in hand. The function gives out the factorization of N and the private exponent using Fact1. After that it is a simple calculation to recover the message. It is also possible to modify this function so that it does not assume we have the ciphertext in hand. Then, this function would return the private exponent and this could be stored for later use. We would be able to decrypt any messages meant for Alice.

```
def commonmod(public, private, public_a, C):
    #Returns the factors of N: p and q, private exponent d and the message M
    N = public[0]
    e = public[1]
    d = private[1]
    e_a = public_a[1]
    factors = dtoN(d, [N,e]) #factorizes N
    p = int(factors[0])
    q = int(factors[1])
    print("p:", p, "q:",q)
    d_a = Ntod(p,q, public_a) #computes the private exponent
    print("d:", d_a)
    M = pow(C, d_a) % N
    print("M:", M)
```

Example input :

```
commonmod([21, 11], [21,11],[21, 5], 4)
```

Example output:

```
p: 3 q: 7
d: 5
M: 16
```

C.2 Wiener Attack: The wiener attack employs the euclidean algorithm from Appendix A and modifies it to return a list of quotients. This gives the tuple representation of the continued fraction expansion of $\frac{a}{b}$. Then, we find the convergents of the continued fraction expansion using a recurrence like structure. Once we have found the convergents we have to check which one is the correct one and having found the correct one we compute the private exponent.

```
import Cryptomath, math
from fractions import Fraction

def euclid(a,b):
    # Returns tuple representation of the continued fraction expansion of a/b
    x, y = int(a), int(b)
    list = []
    while y != 0:
        list.append(x//y)
        r = x % y
        x = y
        y = r
    return list

def wiener(public):
    #Returns p,q and d using convergents of continued fraction expansion
    N = int(public[0])
    e = int(public[1])
    a = euclid(e,N)
    print('expansion:', a)
    #starting the recurrence
    n0 = a[0]
    d0 = 1
    n1 = a[0]*a[1] + 1
    d1 = a[1]
    list = [Fraction(n0,d0),Fraction(n1,d1)]

    for x in a[2:]:
        n = x*n1 + n0 # numerator
        d = x*d1 + d0 # denominator
        list.append(Fraction(n,d))
        n1, n0 = n, n1
        d1, d0 = d, d1

    #Checking for the right convergent
    for l in list:
        if l.numerator != 0:
            M = ((e*l.denominator) -1)/l.numerator
            b = (N + 1 - M)*(-1)
            c = N
            r = pow(b,2) - 4*c
            p = 0
            q = 0
            if r > 0:
                p = ((((-b) - math.sqrt(r))/(2)))
                q = (((-b) + math.sqrt(r))/(2))
                if (p * q) == N:
                    phi = M
                    d = Cryptomath.findModInverse(e,phi)
                    if d!= None:
                        break
    print('p:', p)
    print('q:', q)
    print('d:', d)
```

Example input:

```
wiener([60477719, 47318087])
```

Example Output:

```
expansion: [0, 1, 3, 1, 1, 2, 8, 1, 9, 4, 1, 4, 1, 1, 4, 2, 1, 1, 2, 2, 3]
p: 6719.0
q: 9001.0
d: 23.0
```

C.3 Hastad's Broadcast attack: We use that the public exponents are equal to 3 and that the number of messages is also 3.The attack involves using the Chinese remainder theorem. We implement the Chinese Remainder Theorem using Gauss' algorithm. It is important to notice that this attack can be generalized to a greater number of messages by creating a list of the moduli and the ciphertexts and iterating over them as long as the number of messages is greater than e.

```
def hastad(e=3, public1 = [], public2= [], public3=[], c1 =0, c2=0, c3 = 0):
    #Returns the message M using the Chinese remainder Theorem
    N1 = public1[0]
    N2 = public2[0]
    N3 = public3[0]
    N = N1*N2*N3
    n1 = N/N1
    n2 = N/N2
    n3 = N/N3
    d1 = Cryptomath.findModInverse(n1, N1)
    d2 = Cryptomath.findModInverse(n2, N2)
    d3 = Cryptomath.findModInverse(n3, N3)
    #Computing C using Gauss' algorithm
    C = (c1*d1*n1 + c2*d2*n2 + c3*d3*n3)%N
    M = int(round(pow(C,1/3)))
    print("Message:",M)
```

Example input:

```
hastad(3, [87,3],[115,3], [187,3], 43, 80, 65)
```

Example Output:

```
C: 1000.0
Message: 10
```

C.4 Franklin-Reiter Related Message Attack: The attack is implemented in Sage. We use the already existing roots method in Sage which returns the roots of a given polynomial. The attack compares the roots of the two polynomials $g1$ and $g2$ and computes a list of common roots. Then, it proceeds to finding the largest common root which is $M2$. Once $M2$ has been found, $M1$ can be computed simply by using the publicly known polynomial, f.

```
def frm(public =[N,e], C1 = 0, C2 =0, f =0):
    N = public[0]
    e = public[1]
    S = PolynomialRing(Zmod(N), 'x')
    g1 = (f^e - C1)
    g1 = S(g1)
    g2 = x^e - C2
    g2 = S(g2)
    g1 = g1.roots(multiplicities = False)
    g2 = g2.roots(multiplicities = False)
    common = []
    for g in g1:
        for j in range(len(g2)):
            if g2[j]==g:
                common.append(g)

    m2 = max(common)
    m1 = (f(m2))
    print("M1",m1)
    print("M2",m2)
```

Example input:

```
frm([15,3], 4, 13, x-3)
```

Example Output:

```
('M1', 4)
('M2', 7)
```

C.5 Blinding Attack: The attack requires two functions. The first function picks a random $r \in \mathbb{Z}_N^*$ and creates the blinded message. The function returns the r which needs to be used later. The second function converts the signature on the blinded message to the original signature. The second function uses the signature sent by Alice as an input. For the sake of the example, we compute it using Alice's private exponent. Even though the computation should be as simple as $\frac{s}{r}$ we have to be careful because we are dealing with modular arithmetic. The signature Bob receives might not be divisible by r because it is mod N. We have to check all possible values of s because it can be $s+$ any multiple of N. Once, we found the correct one we can divide it by r to retrieve the original signature.

```
mport Cryptomath, random
from fractions import Fraction

def blind(M, public):
    #Returns a blinded message and the random r used to blind the message
    N = public[0]
    e = public[1]
    foundr = False
    while not foundr:
        r = random.randint(2, N-1)
        if Cryptomath.gcd(r, N) == 1:
        foundr = True
    m = (pow(r,e)*M)%N
    print("r:", r)
    print("m:", m)

def signature(s,r, public):
    #Converts the signature on the blinded message to signature on the original message
    N = public[0]
    #Finding the correct value of s which is divisible by r
    for i in range(1000):
        m = s + i*N
        S = Fraction(m/r)
        if S.denominator == 1:
            break
    S = S%N
    print("S:",S)
```

Example input:

```
blind(2, [15,3])
signature(11, 7, [15,3])
```

Example Output:

```
r: 7
m: 11
S: 8
```

References

[1] Kalliski, Burk. "The Mathematics of the RSA Public-Key Cryptosystem". RSA Laboratories, http://www.mathaware.org/mam/06/Kaliski.pdf.

[2] Boneh, Dan. "Twenty years of attacks on the RSA cryptosystem". Notices of the AMS 46.2, pages 203-213, 1999.

[3] "Time complexity". Wikipedia. Wikipedia.org. 26 Jan, 2016, https://en.wikipedia.org/w/index.php?title=Time_complexity&action=history.

[4] Kleinjung; et al. 'Factorization of a 768-bit RSA modulus". International Association for Cryptologic Research, https://eprint.iacr.org/2010/006.pdf.

[5] Shor, Peter W. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer". SIAM review, 41.2, pages 303-332, 1999.

[6] Backes, Michael. "Wiener's attack". Foundations of Cybersecurity, Lecture 9, Winter 16/17, https://www.infsec.cs.uni-saarland.de/teaching/16WS/Cybersecurity/lecture_notes/cysec16-ln05.pdf.

[7] Rothe Jorg. "Complexity Theory and Cryptology: An Introduction to Cryptocomplexity". Springer, 2010.

[8] Weisstein, Eric W. "Chinese Remainder Theorem". Wolfram MathWorld, http://mathworld.wolfram.com/ChineseRemainderTheorem.html

[9] Knott, Ron. "Continued Fractions". University of Surrey, 1996, http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/cfINTRO.html.