How to break RSA

1st Jakob Stadelmann
University of Applied Sciences
Ingolstadt, Germany
jas4613@thi.de
00121716

2nd Johan Bücker
University of Applied Sciences
Ingolstadt, Germany
job8197@thi.de
00121098

Abstract

This paper explains the inner workings of RSA and then details a selected set of attacks against the cryptographic primitive as well as its implementations. Classical attacks, side-channel attacks, and Shor´s algorithm were chosen to cover a variety of attacks. However, it is important to note that non of these actually break RSA in practice.

Index Terms

RSA, Quantum Computing, Shors algorithm, Side channel attacks

Acronyms

CPU	Central Processing Unit
Qubit	Quantum Bit
RSA	Rivest–Shamir–Adleman
SSH	Secure Shell

I. Introduction

Developed in 1978 by Rivest, Shamir and Adleman, the RSA algorithm revolutionized the field of information security by establishing trust in an increasingly more connected world. Unlike traditional symmetric encryption methods, where a single secret key is used for both encryption and decryption, RSA relies on the concept of asymmetric encryption, also known as public-key encryption. The advantage of asymmetric encryption is that different public and private keys are used for each operation. This approach is a major milestone for the key distribution problem plaguing all ciphers but also is a key component of digital signatures.

The widespread adoption of RSA has transformed Internet communications. RSA provides the foundation for online banking, e-commerce, secure messaging and data transmission. With individuals, organizations and governments worldwide relying on RSA as a trusted encryption standard. The security of RSA must be guaranteed to prevent data breaches, financial losses, privacy violations and the weakening of public trust in digital security. However, the very ubiquity of RSA makes it a prime target for attackers looking to exploit vulnerabilities. Various types of attacks have emerged over the years, including factorization, side-channel attacks containing timing attacks and quantum computing-based attacks. Among these, quantum computing-based attacks such as the Shor algorithm pose an increasing threat with the advancement of quantum computing.

This paper will first give an introduction to the inner workings of RSA. Afterwards classical attacks on the cryptosystem are discussed. This section includes a theoretical break if certain conditions are met as well as side-channel attacks that exploit weaknesses in implementations of RSA. Following this Shor's algorithm is presented with the basics of quantum computing.

II. RSA CRYPTOSYSTEM

The RSA cryptosystem is widely used in the age of internet and well known. The algorithm was proposed by Rivest, Shamir and Adleman in the paper "A method for obtaining digital signatures and public-key cryptosystems" [8] in 1978. It is a public-key cryptosystem based on number theory. The security is based on so called *trapdoor function*. Those functions are easy to calculate in one direction, but very difficult to calculate in the other direction. An example for such a trapdoor function is the multiplication of two large prime numbers. It is easy to calculate the product of two prime numbers, but it is very hard to deduce the two prime factors from the product. This mathematical problem - the difficulty of the decomposition of large numbers - is the base for the security of RSA. "The security of the RSA algorithm has so far been validated, since no known attempts to break it have yet been successful." [6, p. 1] RSA is the first algorithm suitable for both data encryption and digital signature applications [6, p. 1].

A. Public-key cryptosystems

In public-key cryptosystems the encryption and decryption of messages M are done with two different procedures. The encryption key E is publicly shared, while the decryption key D is kept secret. Both keys can be used for both operations with RSA. You can use the private key to sign something or to decrypt a message only meant for you. Both keys are connected mathematically with each other. In the case of RSA those keys are sets of two special numbers. Public-key cryptostystems must have the property that the original message M is obtained when decprypting and encrypting M. In specific $D(E(M)) = M$ needs to be fulfilled. The reversed procedure $E(D(M))$ still returns M. This property is needed to provide signatures. It is also important that E and D are easy to compute. To ensure security it must not be possible to derive D from E. D must be kept a secret even when publishing E. It is important that the private key cannot be derived from the public key, so that there is no risk of leaking the secret when publishing the public key [6, p. 2].

B. The RSA Algorithm

After this short introduction into public-key cryptosystems the functionality of the RSA Algorithm is explained [12]:

1) Choose two very large prime numbers p and q. Those two prime numbers must be secret. The bigger the prime numbers are, the higher is the level of security. In practise the prime numbers usually have a length of 3000 bits and more to ensure security. As a small example for illustration, let $p = 5$ and $q = 13$.
2) The product N of the two primes is calculated: $N = p \cdot q$. It must not be possible to factor N into its two prime factors p and q. In the example $N = 5 \cdot 13 = 65$
3) In the next step $\varphi(N)$ is calculated. The φ-function $\varphi(N)$ indicates how many whole numbers, that are smaller than N, are relatively prime to N. In this case it is easy to calculate, because p and q are prime factors. This means $\varphi(N) = \varphi(p \cdot q) = (p-1) \cdot (q-1)$. Applied on the example: $\varphi(65) = \varphi(5 \cdot 13) = (5-1) \cdot (13-1) = 48$.
4) Choose e with $1 < e < \varphi(N)$ and $\gcd(\varphi(N), e) = 1$. The second rule ensures that e and $\varphi(N)$ are relatively prime. The condition is necessary, because the multiplicative inverse of $e \mod \varphi(N)$ is required for the private key. This multiplicative inverse is called d. It only exists for sure if e is not relatively prime to $\varphi(N)$. In this example, $\varphi(N) = 48, e = 7$ => $1 < e < 48$ and $\gcd(48, 7) = 1$.

5) d is calculated with the extended euclidean algorithm [12, p. 7]: $e \cdot d = 1 \mod \varphi(N)$ Using this algorithm on the example, $7 \cdot 55 = 1 \mod 48$. So the matching $d = 55$ to $e = 7$ is found.
6) The encryption key (e, N) in the case of the example $(7, 65)$, is published. The private key $(d, N) = (55, 65)$ is kept secret. This means that the security of RSA is based on the secret number d. This decrypting number is easy to find when the factorisation of $N = p * q$ is known and difficult to find from the given N and e - given in the public key - when the factorisation is not known.

To encrypt the message m the public key (e, N) is used. For the encryption m is raised to the power e and reduced by $\mod N$. The result of this calculation is the encrypted message $c = x^e \mod N$, also called cyphertext c. The receiver of the message c uses the private key (d, N) to decrypt the message. To obtain the original message m the encrypted text c is raised by the power d and reduced $\mod N$, which results in the equation $m = c^d \mod N$ [12, p. 12].

For a better understanding of the process an example is illustrated: The message $m = 6$ needs to be encrypted. For this the public key $(7, 65)$ is used. Now the encrypted text, also known as cyphertext c, can be calculated, $c = m^e \mod N = 6^7 \mod 65 = 46$. Then $c = 46$ is decrypted with the private key $(d, N) = (55, 65)$ to obtain $m = c^d \mod N = 46^{55} \mod (N) = 6$. Why does c^d results in the original message m? A closer look at the decryption reveals that,

$$c^d \equiv (m^e)^d \equiv m^{e \cdot d} \tag{1}$$

It is known that d is the multiplicative inverse to e. Therefor $e \cdot d = 1 \mod \phi(N)$. This equation can also be written as

$$e \cdot d \equiv k \cdot \phi(N) + 1 \tag{2}$$

Equation 2 is now inserted into Equation 1, which results in the following equation:

$$c^d \equiv (m^e)^d \equiv m^{e \cdot d} \equiv m^{k \cdot \phi(N)+1} \tag{3}$$

Now Euler's theorem can be used on this new equation. Euler's theorem states that the following Equation 4 applies for all natural numbers k, m and N with $m < N$ [12, p. 10, 11]:

$$m^{k \cdot \phi(N)+1} \equiv m \mod N \tag{4}$$

Using Euler's theorem the following equation is obtained,

$$c^d \equiv (m^e)^d \equiv m^{e \cdot d} \equiv m^{k \cdot \phi(N)+1} \equiv m \mod N \tag{5}$$

The steps above illustrate that $c^d = m$. With that the decryption is completed and the message $m = 6$ was successfully encrypted and decrypted with the RSA-Algorithm.

III. Classical attacks

Generally attacks on RSA focus on factoring N as "if an efficient factoring algorithm exists, then RSA is insecure." [2, p. 3] To componsate for ever faster algorithms and increases in computational power the size of N in bit has increased over the years. The German Federal Office for Information Security current advisory for RSA states that "the length of the modulus n should be at least 3000 bit." [3, p. 31] However, even when using a large enough N there are attacks that could break RSA albeit they require a specific set of conditions to be true.

A. Reused Modulus

Generating truly random numbers and testing them for primality takes a lot of time. Assuming someone would generate a RSA key pair for a third party they could reuse an already generated modulus that was used with another third party. This then increases how many key pairs can be generated in a given amount of time. Users would then receive a unique key pair (d_i and e_i respectively) and a shared modulus N.

One of the users could then use their own private key d to recover the initial primes p and q that make up N. Using the Chinese Remainder Theorem these "can be be efficiently computed in time $O(n^3)$ where $n = \log_2 N$" [2, p. 3]. Having recovered p and q it is possible to reveal the private key for any public key using the same modulus. This then compromises every other user with this N.

B. Reused prime

As an alternative to sharing both p and q one might try to only generate one new prime number for every N and reuse the one generated for the previous N as the second factor. This would reduce the needed work to generate a new RSA key pair.

However, given three primes p_0, p_1, p_2 and two moduli N_0 and N_1 with $N_0 = p_0 \cdot p_1$ and $N_1 = p_1 \cdot p_2$ one can easily see that $gcd(N_0, N_1) = p_1$. The greatest common divisor can be efficiently computed with complexity $\mathcal{O}(\log N)$ which breaks the RSA cryptosystem in this particular case.

C. Insufficient private key size

Working with large integers is computationally expensive and requires a lot of Central Processing Unit (CPU) cycles. Typically as the size of the integer being operated grows more cycles are necessary and more time is required to complete the operation. To reduce the amount of time a cryptographic operation using RSA takes one could be inclined to use a small private key.

Unfortunately using "a small d results in a total break of the cryptosystem." [2, p. 4]. In [7] it is experimentally shown that attacks on private keys less than $\sqrt[4]{N}$ are feasible. For this two different approaches were leveraged.

The first attack can "recover the private exponent d based on the solution of the shortest problem finding for two dimension lattice." [7, p. 1] It uses the Gaussian's lattice reduction algorithm which only has logarithmic complexity. This makes it possible to do the required computations in an acceptable time frame.

The second attack called the Wiener attack can recover the private key "if $p < q < 2p, e < pq$ and $d < N^{1/4}$" [7, p. 2]. It exploits the fact that the private and public are constrained by $ed \equiv 1 \mod \varphi(N)$. Consequently "there is a $k \in \mathbb{Z}$ such that $ed = 1 + k\varphi(N)$" [7, p. 2]. This can be rewritten as:

$$\left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| = \frac{1}{d\varphi(N)}$$

Because $1 < d \ll \varphi(N)$ can be assumed for virtually all RSA keys $\frac{1}{d\varphi(N)}$ will almost be zero and as a result $\frac{e}{\varphi(N)}$ is approximately equal to $\frac{k}{d}$. Additionally $\varphi(N) \approx N$ can be used to replace the unknown order of N with the public modulus. Since "$\varphi(N) = N - (p + q - 1)$ and $p + q - 1 < 3\sqrt{N}$ we have $|N - \varphi(N)| < 3\sqrt{N}$" [7, p. 2]. This can be used to further rewrite the equation (replacing $\varphi(N)$ with N).

$$\left| \frac{e}{N} - \frac{k}{d} \right| = \frac{1}{dN} \leq \frac{3k}{d\sqrt{N}} < \frac{1}{2d^2}$$

"So $\frac{k}{d}$ is a convergents of the continued fraction expansion of $\frac{e}{N}$" [7, p. 2]. With this knowledge one can create a continued fraction expansion from the publicly available e and N and then calculate the convergents of that fraction.

"One of these will equal $\frac{k}{d}$" [2, p. 5] and can be identified by the fact that "$\gcd(k, d) = 1$ and hence $\frac{k}{d}$ is a reduced fraction." [2, p. 5]

D. Passive fault attack

Faults in RSA operations can lead to the leakage of information. Usually an attacker has to actively provoke such a fault which would require access to the device on which the operation is performed. In 2023, it was shown that a passive attacker eavesdropping on a network "can opportunistically obtain private RSA host keys from an SSH server that experiences a naturally arising fault during signature computation." [9, p. 1]

It was shown that out of 5.2 billion SSH records there were 590,000 vulnerable signatures. A total of 4,900 signatures revealed 189 unique private keys. The problem lies in the fact that the faults enabling this attack were not actively created but occurred on their own. As a result "implementations should validate signatures before sending them" [9, p. 13] to prevent the leakage of the private key.

IV. Side channel attacks

While cryptographic algorithms might be perfectly secure in theory, their implementations can be vulnerable to side channel attacks. These kinds of attacks exploit the fact that algorithms might leak information in addition to the expected output through unintended side effects that can then be measured using side channels. In the worst case would this leaked information allow an attacker to recover secrets used in the algorithm. Regarding RSA this would mean the disclosure of the private key d.

Side channels can take on a variety of form. A few typical examples include:

- Run time
- Power consumption
- Electromagnetic radiation

A. Timing based attacks

Usually algorithms are optimized to improve one ore more characteristics of it. Commonly the run time of a algorithm is the primary objective of optimization efforts. Listing 1 shows how a function comparing to character sequences, also called a string, could be implemented. A non zero return value would indicate either a mismatch in string length or content. If such a function would be used to check an entered password by comparing the user input to the correct password an attacker could find out the secret by using a bruteforce attack in a much shorter time frame.

A bruteforce attack involves simply trying all possible input combinations until a working input is found. While any classical cryptographic algorithm is vulnerable against such an attack in theory, it can be easily mitigated by increasing the average amount of time required to find a valid input. This can be achieved by e.g. increasing the complexity of the function or limiting the rate by which an attacker can try inputs.

Using the implementation of the password validation as an example it is apparent that it lacks the necessary complexity to prevent such an attack if additional information from a side channel is used. Given that the correct password is l characters long and every character can be any member of the set $C = \{c_0, c_1, \ldots, c_n\}$ the password can be any one of $(n + 1)^l$ combination.

```
1  char str_cmp(char* lhs char* rhs) {
2      char diff;
3      while(*lhs != '\0' && *rhs != '\0') {
4          if((diff = *lhs - *rhs) != 0)
5          return diff;
6          lhs++;
7          rhs++;
8      }
9      return *lhs - *rhs;
10 }
```

Listing 1: String compare function

```
1  def exp_mod(m: int, exp: int, N: int) -> int:
2      if exp == 0:
3          return 1
4      res: int = 1
5      while exp != 0:
6          if exp & 1 == 1:
7              res = (res * m) % N
8          m = (m * m) % N
9          exp = exp >> 1
10 return res
```

Listing 2: Repeated squaring algorithm

However, due to the fact that the function will exit as soon as a difference between the two inputs is detected in line 4 an attacker can try individual characters at a single time. The attacker would start with an empty guessed password g. He would then enter a new guess $g \oplus c_i$ with $c_i \in C$. Then he measures the time t_i it takes for the system to respond. If the system returns success $g \oplus c_i$ is the correct password. Otherwise he checks whether t_i is longer than for any other guess $g \oplus c_i$. If so, he append c_i to g and tries the process again with the new g. If not he tries a different $c_i \in C$.

This algorithm will take at most $l \cdot n$ iterations and as such has only $\mathcal{O}(n \cdot l)$ complexity. The bruteforce approach has $\mathcal{O}(n^l)$ complexity because it can only try a complete password instead of individual characters. It is apparent that $\mathcal{O}(n \cdot l) \ll \mathcal{O}(n^l)$.

B. Timing attack on RSA

Similarly the runtime of RSA operations might depend on the input and the secret key. A naive implementation of the exponentiation modulo N required for decryption and encryption could use the repeated squaring algorithm as described in Listing 2. In the case of RSA private key operations `exp` equals d.

This reduces the necessary multiplications by squaring m and multiplying it with the result whenever the corresponding bit in the exponent is set. Squaring a power of m such as m^n has the same effect as multiplying m^n by m n times. This is possible due to:

$$(m^n)^2 = m^n \cdot m^n = m^{2n} = m^n \cdot \underbrace{m \cdot \dots \cdot m}_{\times n}$$

Assuming the implementation in Listing 2 is used on a smartcard, which can perform cryptographic operations without disclosing the private key inside it. One would ask "the smartcard to generate signatures on a large number of random messages $M_1, \dots, M_k \in \mathbb{Z}_N^*$ and measures

the time T_i it takes the card to generate each of the signatures." [2, p. 12] The least significant bit of d is 1 because it is odd. Consequently one would start with the next bit d_1.

If d_1 is 1, the statement in line 7 would be executed which for d_1 equals $M_i \cdot M_i^2 \mod N$. Both parameters of this step are known to the attacker. Because the time t_i to compute $M_i \cdot M_i^2 \mod N$ depends on the value of M_i one can measure it in advance on an identical smartcard. If t_i is comparatively large and the currently tested bit is true it likely has an effect on the overall time T_i. This correlation of t_i to T_i can be measured.

If the measurement confirms a correlation d_1 is likely 1. In the case that they "behave as independent random variable" [2, p. 12] it can be assumed that d_1 is 0.

With the knowledge of d_0 and d_1 the attacker can now move on to d_2. He can again measure t_i for next iteration and then see if there is a correlation. In the case that d_1 is 0 he does the computation $M_i \cdot M_i^4 \mod N$ and else $M_i^3 \cdot M_i^4 \mod N$. Continuing like this the attacker can recover the private key from a secure storage medium like a smartcard bit by bit.

C. Power consumption attacks

Cryptographic operations consume power when running. This exposes a side channel because power consumption can vary depending on the input to a function. An attacker can send a smartcard a large amount of inputs that the smartcard should sign. The attacker than measures the power consumption of the device and tries to recover the private key.

Modern hardware tries to prevent the exposure of secret information through this side channel using "message blinding, exponent blinding and the multiply-always exponentiation scheme" [13, p. 1]. However, there are still ways around these protections. In the "case where both a short public exponent and a randomization of the private exponent are used [...] information on the private exponent can be obtained from the public key and can be used to efficiently recover the whole private key." [13, p. 1]

In Zhao et al. [13] the Montgomery modular multiplication is attacked. They get around message blinding, a technique used to break the direct correlation between power consumption or other side channels and the private key. By using a random number that masks the message itself, while the private key operates on it, an attacker does not know the value of M_i. E.g. the attack described in subsection IV-A relies on the knowledge of M_i. By trying out many different blinding factors and measuring the power consumption they can recover it. This allows them to look through the blinding and recover the private key.

D. Acoustic side channel attacks

Typical hardware components are not completely silent. They might vibrate just a little bit under changing electromagnetic forces. E.g. lower end Notebooks might exhibit a high pitched noise when charging. This noise is more than random noise it "can [...] in particular leak sensitive information about security-related computations." [5, p. 1]

In Genkin et al. [5] they attack GNUPG an open source software used to perform asymmetric cryptographic operations. It uses Karatsuba multiplication algorithm which is more resistant to timing based side channels. However, it is vulnerable to acoustic side channel leakage. It is possible to recover the private key from a distance of 4 meters using a parabolic microphone. [5, cf. p. 12]

E. Electromagnetic side channel attack

Sound is not the only side channel that does require access to the device performing the cryptographic operation. Electromagnetic radiation can be used as well to recover secret key

material. Researchers have shown that with common measurement equipment an attack can be successful within seconds requiring only 16 ciphertexts. [4, cf. p. 4]

V. Quantum computing theory

Traditional computers use classical bits to store information and to use them in computation. These bits can be in exactly one of two states 0 or 1. There is no other option. This discreteness provides the advantage of being implementable with readily available components such as integrated transistors. And while machines build to use bits can be Touring complete there are certain computational problems that can be solved faster using algorithms involving quantum bits or more commonly known Qubits.

This section will serve as a general introduction into quantum computing and how it effects the way one can solve problems.

A. Quantum bits

A Qubit is defined as $\alpha \cdot (1\ 0)^T + \beta \cdot (0\ 1)^T$ where $|\alpha|^2 + |\beta|^2 = 1$ and $\alpha, \beta \in \mathbb{C}$. It shows how the state of the Qubit is made up from two vectors. $(1\ 0)^T = |0\rangle$ is the 0 part of the Qubit, with $|\alpha|^2$ describing how likely it is to measure a 0 when observing the Qubit. In contrast to this $(0\ 1)^T = |1\rangle$ is the 1 part of the Qubit, with $|\beta|^2$ describing how likely it is to measure a 1 when observing the Qubit.

This definition allows for the Qubit to assume a superposition where it is possible to either measure a 0 or a 1. E.g. $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is a Qubit state that has equal chance to be measured as 0 or a 1. The superposition collapses and any subsequent measurements will read the same value. The factor of $\frac{1}{\sqrt{2}}$ is necessary to keep the sum of the probabilities to 1. Note that measuring a Qubit destroys its internal state. It is also not possible to copy an Qubits internal state which prevents one from just copying the Qubit and measuring one while keeping one copy intact.

The values of $|\alpha|^2$ and $|\beta|^2$ can be estimated by running the same quantum circuit multiple times and creating a statistic. Figure 5 shows a box diagram plotting the count of $|0\rangle$ and $|1\rangle$ by simulating a circuit, creating a superposition, and then measuring the Qubit (Figure 4). Listing 5 shows the required Python code to perform this simulation. The instructions to install the necessary software can be found here: https://docs.quantum.ibm.com/start/install

By using complex exponential form for α and β another core property of Qubits can be shown. Reforming the Qubit definition yields: $\alpha|0\rangle + \beta|1\rangle = e^{i\phi_0}|0\rangle + e^{i\phi_1}|0\rangle = e^{i\phi_0}\left(|0\rangle + e^{i(\phi_1-\phi_0)}|1\rangle\right) = e^{i\phi_G}\left(|0\rangle + e^{i\phi_L}|1\rangle\right)$. ϕ_G is called the global phase and ϕ_L the local phase. The first one is not known to have any effect on the Qubit while the local phase does. However, it is impossible to measure it directly.

B. Quantum gates

To implement algorithms that use the unique properties of Qubits special operators need to be defined. In contrast to the traditional boolean operators for bits like AND, OR and, XOR, operators for Qubits have to be reversible. This means for any output of the gate it has to be clear and unambiguous what input produced it. E.g. the AND gate is not reversible because for an output of 0 there are 3 different possible inputs: $\{(0,0),(0,1),(1,0)\}$.

The quantum operators, often called gates, are defined using matrices that operate on the Qubit. There is a set of standard operators based on the Pauli matrices named after Wolfgang Pauli. There is a total of 3 Pauli gates X, Y and Z. Here only Pauli X will be shown:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

It flips the probabilities for $|0\rangle$ and $|1\rangle$.

$$\sigma_x \cdot (\alpha\,|0\rangle + \beta\,|1\rangle) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} = \beta\,|0\rangle + \alpha\,|1\rangle$$

This is the quantum equivalent for the NOT Gate.

Arguably the most important gate for quantum algorithms is not one of the Pauli gates but rather the Hadamard gate with its matrix being:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

It has the effect of turning a base state Qubit into a superposition:

$$H \cdot |0\rangle = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) = |+\rangle$$

Typically quantum algorithms start with applying a Hadamard gate to all Qubits to turn them into a superposition that then enables the unique properties of quantum computing. The circuit in Figure 4 uses such a gate to to turn the $|0\rangle$ Qubit into a superposition that when measured has equal chance to be a 0 or 1.

C. Combining gates

When constructing more complex circuits using multiple gates these matrices can be combined.

Consecutive gates e.g. $H \to \sigma_x$ are multiplied in inverse order:

$$\sigma_x \cdot H \cdot |q_0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_0 \\ \beta_0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_0 \\ \beta_0 \end{pmatrix}$$

For parallel gates that apply to different Qubits as shown with the Hadamard gate for q_0 and the omitted identity matrix for q_1 in Figure 6 the Tensor product $\otimes$ is used. Given two matrices A and B with the respective dimensions $n_0 \times m_0$ and $n_1 \times m_1$ it is defined as such:

$$A \otimes B = \begin{pmatrix} a_{1,1} & \cdots & a_{1,m_0} \\ \vdots & \ddots & \vdots \\ a_{n_0,1} & \cdots & a_{n_0,m_0} \end{pmatrix} \otimes \begin{pmatrix} b_{1,1} & \cdots & b_{1,m_1} \\ \vdots & \ddots & \vdots \\ b_{n_1,1} & \cdots & b_{n_1,m_1} \end{pmatrix} =$$

$$\begin{pmatrix} a_{1,1} \begin{pmatrix} b_{1,1} & \cdots & b_{1,m_1} \\ \vdots & \ddots & \vdots \\ b_{n_1,1} & \cdots & b_{n_1,m_1} \end{pmatrix} & \cdots & a_{1,m_0} \begin{pmatrix} b_{1,1} & \cdots & b_{1,m_1} \\ \vdots & \ddots & \vdots \\ b_{n_1,1} & \cdots & b_{n_1,m_1} \end{pmatrix} \\ \vdots & \ddots & \vdots \\ a_{n_0,1} \begin{pmatrix} b_{1,1} & \cdots & b_{1,m_1} \\ \vdots & \ddots & \vdots \\ b_{n_1,1} & \cdots & b_{n_1,m_1} \end{pmatrix} & \cdots & a_{n_0,m_0} \begin{pmatrix} b_{1,1} & \cdots & b_{1,m_1} \\ \vdots & \ddots & \vdots \\ b_{n_1,1} & \cdots & b_{n_1,m_1} \end{pmatrix} \end{pmatrix}$$

It is apparent that the resulting matrix has the dimensions $(n_0 \cdot n_1) \times (m_0 \cdot m_1)$. A gate operating on a single Qubit has the dimensions 2×2. Due to how the Tensor product increases the matrix size, operating on n Qubits results in matrix size of $2^n \times 2^n$. This makes it hard to almost impossible to simulate larger actually useful quantum circuits.

D. Quantum entanglement

Another core property of these algorithms is quantum entanglement. It refers to the fact that the state of two Qubits can be connected with each other so that when one quantum state

collapses the other one does so as well.

The simplest way of creating quantum entanglement is to create a Bell-State using a Hadamard and a controlled not (CNOT) gate. The CNOT gate applies a Pauli X gate on the controlled Qubit if the controlling Qubit is 1. Its matrix looks like this:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The algorithm starts by putting the controlling Qubit q_0 into a superposition using a Hadamard gate. Afterwards a CNOT gate is applied with q_0 as the controlling Qubit and q_1 being the controlled Qubit. At this point both Qubits are in superposition because there is no clear way to tell if q_1 was flipped

The math behind this circuit is as follows:

$$\text{CNOT} \cdot (H \otimes I) \cdot (|0\rangle \otimes |0\rangle) = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} =$$

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

The result is that the two Qubits can either be both $|0\rangle$ or $|1\rangle$ but it is impossible for one to be $|0\rangle$ and the other to be $|1\rangle$. If q_0 is $|0\rangle$ q_1 wont be flipped and remain to be $|0\rangle$. In the case q_0 is $|1\rangle$ q_1 will be flipped and turn into a $|1\rangle$ as well.

Note that it does not matter in which order the Qubits were measured. So, if q_1 is first, then the superposition of q_0 will collapse to yield the same value as q_0. This inverse causality of the controlled Qubit influencing the controlling one is referred to as quantum kickback.

Figure 5 shows a simulated measurement of a Bell-State created using the code in Listing 6 that resembles the circuit in Figure 6.

The superposition and quantum entanglement allow quantum computers to run algorithms that are impossible for traditional computers to complete. A quick example is Deutsch's Algorithm. It can determine whether a function $f : 0, 1 \to 0, 1$ is balanced ($f(0) = f(1)$). A traditional computer needs to try both $f(0)$ and $f(1)$ to find out. Deutsch's Algorithm can do the same with just one evaluation of f. However, to find out whether $f(0)$ equals 0 or 1 requires two evaluations with this algorithm so there is a trade of to be made between the classical and the quantum approach.

This algorithm is important in the sense that it was able to solve a problem faster than a traditional method. The following section shows a much more complicated use of quantum computers that solves the factorization of large numbers which consequently breaks the RSA cryptosystem. Current quantum computers still do not have enough Qubits with a low enough error rate to make any of the following possible. So at this point in time there is no threat for long enough key lengths, albeit it is only a matter of time until this will not suffice.

VI. Shor's algorithm

The security of the RSA cryptosystem, as discussed in Section II, depends on the challenge of factorizing large integers, a task that remains computationally hard for classical computers.

However, in 1994, Peter Shor presented an algorithm in his paper "Algorithms for Quantum Computation: Discrete Logarithms and Factoring" [10] that has the power of changing the landscape of cryptographic security. Shor's algorithm exploits the capabilities of quantum computers to efficiently factorize large integers. This ability presents a significant threat to RSA encryption.

The algorithm works as follows: The large number N is the starting point of the algorithm. N is the product of the two prime numbers p and q. The goal of Shor's algorithm is to extract p and q from N. Therefore the algorithm starts by making a random guess g of a number that is smaller than N. Shor's algorithm is able to turn this random guess, that probably doesn't share factors with N, into a pair of improved guesses. Those new guesses indeed share factors with N. With the help of Euclid's algorithm the factors p and q can be deduced from the two guesses [11, p. 15]. It is possible to run Shor's algorithm on a classical computer to factor big numbers. The algorithm doesn't contain anything specific to quantum mechanics, that can't be done on a normal computer. The problem is that the process of turning the bad guess into a pair of better guesses takes a long time on classical computer. On a quantum computer on the other hand this process can be performed really fast.

The next section will explain how Shor's algorithm turns the bad guess into a pair of better guesses, which is pure mathematics. Afterwards the quantum mechanical principles that speed up the process will be discussed.

A. Optimizing guesses

As already explained in the RSA cryptosystem, the security of the encrypted data relies on the inability to factorize the large number N into its prime factors p and q. To break the encryption these prime factors, initially unknown, need to be discovered. However, instead of directly guessing a prime factor of N, it is sufficient to guess a number that shares factors with N. Utilizing Euclid's algorithm, one can then determine the shared factor between N and the guessed number. At the point when one shared factor with N is found, the encryption is effectively broken. Only the division of N by the shared factor needs to be performed to obtain the second prime factor. While this process may seem straightforward, the sheer magnitude of the numbers utilized in today's encryption makes it highly unlikely, that any single guess will share a factor with N.

Shor's algorithm solves this challenge in an ingenious way by using a mathematical insight [10, p. 7]: A pair of numbers A and B that don't share factors is given. If the first number A is multiplied by itself enough times it will be equal to a multiple of B plus 1. This leads to the equation:

$$A^p \equiv m \cdot B + 1 \equiv 1 \mod B \tag{6}$$

This pivotal mathematical insight forms the foundation of Shor's algorithm. It is the reason for the effectiveness in breaking the RSA encryption for large composite numbers. To prove this observation it would take a long time, but it can be shown with a simple example. Let $A = 5$ and $B = 91$. The results are shown in the table 1.

This principle applies to any pair of numbers (A, B) that are coprime. Shor's algorithm makes use of this observation. Initially, a random number g with $g < N$ is selected [11, p. 15]. Because of the mathematical insight from Equation 6 and Table 1, it is certain that, there is a p that fulfills the equation,

$$g^p \equiv m \cdot N + 1 \tag{7}$$

A^x	Result	$A^x/91$	Remainder
A^1	5	0	5
A^2	125	0	25
A^3	625	1	34
A^4	3125	6	79
A^5	15625	34	31
A^6	78125	171	64
A^7	390625	858	47
A^8	1953125	4292	53
A^9	9765625	21462	83
A^{10}	48828125	107314	51
A^{11}	244140625	536572	73
A^{12}	1220703125	2682864	1

Figure 1: Remainder of $5^x/91$

Equation 7 can be rearranged to equation 8,

$$g^p - 1 \equiv m \cdot N \tag{8}$$

Equation 8 can also be written as:

$$(g^{p/2} + 1) \cdot (g^{p/2} - 1) \equiv m \cdot N \tag{9}$$

The terms on the left-hand side of equation 9 precisely represent the improved guesses for the factors of N. These terms are likely multiples of factors of N. However, this is no problem, as it is possible to get the factors of N using Euclid's algorithm [11, p. 15]. This method is now used in the previous example with $g = 5$ and $N = 91$: $5^{12/2} + 1 = 156265^{12/2} - 1 = 15624$ Both of the results 15626 and 15624 are not factors of $B = 91$. But the results share factors with 91. 15626 and 91 share the factor 13 and 15624 shares the factor 7 with 91. To identify the shared factors Euclid's algorithm is used. It would be sufficient to only calculate one of the factors, since one can get the second factor by dividing 91 by the first found factor. At this point the encryption is broken since both prime factors of N are found.

However the following three problems can occur with these new and improved guesses [11, p. 15]:

1) One of the new guesses happens to be a multiple of N. This means the other guess would be a factor of m. Both guesses are useless in obtaining the prime factors.
2) The power p is an odd number. Then $p/2$ does not yield a whole number. The result would be that the new guesses are also no whole numbers and therefore useless.
3) The computation of the power p on a classical computer is time-consuming and inefficient. The process is impractical for classical systems.

For an initial random guess, it has been observed that neither problem one nor problem two occur in 37.5% of cases [11, p. 15][10, p. 7]. This means the process is worth repeating. With fewer than 10 random guesses, the likelihood of discovering the prime factors of N exceeds 99%. The remaining challenge lies in accelerating the computation of p. This is where the quantum aspect of Shor's algorithm comes into effect.

B. Accelerating the computation with quantum computation

Unlike classical computations, which yield only a single answer for a given input, quantum computations can simultaneously compute numerous potential possible answers for a single input by using a quantum superposition. However, when measuring the superposition, only

one answer is randomly returned, with each potential answer having different probabilities. Quantum superposition is explained in section V. The key behind fast and reliable quantum computations is to setup a quantum superposition that calculates all possible answers at once, while being cleverly arranged, so that all the wrong answers destructively interfere with each other. This ensures that upon measurement, the most probable outcome is the correct answer. Formulating a problem into a quantum format so that all the wrong answers destructively interfere with each other poses a huge challenge. But this is exactly what Shor's algorithm does when searching for p [11, p. 14].

First the Qubits are split up into two sets. The first set is prepared in a superposition of:

$$|x\rangle = |0\rangle + |1\rangle + |2\rangle + |3\rangle + |4\rangle + \cdots + |10^{1234}\rangle$$

This is a huge superposition. But with perfect Qubits only around 4100 Qubits would be required. The other set contains around 2050 Qubits that are all left in the 0-State [11, p. 15]:

$$|w\rangle = |0\rangle|0\rangle|0\rangle|0\rangle|0\rangle...|0\rangle$$

Now guess g, which most likely doesn't share factors with N, is raised to the power of the first set of Qubits and divide by N.

$$|g^0/N\rangle, |g^1/N\rangle, |g^2/N\rangle, |g^3/N\rangle, |g^4/N\rangle, ..., |g^{10^{1234}}/N\rangle$$

The remainder r is stored in the second set of Qubits [11, p. 16].

$$|rem_0\rangle, |rem_1\rangle, |rem_2\rangle, |rem_3\rangle, |rem_4\rangle, ...$$

The first set of Qubits is left as it was. Now there is a superposition with all the numbers that g was raised to and another superposition with the remainder $rem = g^x/N$. Through this operation the two sets of Qubits are entangled with each other:

$$|0\rangle|rem_0\rangle + |1\rangle|rem_1\rangle + |2\rangle|rem_2\rangle + |3\rangle|rem_3\rangle + |4\rangle|rem_4\rangle + ...$$

Measuring this superposition directly would yield a single random element from the superposition. Such an approach would offer no improvement over randomly guessing powers, a task achievable with a classical computer. To achieve meaningful results, a more sophisticated strategy is required. A strategy that ensures that all non-p answers destructively interfere and nullify each other, leaving behind only the desired answer, p.

This goal is achieved by using another crucial mathematical insight. The graph depicted in Figure 2 illustrates the values of the remainder of 5^x. An important observation emerges: the property $A^p = m * B + 1$ has a periodicity. In this case the remainder $r = 1$ emerges with a periodicity of $p = 12$. But also all the other remainders emerge with this periodicity of $p = 12$.

The graph in Figure 2 revealed that the property $A^p \equiv m \cdot B + 1$ has a periodicity. Specifically, for each exponent, there exists a period p during which the same remainder consistently arises.

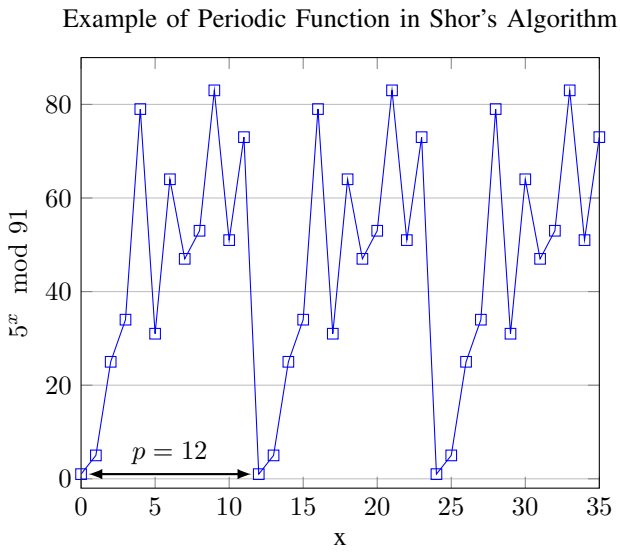

Figure 2: Periodic Function Graph

Applying this insight for the guess g and the product N:

$$g^x = m_1 * N + r$$
$$g^{x+p} = m_2 * N + r$$
$$g^{x+2p} = m_3 * N + r$$
$$...$$

This means that the power p that is searched for to break encryption has a repeating property. When considering another power and adding or subtracting p from it, the resulting surplus over a multiple of N, called r, remains consistent [11, p. 15].

$$g^x \to +r$$
$$g^{x+p} \to +r$$
$$g^{x-p} \to +r$$
$$g^{x+2p} \to +r$$

This repeating property can't be found by examining the guess for just a single power. It represents a structural relationship between different powers. However, this inherent relationship can be taken advantage of, as quantum computations can operate on superpositions of different possible powers.

$$g^2 \to +8$$
$$g^{12} \to +8$$
$$g^{22} \to +8$$

This knowledge can be used when measuring the superposition

$$|1, +17\rangle + |2, +5\rangle + |3, +92\rangle + ...$$

Upon measuring solely the r component, a random r value is obtained, such as $r = 8$. While the specific number holds no significance, what is crucial is that the quantum computer enters

a superposition exclusively comprising powers that yield the same r [11, p. 16]. The resulting superposition takes the form:

$$|2, +8\rangle + |12, +8\rangle + |22, +8\rangle + ...$$

This property is unique to quantum computations: when a superposition is used as the input and a result is measured that could have originated from multiple elements, the output becomes a superposition containing all those elements associated with the measured result [11, p. 16]. In this scenario, due to the repeating property, these elements consist of powers spaced p units apart. Consequently, the quantum computer enters a superposition of numbers repeating with a frequency of $f = 1/period = 1/p$. To obtain the desired power p for decrypting the encryption, the frequency f needs to be found. The most effective approach for identifying frequencies is the *Fourier Transform*. There is also a quantum version applicable to the superposition repeating with the frequency $f = 1/p$ [11, p. 16]. The Quantum Fourier transformation causes destructive interference between all existing frequencies, so that only a single quantum state remains [11, p. 16]: the frequency $f = 1/p$. This frequency can then be measured to yield the actual output of the computation: $f = 1/p$. This outcome can be inverted to derive p.

If p is even, equation 9 can be used to transform the initial guess g into a pair of improved guesses. If these new guesses are not multiples of N, it is ensured that the new pair of guesses shares factors with N. Consequently, Euclid's algorithm can be used to obtain the shared factors with N, thereby breaking the encryption. The encrypted data can now be successfully decrypted. When looking at the steps taken to find p, it becomes evident that the challenge of factorization has been transformed to a period-finding problem. Remarkably, the period-finding problem can be efficiently solved using quantum computations, achieving polynomial time complexity [11, p. 14].

C. Implementation of algorithm

Now with the understanding of the functionality of Shor's Algorithm, it is time to implement Shor's Algorithm. IBM Qiskit is selected for the implementation, so that the code can be run on a real Quantum Computer. The paper "Circuit for Shor's algorithm using 2n+3 qubits" [1] and the Youtube video of the IBM chanel titled Shor's Algorithm — Programming on Quantum Computers — Coding with Qiskit S2E7 served as references. The initial stage of the program requires the imports in Listing 7. Furthermore the connection to the IBM Quantum Computer is set.

As already explained, the problem of factorization can be reinterpreted as a problem of period finding. In this example, the period finding problem for $g = 13$ and $N = 15$ is solved. To accomplish this the modular exponentiation function, c_amod15 is used. This function yields the controlled-U gate for g, iterated $power$ times. The function is hardcoded for factorizing $N = 15$ and the guess $g = 13$.

```
1  def c_amod15(g, power):
2      """Controlled multiplication by g mod 15"""
3      if g != 13:
4          raise ValueError("'g' must be 13")
5      U = QuantumCircuit(4)
6      for iteration in range(power):
7          U.swap(0,1)
8          U.swap(1,2)
```

```
9           U.swap(2,3)
10          for q in range(4):
11              U.x(q)
12      U = U.to_gate()
13      U.name = "%i^%i mod 15" % (g, power)
14      c_U = U.control()
15      return c_U
```

Listing 3: Modular exponentiation function

Additionally, the Quantum Fourier Transform circuit needs to be implemented as follows:

```
1  def qft_dagger(n):
2      """n-qubit QFTdagger the first n qubits in circ"""
3      qc = QuantumCircuit(n)
4      for qubit in range(n//2):
5          qc.swap(qubit, n-qubit-1)
6      for j in range(n):
7          for m in range(j):
8              qc.cp(-np.pi/float(2**(j-m)), m, j)
9          qc.h(j)
10     qc.name = "QFTdagger"
11     return qc
```

Listing 4: Quantum-Fourier-Transform Circuit

Using these building blocks, constructing the circuit for Shor's algorithm becomes straightforward. The complete quantum circuit is shown in Figure 3.

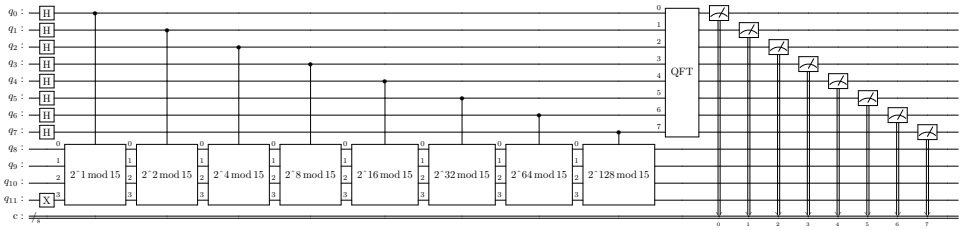

Figure 3: Quantum Circuit Shor's Algorithm

With the quantum circuit in place Shor's Algorithm can be realised. The function qpe_amod15(g) puts everything together. It performs Shor's algorithm on a guess g that gets passed to the function as an argument. Initially, the number of qubits $n_{count} = 8$ is defined (line 2). Subsequently, the quantum circuit and qubits are initialized. Next, the guess g is exponentiated using the quantum circuit for the Modular exponentiation function. Following this, the Quantum Fourier Transform is executed on the qubits. The qubits are then measured to obtain the result of the computation. The IBMQ_16_Melbourne Quantum Computer is used to execute the computation. Finally, the register and corresponding phase are printed. As a result the function returns the corresponding phase s/p. The code is shown in Listing 8.

Quantum computations may yield incorrect results. This makes it necessary that multiple iterations need to be run to obtain the desired outcome. That's why in some cases the computation needs to be repeated a few times to get the desired result. The following code snippet in Listing 9 implements a loop that checks the results of the quantum computation

and repeats it until a satisfactory result is achieved. The function qpe_amod15(g) returns the phase s/p that was calculated by the quantum circuit, displayed in Figure 3. The period p - the magic number to break the encryption - is derived from the phase in line 12 and 13. Using the period p the prime factors are calculated. When discovering at least one non-trivial factor of N, the factor is printed, and the loop terminates.

With the code implemented in Listing 9, the algorithm is now ready to execute on a real quantum computer to factor $N = 15$ with the guess $g = 13$. Qiskit dynamically allocates the necessary qubits for the program, typically queuing the program until the required qubits become available. Qubits vary in quality, with production being both challenging and costly. Imperfections in some qubits may lead to inaccuracies in results. Higher-quality qubits entail longer waiting times. For the specific task of period finding in this program, qubit quality is of minor importance since the program is primarily for demonstration purposes. Hence, lower-quality qubits were selected to minimize waiting times. When the program runs, it successfully calculates the prime factors of 15. Typically, only one iteration of the loop in Listing 9 is needed to identify at least one prime factor, though occasionally, two to three iterations may be necessary.

VII. Conclusion

While most of the attacks detailed in this paper can be mitigated in practice there is still room for exploitation. First and foremost Shor´s algorithm is slowly approaching practicality with Qubits getting less and less noisy as well as greater in count for a single quantum computer. At the point where this happens all RSA private keys of insufficient length are vulnerable and have to be considered leaked. Of cause one could always ever larger RSA keys but at some point using them becomes inpractical.

As of today the internet still relies heavily on RSA to secure the communication from clients to servers. Moving these parts over to post-quantum algorithms that are resistant to attacks taking advantage of quantum algorithms will be major challenge and has to be complete before Shor starts breaking actual usages.

On the other hand software and users are not perfect. Flaws in the implementation of RSA or its configuration can break it even in the absence of Shor. And as cryptographic code is often reuse to prevent the same basic mistakes from happening over and over again this means that a issue in such a code base could affect a lot of users.

To prevent a single flaw from compromising a whole product indefinitely one has to be cryptoagnostic which means being able to change out the cryptographic algorithm if the needed arises without too much work.

VIII. Appendix

A. Large code examples

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram

qc = QuantumCircuit(1,1)

qc.h(0)

qc.measure_all(add_bits=False)
qc.draw()

simulator = AerSimulator()
result = simulator.run(qc, shots=10**4).result()
counts = result.get_counts(qc)
plot_histogram(counts, title='Superposition counts')
```

Listing 5: Code to simulate superposition

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram

qc = QuantumCircuit(2,2)

qc.h(0)
qc.cx(0, 1)
qc.measure_all(add_bits=False)

qc.draw()

simulator = AerSimulator()
result = simulator.run(qc, shots=10**4).result()
counts = result.get_counts(qc)
plot_histogram(counts, title="Bell-State counts")
```

Listing 6: Code to simulate Bell-State

```
import matplotlib.pyplot as plt
import numpy as np
from qiskit import QuantumCircuit, compiler
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from qiskit_ibm_runtime import QiskitRuntimeService
from math import gcd
from numpy.random import randint
import pandas as pd
from fractions import Fraction
print("Imports Successful")

# set connection to provider
service = QiskitRuntimeService(channel='ibm_quantum',token="")
backend = service.least_busy(operational=True,
min_num_qubits=2, simulator=False)
```

Listing 7: Import libraries

```
def qpe_amod15(g):
n_count = 8
qc = QuantumCircuit(4+n_count, n_count)
for q in range(n_count):
qc.h(q)     # Initialize counting qubits in state |+>
qc.x(3+n_count) # And auxiliary register in state |1>
for q in range(n_count): # Do controlled-U operations
qc.append(c_amod15(g, 2**q), [q] + [i+n_count
for i in range(4)])
# Do inverse-QFT
qc.append(qft_dagger(n_count), range(n_count))
qc.measure(range(n_count), range(n_count))
# Execute on Quantum Computer
new_qc = compiler.transpile(qc, backend)
job = backend.run(new_qc)
print(job.job_id())
job.wait_for_final_state(timeout=60)
print(f"Done: {job.in_final_state()}")
if job.errored():
print(job.error_message())
else:
result = job.result()
print(result.get_counts())
plot_histogram(result.get_counts(),
title="Bell-State counts")
readings = result.get_memory()
print("Register Reading: " + readings[0])
phase = int(readings[0],2)/(2**n_count)
print("Corresponding Phase: %f" % phase)
return phase
```

Listing 8: Function implementing Shor's Algorithm

```
# Specify variables
g = 13
N = 15
factor_found = False
attempt = 0

while not factor_found:
attempt += 1
print("\nAttempt %i:" % attempt)
phase = qpe_amod15(g) # Phase = s/p
# Denominator should tell us p
frac = Fraction(phase).limit_denominator(N)
p = frac.denominator
print("Result: p = %i" % p)
if phase != 0:
# Guesses for factors are gcd(x^{p/2} \pm1 , 15)
guesses = [gcd(g**(p//2)-1, N), gcd(g**(p//2)+1, N)]
print("Guessed Factors: %i and %i" % (guesses[0],
guesses[1]))
for guess in guesses:
# Check to see if guess is a factor
if guess not in [1,N] and (N % guess) == 0:
print("*** Non-trivial factor found: %i
***" % guess)
factor_found = True
```

Listing 9: Implemented Shor's algorithm

B. Figures

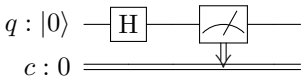

Figure 4: Superposition circuit diagram

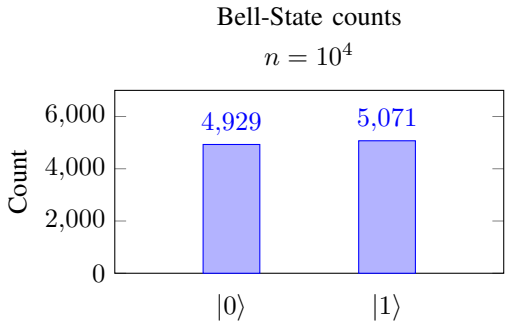

Figure 5: Superposition simulation plot

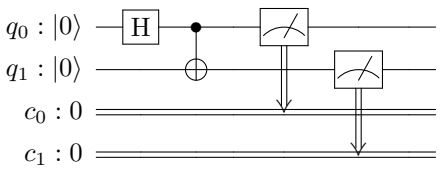

Figure 6: Bell-State circuit diagram

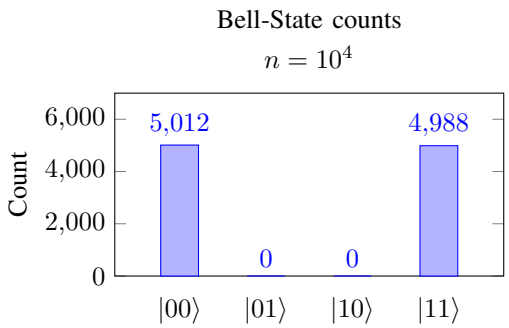

Figure 7: Bell-State simulation plot

C. References

[1] S. Beauregard, "Circuit for shor's algorithm using 2n+ 3 qubits," *arXiv preprint quant-ph/0205095*, 2002.

[2] D. Boneh, "Twenty years of attacks on the rsa cryptosystem," *Notices of the American Mathematical Society*, vol. 46, pp. 203–212, 1999. [Online]. Available: https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf.

[3] "Bsi – technical guideline," Federal Office for Information Security, P.O.B. 20 03 63, 53133 Bonn, Germany, Tech. Rep. BSI TR-02102-1, Feb. 2024. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile&v=7.

[4] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, *Stealing keys from pcs using a radio: Cheap electromagnetic attacks on windowed exponentiation*, Cryptology ePrint Archive, Paper 2015/170, https://eprint.iacr.org/2015/170, 2015. [Online]. Available: https://eprint.iacr.org/2015/170.

[5] D. Genkin, A. Shamir, and E. Tromer, *Rsa key extraction via low-bandwidth acoustic cryptanalysis*, Cryptology ePrint Archive, Paper 2013/857, https://eprint.iacr.org/2013/857, 2013. [Online]. Available: https://eprint.iacr.org/2013/857.

[6] E. Milanov, "The rsa algorithm," *RSA laboratories*, pp. 1–11, 2009.

[7] T. D. Nguyen, T. D. Nguyen, and L. D. Tran, "Attacks on low private exponent rsa: An experimental study," in *2013 13th International Conference on Computational Science and Its Applications*, Jun. 2013, pp. 162–165. DOI: 10.1109/ICCSA.2013.32.

[8] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[9] K. Ryan, K. He, G. A. Sullivan, and N. Heninger, "Passive ssh key compromise via lattices," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 2886–2900, ISBN: 979-8-40070-050-7. DOI: 10.1145/3576915.3616629. [Online]. Available: https://doi.org/10.1145/3576915.3616629.

[10] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[11] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, ISSN: 1095-7111. DOI: 10.1137/s0097539795293172. [Online]. Available: http://dx.doi.org/10.1137/S0097539795293172.

[12] W. P. Wardlaw, "The rsa public key cryptosystem," in *Coding Theory and Cryptography: From Enigma and Geheimschreiber to Quantum Theory*, Springer, 2000, pp. 101–123.

[13] B. Zhao, L. Wang, K. Jiang, X. Liang, W. Shan, and J. Liu, "An improved power attack on small rsa public exponent," in *2016 12th International Conference on Computational Intelligence and Security (CIS)*, Dec. 2016, pp. 578–581. DOI: 10.1109/CIS.2016.0140.