

# Draft Report

## Quantum-Safe Cryptography: Mitigating Vulnerabilities in Post-Quantum era

Unit: COIT20265  
Student 1: Ayush Keshar Prasai  
Student 2: Jalay shah  
Student 3: Ronit Maheshwori  
Student 4: Virajsinh jeetendra sinh rahevar  
Project Mentor: Mohammad Mohammad  
Date: 26/08/2024

CQUniversity Australia

### 1 Introduction

This report conceptualizes advancement from modern era when classical computers are considered rudimentary and a thing of the past. Such era in this report will often be signified as post-quantum era. Post-quantum era refers to the technological advancement in computational ability that any form of security relied upon in modern times through calculation from classical computers are now vulnerable.

To understand post-quantum cyptography, let us first dive into cyptography today. Since the birth of internet, it can be argued that there are equal number of exceptional practitioners that have used the combination of hardware and software to protect and attack the transmitted data in various ways. There is this thin thread of security that has revolutionized civilization which depends upon standards, algorithms and combination of 0's and 1's. it is due to this standard which everyone follows, algorithms that is calculated and 0's and 1's that are intrepeted in specific manner from which confidentiality, integrity and availability is provided to the data over the internet or anywhere else.

Up until now, cryptography is practised when algorithms are enforced and standards are maintained. Post-quantum era now imagines an edge in this constant battle between the attacker and the protector through means of exceptionally powerful hardware. Thus, quantum-safe cryptography refers to post-quantum era where standards, algorithms and interpretation of data is practised with depiction of ability of quantum computers. This report focuses primarily on finding vulnerabilities in modern day cyptography from post-quantum era and mitigating such

vulnerabilities for quantum-safe cryptography. Furthermore, this report studies vulnerabilities of algorithms used for cryptography today and assesses its risk to post-quantum cryptography.

## 2 System Overview

The system in process of being designed consists of following parts:

1. Technical description of RSA and justification of why RSA has been considered vulnerable.

**Table 1 - Impact of Quantum Computing on Common Cryptographic Algorithms**

Cryptographic Algorithm	Type	Purpose	Impact from large-scale quantum computer
AES	Symmetric key	Encryption	Larger key sizes needed
SHA-2, SHA-3	-----	Hash functions	Larger output needed
RSA	Public key	Signatures, key establishment	No longer secure
ECDSA, ECDH (Elliptic Curve Cryptography)	Public key	Signatures, key exchange	No longer secure
DSA (Finite Field Cryptography)	Public key	Signatures, key exchange	No longer secure

*Figure 1: (Chen et al., 2016) impact on algorithms*

### Why RSA amongst other asymmetric cyptography algorithms?

Symetric cryptography:

All symetric cyptography is quantum safe. Symetric algorithms such as SHA and AES do not rely upon mathematical calculations making it quantum safe(Azure, 2024).

Asymetric cryptography

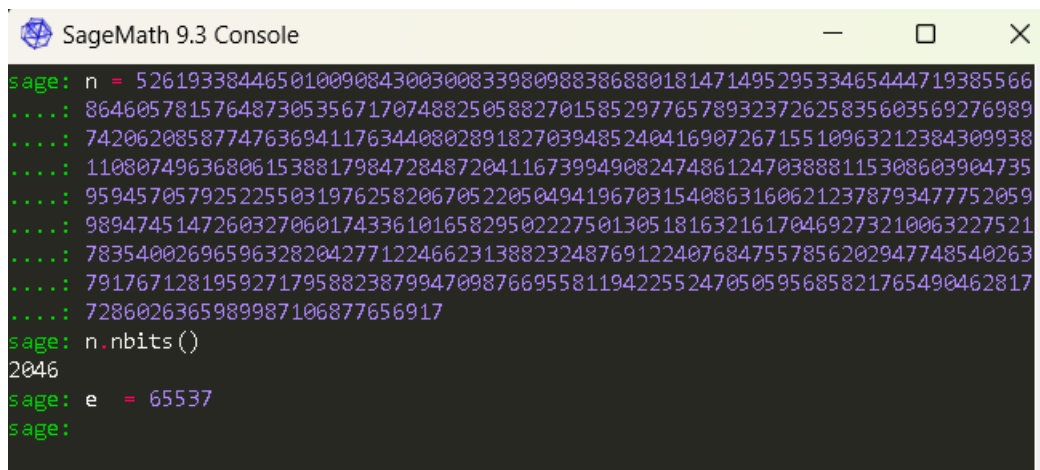
RSA is a asymmetric cryptography algorithm consisting of pair of keys i.e. public key and private key. The keys are generated with cryptographic algorithms that rely on mathematical calculations based on one-way functions thus making it vulnerable to quantum cryptography. RSA amongst other asymmetric algorithms is chosen in modern day cryptography for its ability to provide key distribution and secrecy along with digital signatures which reduces the use of multiple asymmetric algorithms.

### Vulnerability assessment of RSA:

vulnerability assessment of RSA is depicted on technical Artefact PG15-Algo-vulnerability.docx where RSA and its vulnerabilities to different post-quantum algorithms are listed in a table.

2. Brute force attack simulating a post-quantum calculation through fermat's factorization algorithm on RSA to find the private-key from public key.

PG15-fermat-Algo.docx has detailed description along with codes using python and sagemath to showcase the breaking of RSA to figureout the private-key. Post-quantum computational ability is depicted here on a classical computer by limiting the calculation done by choosing the prime number  $n$  and  $e$  specifically.

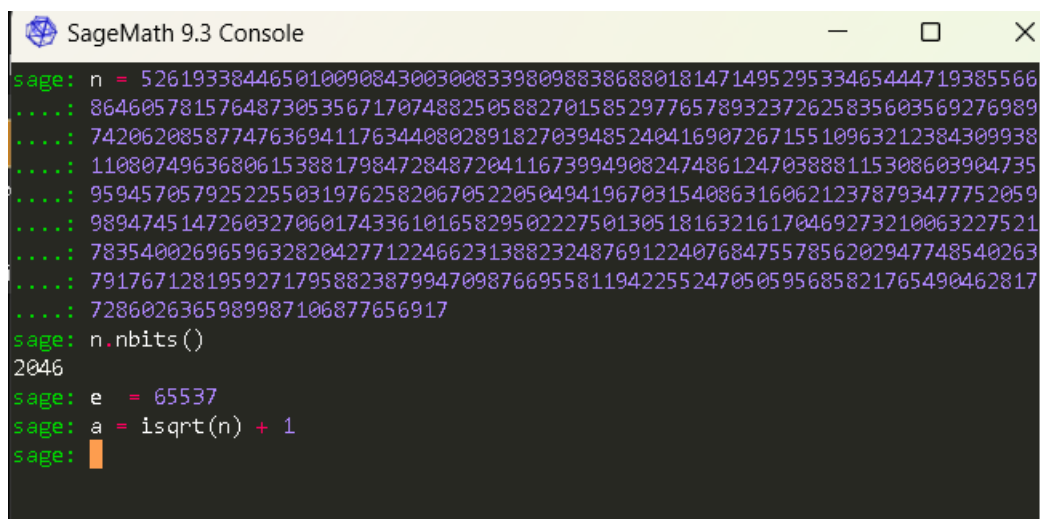


```

sage: n = 5261933844650100908430030083398098838688018147149529533465444719385566
.....: 86460578157648730535671707488250588270158529776578932372625835603569276989
.....: 74206208587747636941176344080289182703948524041690726715510963212384309938
.....: 11080749636806153881798472848720411673994908247486124703888115308603904735
.....: 95945705792522550319762582067052205049419670315408631606212378793477752059
.....: 98947451472603270601743361016582950222750130518163216170469273210063227521
.....: 78354002696596328204277122466231388232487691224076847557856202947748540263
.....: 79176712819592717958823879947098766955811942255247050595685821765490462817
.....: 7286026365989987106877656917
sage: n.nbits()
2046
sage: e = 65537
sage:

```

Figure 2: public key pair  $(e,n)$



```

sage: n = 5261933844650100908430030083398098838688018147149529533465444719385566
.....: 86460578157648730535671707488250588270158529776578932372625835603569276989
.....: 74206208587747636941176344080289182703948524041690726715510963212384309938
.....: 11080749636806153881798472848720411673994908247486124703888115308603904735
.....: 95945705792522550319762582067052205049419670315408631606212378793477752059
.....: 98947451472603270601743361016582950222750130518163216170469273210063227521
.....: 78354002696596328204277122466231388232487691224076847557856202947748540263
.....: 79176712819592717958823879947098766955811942255247050595685821765490462817
.....: 7286026365989987106877656917
sage: n.nbits()
2046
sage: e = 65537
sage: a = isqrt(n) + 1
sage:

```

Figure 3: ceiling function  $a$

```

sage: e = 65537
sage: a = isqrt(n) + 1
sage: while True:
.....:     b2 = a^2 - n
.....:     if is_square(b2):
.....:         b = sqrt(b2)
.....:         break
.....:     a = a + 1
.....:
sage: a
72539188337409048434517657668785982436503618029818802387833126880251213106684983
30184745928175617387284965598034198343521347625158194125197938571884477981179390
35054477893594270832856191204357087252629381536836540664036359542928831056089039
36926001552609343022442053757361595080026446437841528409159732216549

```

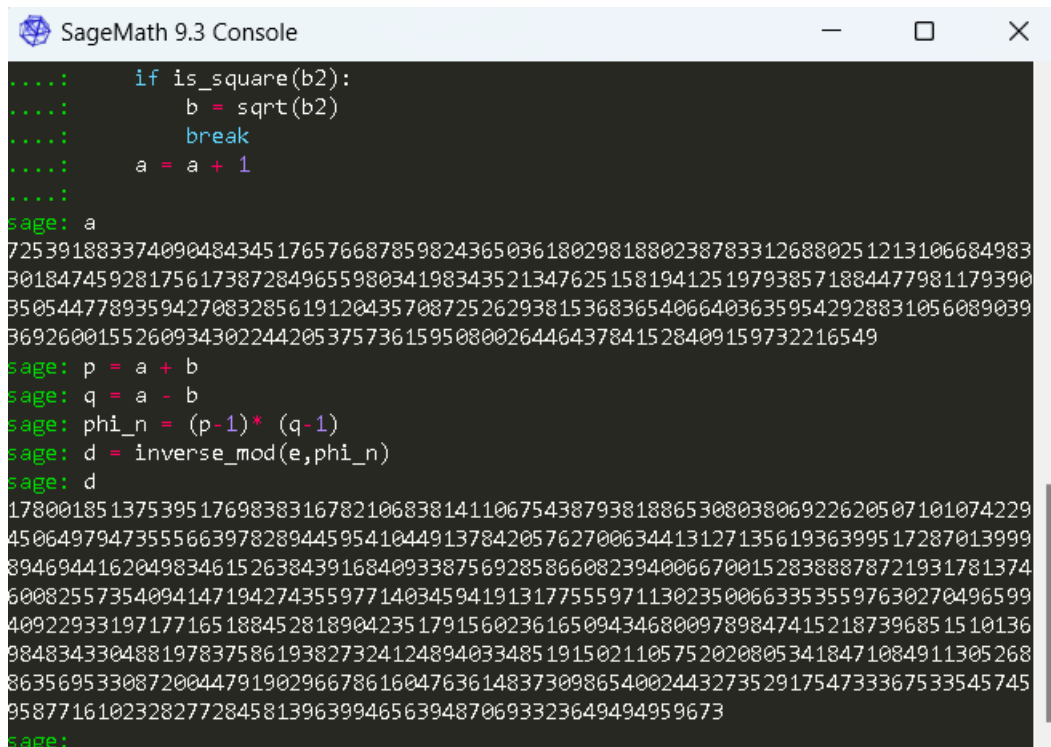
Figure 4: loop that rounds the value of  $a$  to the nearest integer until  $b$  is found

```

.....: 95945705792522550319762582067052205049419670315408631606212378793477752059
.....: 98947451472603270601743361016582950222750130518163216170469273210063227521
.....: 78354002696596328204277122466231388232487691224076847557856202947748540263
.....: 79176712819592717958823879947098766955811942255247050595685821765490462817
.....: 7286026365989987106877656917
sage: n.nbits()
2046
sage: e = 65537
sage: a = isqrt(n) + 1
sage: while True:
.....:     b2 = a^2 - n
.....:     if is_square(b2):
.....:         b = sqrt(b2)
.....:         break
.....:     a = a + 1
.....:
sage: a
72539188337409048434517657668785982436503618029818802387833126880251213106684983
30184745928175617387284965598034198343521347625158194125197938571884477981179390
35054477893594270832856191204357087252629381536836540664036359542928831056089039
36926001552609343022442053757361595080026446437841528409159732216549
sage: p = a + b
sage: q = a - b
sage:

```

Figure 5: value of  $p$  and  $q$  where  $n = p * q$



```

SageMath 9.3 Console
.....:   if is_square(b2):
.....:       b = sqrt(b2)
.....:       break
.....:       a = a + 1
.....:
sage: a
72539188337409048434517657668785982436503618029818802387833126880251213106684983
30184745928175617387284965598034198343521347625158194125197938571884477981179390
35054477893594270832856191204357087252629381536836540664036359542928831056089039
36926001552609343022442053757361595080026446437841528409159732216549
sage: p = a + b
sage: q = a - b
sage: phi_n = (p-1)*(q-1)
sage: d = inverse_mod(e,phi_n)
sage: d
17800185137539517698383167821068381411067543879381886530803806922620507101074229
45064979473555663978289445954104491378420576270063441312713561936399517287013999
89469441620498346152638439168409338756928586608239400667001528388878721931781374
60082557354094147194274355977140345941913177555971130235006633535597630270496599
40922933197177165188452818904235179156023616509434680097898474152187396851510136
98483433048819783758619382732412489403348519150211057520208053418471084911305268
86356953308720044791902966786160476361483730986540024432735291754733367533545745
95877161023282772845813963994656394870693323649494959673
sage:

```

Figure 6: euler totient to find n and n to find d(private key)

3. Demonstration of use of Quantum Computing to leverage Shor's algorithm for cracking RSA Encryption.

Here we demonstrate that RSA encrypted message can be decrypted using Shor's algorithm (Smaranjithose, 2020).

Calculating Modular Inverse:

```

def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return -1

```

Checking for primality:

```

def isprime(n):
    if n < 2:
        return False
    elif n == 2:
        return True
    else:
        for i in range(1, int(sqrt(n)) + 1):
            if n % i == 0:
                return False
    return True

```

Generating Key value pairs:

```
def generate_keypair(keysize):
    p = randint(1, 1000)
    q = randint(1, 1000)
    nMin = 1 << (keysize - 1)
    nMax = (1 << keysize) - 1
    primes = [2]
    start = 1 << (keysize // 2 - 1)
    stop = 1 << (keysize // 2 + 1)
    if start >= stop:
        return []
    for i in range(3, stop + 1, 2):
        for p in primes:
            if i % p == 0:
                break
        else:
            primes.append(i)
    while (primes and primes[0] < start):
        del primes[0]
    # Select two random prime numbers p and q
    while primes:
        p = random.choice(primes)
        primes.remove(p)
        q_values = [q for q in primes if nMin <= p * q <= nMax]
        if q_values:
            q = random.choice(q_values)
            break
    # Calculate n
    n = p * q
```

```
        break
    # Calculate n
    n = p * q
    # Calculate phi
    phi = (p - 1) * (q - 1)
    # Select e
    e = random.randrange(1, phi)
    g = gcd(e, phi)
    # Calculate d
    while True:
        e = random.randrange(1, phi)
        g = gcd(e, phi)
        d = mod_inverse(e, phi)
        if g == 1 and e != d:
            break

    return ((e, n), (d, n))
```

Encryption step:

```
def encrypt(plaintext, package):
    e, n = package
    ciphertext = [pow(ord(c), e, n) for c in plaintext]
    return ''.join(map(lambda x: str(x), ciphertext)), ciphertext
```

Decyption step:

```
def decrypt(ciphertext, package):
    d, n = package
    plaintext = [chr(pow(c, d, n)) for c in ciphertext]
    return ''.join(plaintext)
```

## Testing the sample message:

Generating keys:

```
import RSA_module

bit_length = int(input("Enter bit length: "))

public_k, private_k = generate_keypair(2**bit_length)
```

Encryption:

```
plain_txt = input("Enter a message: ")
cipher_txt, cipher_obj = encrypt(plain_txt, public_k)

print("Encrypted message: {}".format(cipher_txt))
```

Decryption:

```
print("Decrypted message: {}".format(decrypt(cipher_obj, private_k)))
```

## Framing shor's algorithm:

```

qasm_sim = qiskit.Aer.get_backend('qasm_simulator')
def period(a,N):

    available_qubits = 16
    r=-1

    if N >= 2**available_qubits:
        print(str(N)+' is too big for IBMQX')

    qr = QuantumRegister(available_qubits)
    cr = ClassicalRegister(available_qubits)
    qc = QuantumCircuit(qr,cr)
    x0 = randint(1, N-1)
    x_binary = np.zeros(available_qubits, dtype=bool)

    for i in range(1, available_qubits + 1):
        bit_state = (N%(2**i)!=0)
        if bit_state:
            N -= 2**(i-1)
            x_binary[available_qubits-i] = bit_state

    for i in range(0,available_qubits):
        if x_binary[available_qubits-i-1]:
            qc.x(qr[i])
    x = x0

```

```

x = x0

while np.logical_or(x != x0, r <= 0):
    r+=1
    qc.measure(qr, cr)
    for i in range(0,3):
        qc.x(qr[i])
    qc.cx(qr[2],qr[1])
    qc.cx(qr[1],qr[2])
    qc.cx(qr[2],qr[1])
    qc.cx(qr[1],qr[0])
    qc.cx(qr[0],qr[1])
    qc.cx(qr[1],qr[0])
    qc.cx(qr[3],qr[0])
    qc.cx(qr[0],qr[1])
    qc.cx(qr[1],qr[0])

    result = execute(qc,backend = qasm_sim, shots=1024).result()
    counts = result.get_counts()

    results = [],[]
    for key,value in counts.items():
        results[0].append(key)
        results[1].append(int(value))
    s = results[0][np.argmax(np.array(results[1]))]
    return r

```



```

def shors_breaker(N):
    N = int(N)
    while True:
        a=randint(0,N-1)
        g=gcd(a,N)
        if g!=1 or N==1:
            return g,N//g
        else:
            r=period(a,N)
            if r % 2 != 0:
                continue
            elif pow(a,r//2,N)==-1:
                continue
            else:
                p=gcd(pow(a,r//2)+1,N)
                q=gcd(pow(a,r//2)-1,N)
                if p==N or q==N:
                    continue
                return p,q

```

```

def modular_inverse(a,m):
    a = a % m;
    for x in range(1, m) :
        if ((a * x) % m == 1) :
            return x
    return 1

```

```

N_shor = public_k[1]
assert N_shor>0,"Input must be positive"
p,q = shors_breaker(N_shor)
phi = (p-1) * (q-1)
d_shor = modular_inverse(public_k[0], phi)

```

```

print('Message Cracked using Shors Algorithm: {}'.format(decrypt(cipher_obj, (d_shor,N_shor))))

```

### 3 Delivered Technical Artefacts

Name	File	Description	PDF?
Algorithm vulnerability assessment	PG15-Algo-vulnerability.docx	Vulnerability Assessment of cyptography algorithms in use today due to mathematical dependency.	Yes
Sagemath Setup Instructions	PG15-Sagemath-setup-instructions.docx	Instructions for deploying the webserver in Azure	No
Fermat's factorization algorithm implemetation	PG15-Fermat-Algo.docx	Detailed description of how a private key can be decrypted from public key pair and factorization of prime number.	Yes
Shor's algorithm decrypting RSA encryption	PG15-Shors-ALGO.ipynb	Source code that demonstartes the shor's algorith and decryption of RSA encrypted message.	No

### 4 Contributions

Student Name	Percent	Summary of Contributions	Technical Lead on Artefacts
Ayush Keshar Prasai	25%	Researched, tested and documented vulnerabilities from the shor's and fermat's algorithm on RSA.	<ul style="list-style-type: none"><li>Fermat's factorization algorithm implemetation</li><li>Shor's algorithm decrypting RSA encryption</li><li>Algorithm vulnerability assessment</li></ul>
Jalay Shah	25%	Researched and proposed algorithms vulnerable to quantum computing.	<ul style="list-style-type: none"><li>Algorithm vulnerability assessment</li></ul>
Ronit Maheshwori	25%	Set up environment to test shor's and grovers algorithm	<ul style="list-style-type: none"><li>Sagemath setup</li><li>Proposed python</li></ul>
Virajsinh Jeetendra Sinh Rahevar	25%	Tested libraries, tools and methodologies essential for building the system.	<ul style="list-style-type: none"><li>Qiskit , numpy, jupiter, python.</li><li>Sagemath setup</li></ul>

### 5 Next steps

Not yet completed

## 6 References

Azure, quantum (2024) *Microsoft, Azure Quantum*. Available at: <https://quantum.microsoft.com/en-us/our-story/quantum-cryptography-overview#:~:text=Most%20commonly%20used%20public%20key,and%20other%20standardization%20bodies%20globally>. (Accessed: 26 August 2024).

Chen, L. *et al.* (2016) *Report on post-quantum cryptography*, CSRC. Available at: <https://csrc.nist.gov/pubs/ir/8105/final> (Accessed: 26 August 2024).

Smaranjitghose (2020) *QUANTUM\_BURGLARY/REQUIREMENTS.TXT at master · smaranjitghose/quantum\_burglary, GitHub*. Available at: [https://github.com/smaranjitghose/quantum\\_burglary/blob/master/requirements.txt](https://github.com/smaranjitghose/quantum_burglary/blob/master/requirements.txt) (Accessed: 26 August 2024).