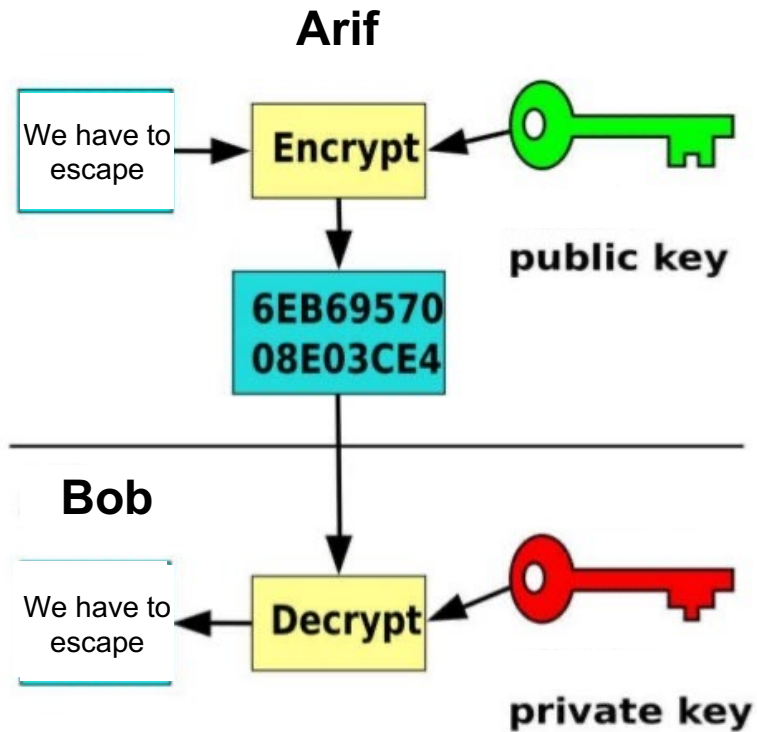
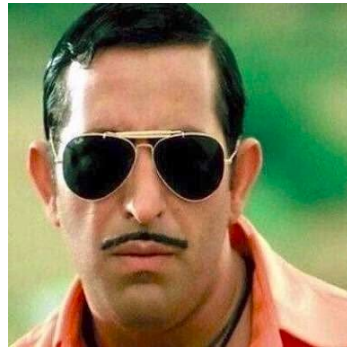




# **RSA decryption with Shor's Algorithm**

Cagil Benibol

# RSA Encryption



# Steps for RSA

1)  $N = p \cdot q$  ( $p, q \in \text{Prime Numbers}$ )

2) Totient Function

$$\lambda(n) = \text{lcm}((p-1)(q-1))$$

3) Choose "e" from  $1 < e < \lambda(n)$   
where e is coprime with  $\lambda(n)$

4) Calculate "d" with extended Euclidean algorithm

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

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

|||

# Extended Euclidean algorithm

```
def eea(a,b):  
    if(a%b==0):  
        return(b,0,1)  
    else:  
        gcd,s,t = eea(b,a%b)  
        s = s-((a//b) * t)  
        print("%d = %d*(%d) + (%d)*(%d)"%(gcd,a,t,s,b))  
        return(gcd,t,s)
```

# Encrypting

- `Public_key = (e,n)`

```
def encrypt(pub_key,message):  
    e,n = pub_key  
    asciiarr = to_ascii(message)  
    crypted_m=[]  
    for i in asciiarr:  
        cm=((int(i)**e) % (n))  
        crypted_m.append(cm)  
    return crypted_m
```

# Decrypting

- Private\_key = (d,n)

```
def decrypt(priv_key,message):  
    d,n = priv_key  
    txt=message.split(',')  
    asciiarr=[]  
    print(txt)  
    for i in txt:  
        dm=((int(i)**d) % (n))  
        asciiarr.append(dm)  
    return from_ascii(asciiarr)
```



# Shor's Algorithm

Fermat's thm.:  $a^p \equiv a \pmod{p} \rightarrow a^{p-1} \equiv 1 \pmod{p}$

$a^r \pmod{N} \equiv 1 \rightarrow$  periodic  
bcs of Fermat's Thm.

Shor's Solution

$$U|y\rangle \equiv |ay \pmod{N}\rangle$$

If we apply  $x$  times: (let's say  $y=1$ )

$$U^x|1\rangle \equiv |a^x \pmod{N}\rangle$$

$$U^{r-1}|1\rangle \equiv |a^{r-1} \pmod{N}\rangle \equiv |1\rangle$$

# Phase Estimation

Known      Estimate

$$U|\psi\rangle = e^{2\pi i \phi} |\psi\rangle$$

$$|\psi_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2\pi i s k}{r}} |a^k \bmod N\rangle$$

$$U|\psi_s\rangle = e^{\frac{2\pi i s}{r}} |\psi_s\rangle \quad \text{where } \phi = \frac{s}{r}$$

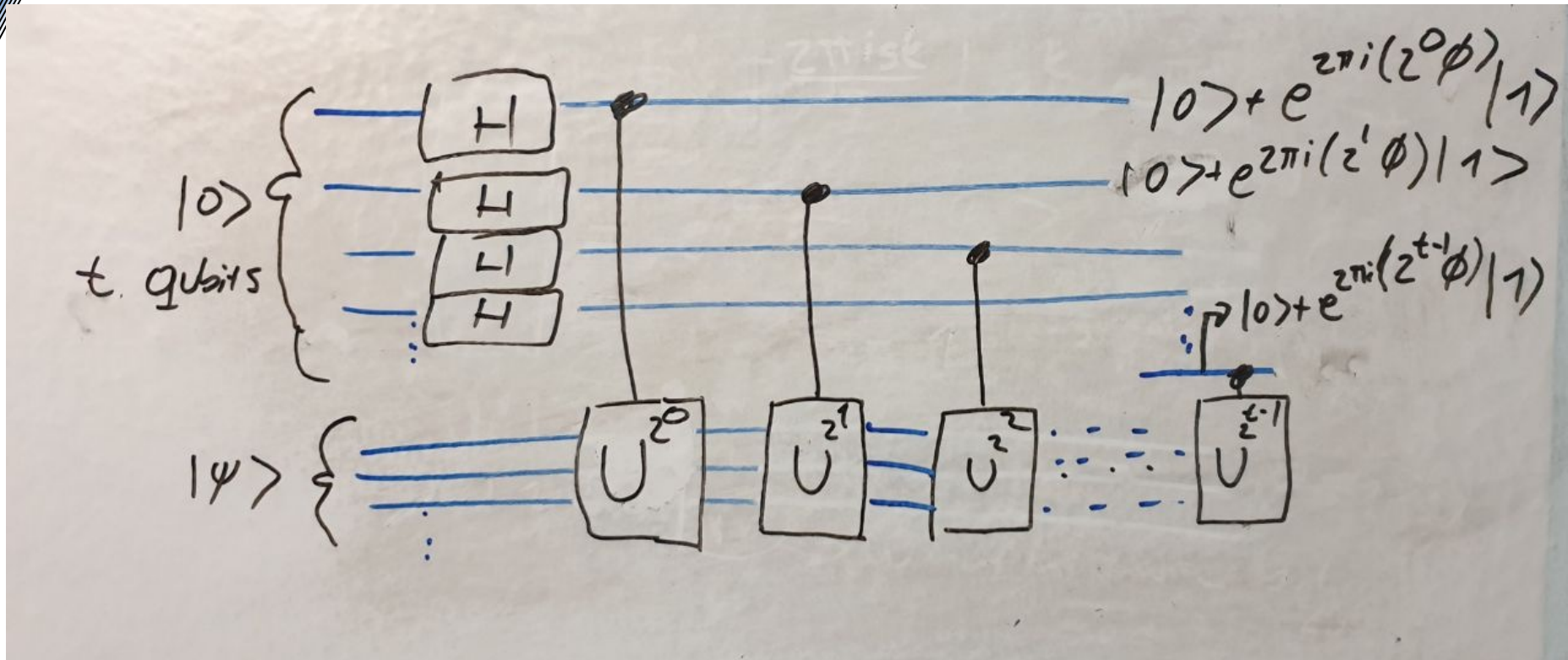
$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |\psi_s\rangle = 1$$

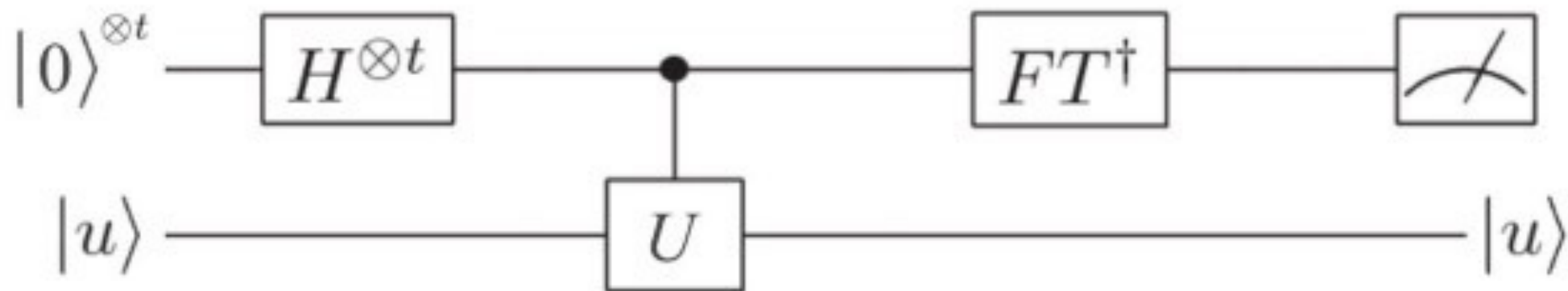
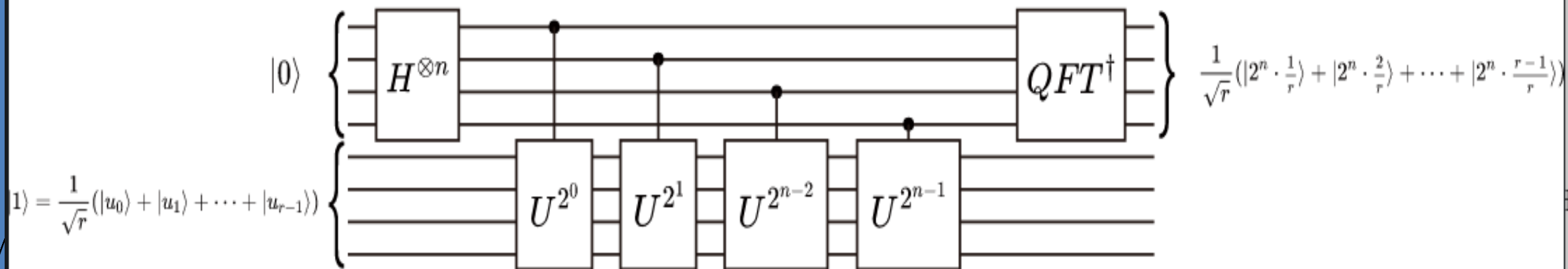
$$\Rightarrow |1\rangle = \frac{1}{\sqrt{r}} (|\psi_0\rangle + |\psi_1\rangle + \dots + |\psi_{r-1}\rangle)$$

what we are looking for.



# Designing Circuit





$$|\psi\rangle = \sum_{k=0}^{2^t-1} |k\rangle U^k |1\rangle = \sum_{k=0}^{2^t-1} |k\rangle x^{k \bmod N}.$$

$$\Downarrow \text{IQFT} \Rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} | \overset{s/r}{\phi} \rangle |v_s\rangle$$

# Fraction Algorithm

$$\frac{415}{93} = 4 + \frac{1}{2 + \frac{1}{6 + \frac{1}{7}}} \quad [4, 2, 6, 7]$$

- . d should be coprime with r
- . r should be even

$$\frac{c}{q} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\dots}}}, \quad d_0 = a_0, \quad d_1 = 1 + a_0 a_1, \quad d_n = a_n d_{n-1} + d_{n-2}$$

$$r_0 = 1, \quad r_1 = a_1, \quad r_n = a_n r_{n-1} + r_{n-2}$$

$$\frac{427}{512} = 0 + \frac{1}{1 + \frac{1}{5 + \frac{1}{42 + \frac{1}{2}}}}, \quad d_0 = 0, \quad d_1 = 1, \quad d_2 = \underline{5}, \quad d_3 = 427$$

$$r_0 = 1, \quad r_1 = 1, \quad r_2 = \underline{6}, \quad r_3 = 512$$

main.py



```
1 from math import gcd
2 from fractions import Fraction
3
4
5 N=119|
6 q=2048/(2**14)
7 print(q)
8 frac=Fraction(q).limit_denominator(N)
9 s, r = frac.numerator, frac.denominator
10 print(r)
11 a=8
12 guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
13 print(guesses)
```

Ln: 5, Col: 6



Run



Share

Command Line Arguments



0.125



8



[7, 17]

```
Result "0 00000000000000" happened 477 times out of 3
Result "0 01000000000000" happened 493 times out of 3
Result "0 01100000000000" happened 513 times out of 3
Result "1 10000000000000" happened 489 times out of 3
Result "1 10100000000000" happened 532 times out of 3
Result "1 11000000000000" happened 485 times out of 3
Result "1 11100000000000" happened 510 times out of 3
Result "0 00100000000000" happened 501 times out of 3
```



```
(base) C:\Users\Bilal Emmi\Desktop\qcomp\project>python tets.py
18 val of t, val of n bit length: 7
17 val of qubit numb
input number was: 119
```

Total number of qubits used: 17

Executing the circuit 3 times for N=119 and a=8

```
ibmqfactory.load_account:WARNING:2022-01-25 02:26:10,442:
Credentials are already in use. The existing account in the session will be
replaced.
Printing the various results followed by how many times they happened
(out of the 3 cases):
```

```
Result "0 000000000000000" happened 477 times out of 3
Result "0 010000000000000" happened 493 times out of 3
Result "0 011000000000000" happened 513 times out of 3
Result "1 100000000000000" happened 489 times out of 3
Result "1 101000000000000" happened 532 times out of 3
Result "1 110000000000000" happened 485 times out of 3
Result "1 111000000000000" happened 510 times out of 3
Result "0 001000000000000" happened 501 times out of 3
```

-----> Analysing result 001000000000000. This result happened in 16700.0000 % of all cases

In decimal, x\_final value for this result is: 2048

Running continued fractions for this case

Approximation number 1 of continued fractions:

Numerator:0            Denominator: 1

Odd denominator, will try next iteration of continued fractions

Approximation number 2 of continued fractions:

Numerator:1            Denominator: 8

The factors of 119 are 17 and 7

Found the desired factors!

Using a=8, found the factors of N=119 in 33700.0000 % of the cases

$$1 = 3*(1) + (-1)*(2)$$

$$1 = 5*(-1) + (2)*(3)$$

$$1 = 48*(2) + (-19)*(5)$$

$$1 = 5*(-19) + (2)*(48)$$

-19 48 einv, \_lambda

29 value d

public key(e,n): (53, 119)

public key(d,n): (29, 119)

[83, 33, 94, 75, 76, 2, 51, 88, 33, 94]

[104, 101, 110, 108, 111, 32, 102, 114, 101, 110]

String: henlo fren

Your decrypted message is: henlo fren

# Qubit Count

```
n=math.ceil(math.log2(val))
```

```
bit_n=2*n+3
```

Is it possible to  
use smaller qubits?

(Shor's algorithm with fewer  
(pure) qubits, Christof Zalka, 2008)

$$r \cdot a + k \cdot N = r'$$

Apply for mod  $N$

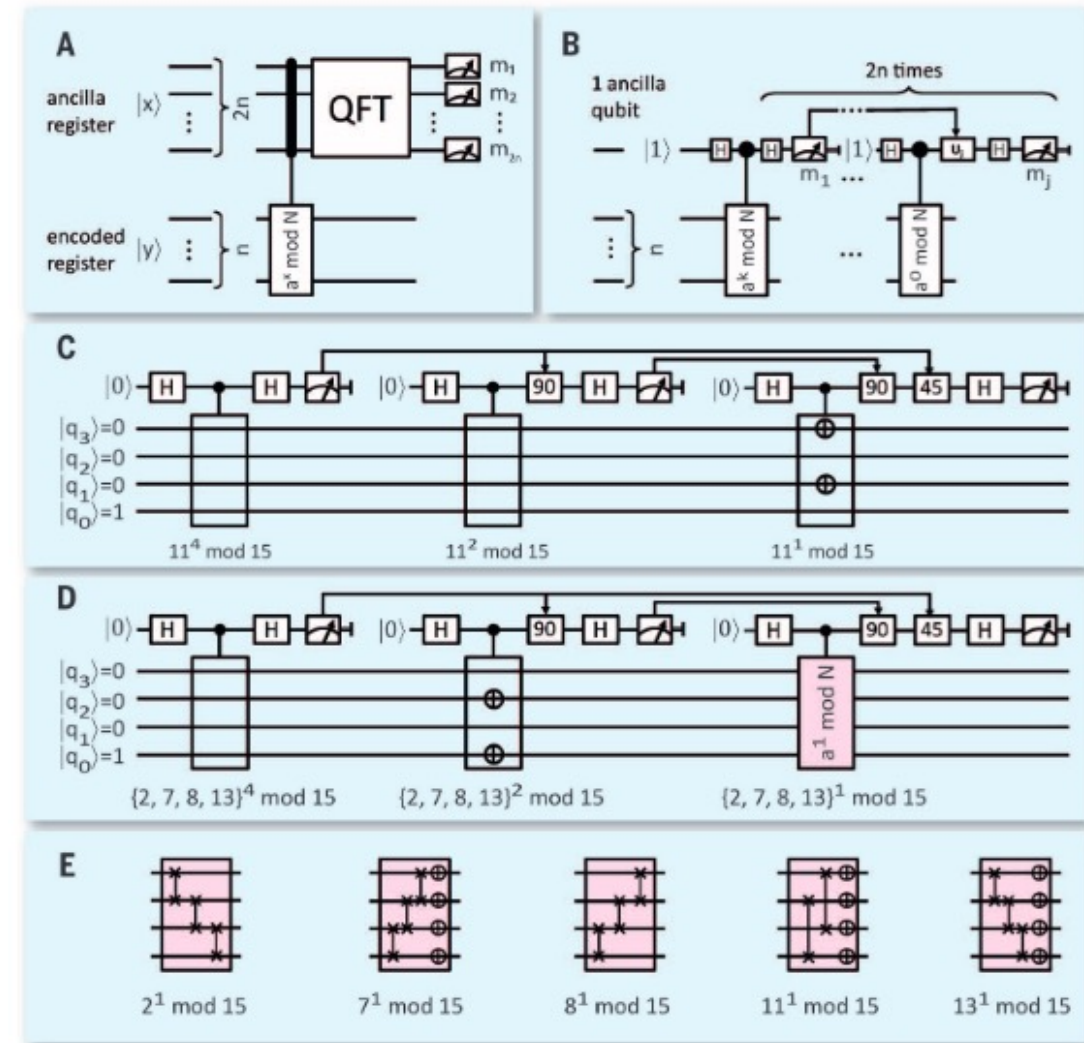
$$r \cdot a \equiv r' \pmod{N}$$
$$a = r' / r \pmod{N}$$

→ This can be found by using Extended Euclid's Algorithm

Realization of a scalable  
Shor algorithm, Monz et al., 2016

Certain algorithms for quantum computers are able to outperform their classical counterparts. In 1994, Peter Shor came up with a quantum algorithm that calculates the prime factors of a large number vastly more efficiently than a classical computer. For general scalability of such algorithms, hardware, quantum error correction, and the algorithmic realization itself need to be extensible. Here we present the realization of a scalable Shor algorithm, as proposed by Kitaev. We factor the number 15 by effectively employing and controlling seven qubits and four “cache qubits” and by implementing generalized arithmetic operations, known as modular multipliers. This algorithm has been realized scalably within an ion-trap quantum computer and returns the correct factors with a confidence level exceeding 99%

<https://sci-hub-se/10.1126/science.aad9480>





**Fig. 1. Quantum circuits.** Diagrams of Shor's algorithm for factoring  $N = 15$ , using a generic textbook approach (A) compared with Kitaev's approach (B) for a generic base  $a$ . (C) The actual implementation for factoring 15 to base 11, optimized for the corresponding single-input state. Here  $q_i$  corresponds to the respective qubit in the computational register. (D) Kitaev's approach to Shor's algorithm for the bases  $\{2, 7, 8, 13\}$ . Here, the optimized map of the first multiplier is identical in all four cases, and the last multiplier is implemented with full modular multipliers, as depicted in (E). In all cases, the single QFT qubit is used three times, which, together with the four qubits in the computation register, totals seven effective qubits. (E) Circuit diagrams of the modular multipliers of the form  $a \bmod N$  for bases  $a = \{2, 7, 8, 11, 13\}$ .



[doi:10.1103/PhysRevLett.127.140503](https://doi.org/10.1103/PhysRevLett.127.140503)

## Factoring 2048-bit RSA Integers in 177 Days with 13 436 Qubits and a Multimode Memory

Élie Gouzien \* and Nicolas Sangouard †

*Université Paris-Saclay, CEA, CNRS, Institut de Physique Théorique, 91191 Gif-sur-Yvette, France*

(Dated: September 29, 2021)

We analyze the performance of a quantum computer architecture combining a small processor and a storage unit. By focusing on integer factorization, we show a reduction by several orders of magnitude of the number of processing qubits compared with a standard architecture using a planar grid of qubits with nearest-neighbor connectivity. This is achieved by taking advantage of a temporally and spatially multiplexed memory to store the qubit states between processing steps. Concretely, for a characteristic physical gate error rate of  $10^{-3}$ , a processor cycle time of 1 microsecond, factoring a 2048-bit RSA integer is shown to be possible in 177 days with 3D gauge color codes assuming a threshold of 0.75 % with a processor made with 13 436 physical qubits and a memory that can store 28 million spatial modes and 45 temporal modes with 2 hours' storage time. By inserting additional error-correction steps, storage times of 1 second are shown to be sufficient at the cost of increasing the run-time by about 23 %. Shorter run-times (and storage times) are achievable by increasing the number of qubits in the processing unit. We suggest realizing such an architecture using a microwave interface between a processor made with superconducting qubits and a multiplexed memory using the principle of photon echo in solids doped with rare-earth ions.

# References

<https://hal.archives-ouvertes.fr/hal-03358148/document>

<https://www.science.org/doi/10.1126/science.aad9480>

<https://arxiv.org/pdf/quant-ph/0601097.pdf>

[https://qiskit.org/documentation/stable/0.26/\\_modules/qiskit/aqua/algorithms/factorizers/shor.html](https://qiskit.org/documentation/stable/0.26/_modules/qiskit/aqua/algorithms/factorizers/shor.html)

<https://github.com/ttlion/ShorAlgQiskit>

<https://qiskit.org/textbook/ch-algorithms/shor.html>

<https://young.physics.ucsc.edu/150/shor.pdf>

<https://jonathan-hui.medium.com/qc-period-finding-in-shors-algorithm-7eb0c22e8202>

<https://www.codespeedy.com/rsa-algorithm-an-asymmetric-key-encryption-in-python/>

Kriptografiye Giris, Erkan Afacan ,epos 2016



# My Github Repository

- For further questions you can look up the codes that I made for this project and you can apply it on your own.
- <https://github.com/Tihulu/quantumcomputing>