THE CANADIAN SCIENCE FAIR JOURNAL

ARTICLE

The Final Countdown: How Much Longer Until Quantum Computers Become the Next Cybersecurity Threat

Ria Patel

16 | Nepean, ON

Awards Bronze Medal at Canada-Wide Science Fair 2021 | Best in Age category (Intermediate) at Ottawa Regional Science Fair 2021 | Divisional award: Information Challenge award (Intermediate) at Ottawa Regional Science Fair 2021 | Special award: Honeywell Aerospace at Ottawa Regional Science Fair 2021 | Interdisciplinary award: First Place (Intermediate) at Ottawa Regional Science Fair 2021

Upon learning about the exponential power of quantum computers, I was concerned about the threat that they pose to encryption algorithms which are currently believed to be unbreakable. I wanted to answer the question: when will quantum computers be able to hack even the most secure encryption? I used Python programming to model how a 64-qubit quantum computer could crack AES-128 bit, RSA-1024 bit, and SHA-256 hashing algorithm-based encryption. Since programming a real quantum computer was not feasible, I used a regular computer to simulate the iterations it would take a simplified quantum computer to crack the encryption. After testing all three simulations, it can be concluded that a 64-bit quantum computer would still take an astronomical number of iterations to crack modern encryption. However, as this technology grows and quantum computers with the same number of qubits as our computers have bits (about 4×10^{12}) are built, quantum hacking could become a real threat.

INTRODUCTION

Everyday, billions of people around the world send trillions of messages, memes and tweets online. This information is kept secure by encryption: the process of encoding information so that only the intended recipient can decode it. Upon learning about the exponential power of quantum computers, I was concerned about the threat that they pose to encryption algorithms which are currently believed to be unbreakable. Hundreds of millions of people around the world use the internet for confidential banking transactions and this number is increasing thanks to the rapid adoption of cryptocurrencies. Cryptocurrencies depend on blockchain: a system that ensures the legitimacy of digital transactions while maintaining complete anonymity between parties. If quantum computers could break blockchain encryption, it would be an international catastrophe.

I wanted to answer the question: when will quantum computers be able to hack even the most secure encryption? I used Python programming to model how a 64-qubit quantum computer could crack AES-128 bit, RSA-1024 bit, and SHA-256 hashing algorithm-based encryption. Since programming a real quantum computer was not feasible, I used a regular computer to simulate the iterations it would take a simplified quantum computer to crack the encryption. Additionally, I could not reasonably compare the efficiency of a quantum computer to a classical computer, because a classical computer would take 1 billion billion years to hack AES-128 using the brute force of my simulated quantum computer (Arora, 2021).

BACKGROUND

Classical computers store information at the most basic level in binary bits, meaning a 1 or a 0. Information is securely communicated by encrypting the bits that represent the message. Texts and emails are encrypted using AES-128 bit symmetric keys. These determine how every 128 bit block of the message is encrypted before the message is sent. The receiver must have the same key to decrypt the message, hence the term "symmetric" (Franklin, 2020). To crack AES-128 without the symmetric key, a computer would need to brute force 2^{128} computational possibilities, far beyond the reach of classical computing (Wood, 2011).

Another kind of encryption is RSA-1024. Unlike AES-128, this is asymmetrical because it uses two different keys. The first key is composed of two large prime numbers, and the second key is their semiprime product. The semiprime product is the public key, meaning it is available to the public and can be used to encrypt any message. It is 1024 decimal digits long, hence the name RSA-1024. The two prime numbers make up the private key and are needed to decrypt any message. Although the semiprime product of two very large prime numbers can be computed very easily, going the other way around to factor a semiprime product into

two very large prime numbers, to decrypt a message without the private key, is very difficult. (Krupansky, 2018). However, unlike AES-128, RSA encryption is vulnerable to attack from classical computers. Larger and larger RSA keys have been broken over the past few decades as classical supercomputers become stronger and factoring algorithms become more efficient. The most recent breakthrough was when Zimmermann et al. (2020) hacked the RSA-250.

Finally, the SHA-256 hashing algorithm is commonly used for online security protocols. It has recently been used in blockchain technology, which will be integral in our future cryptocurrency economy. SHA-256 is not actually an encryption algorithm like AES-128 and RSA-1024. It is a "hash" function that reduces the entire message being sent to a unique set of 256 bits. The SHA-256 hash is not meant to be decrypted, because it is simply a signature to verify the authenticity of the message, like in banking transactions (Stevens, 2020). In fact, there is no way to go from the SHA-256 hash back to the original message, because the original message could be of any length. Even though a quantum computer could test possibilities much more quickly than a classical computer, there is no limit to the number of possibilities that created a 256-bit hash. However, this does not mean SHA-256 is completely secure. Multiple messages could generate the same SHA-256 hash, so one could try and trick the system into verifying a false message as if it was the original text. This is known as a "collision" and it allows the hacker to send false information to the system, like improper cryptocurrency transactions. Although a collision for SHA-160 has been found (Stevens 2017), SHA-256 still remains secure against classical computers. However, quantum computers may be able to find collisions much more easily.

Quantum computing operates on the quantum physics principles of entanglement and superposition (Quantumly, 2017). Entanglement occurs when the states of two particles are linked such that measuring the state of one allows you to definitively know the state of the other particle without measuring it. Superposition is the collection of all the states that a quantum particle experiences simultaneously, and it also describes the probability of each state occurring when the particle is measured. While traditional computing uses bits that can take on a value of 0 or 1, quantum computers use qubits that exist as a superposition of 1 and 0 until measured (Aaronson, 2007. p. 68). In terms of computing power, since qubits are a superposition of multiple states, and those states can be entangled, quantum computers can test multiple possibilities at the same time and then extract a solution using a Quantum Fourier Transform (QFT) (MinutePhysics, 2019).

The QFT is the quantum implementation of the Discrete Fourier Transform (DFT). The DFT computes the frequency domain representation of data provided in the time domain, revealing detailed information about the hidden frequencies that may not be obvious in the time domain. The QFT performs a similar operation to the superposition of quantum particles (QuTech Academy, 2021), and is ultimately used in Shor's algorithm to factor large semiprime numbers.

HYPOTHESIS

I hypothesized that it would take a quantum computer of a controlled size (64 qubits for AES-128 and RSA-1024, 10^{50} for SHA-256) the highest number of iterations of my program to crack the SHA-256 hashing algorithm, the second highest number of iterations to crack AES-128, and the least number of to crack RSA-1024.

The SHA-256 hashing algorithm generates a 256-bit key, so there are 2^{256} possible hashes to test. This is the most of any of the encryption algorithms, so I predicted it would take the most time to crack. The AES-128 encryption uses a 128-bit string, so there are 2^{128} possible strings to test. The RSA-1024 encryption can be broken theoretically using Shor's algorithm that efficiently factors large primes using quantum computers, so I predicted that it would take the least number of iterations.

MATERIALS

The materials used for this project are a Wing Personal IDE for Python 3.3 and Python libraries 'random', 'time', 'numpy', and 'statistics'. From the 'statistics' library, I used the function 'mode'.

VARIABLES

The control variables in this experiment are the program I wrote for each type of encryption, the size of the quantum computer, the fixed operating speed of quantum computers, and the fixed operation speed of my own computer. If I made any error in any of these parts, it would be equally reflected across all my results. The independent variables were the type and size of encryption I tried to crack. The dependent variable was the number of iterations it would theoretically take a quantum computer to crack the encryption based on my simulation.

METHODS

Since I didn't have access to any quantum computers, I used the Python programming language to model them. I designed a program that follows the steps a real quantum computer would take to crack the different encryption algorithms. I designed three separate programs to calculate the speed a 64-qubit quantum computer would take to crack AES-128 bit, RSA-1024 bit, and SHA-256 hashing algorithm based encryption.

AES-128: A quantum computer can test 64 qubits worth of possibilities at once then use the QFT to pick out the right one. A brute force attack against AES-128 can be implemented by calculating the number of iterations it would take 64 qubits to produce keys of 128-bit length until a match is found to the symmetric key. See Appendix A for the full code.

RSA-1024: The following procedure was based on MinutePhysics, 2019 and the Emerging Technology from the arXiv archive page, 2019. Shor's algorithm can be used to factor a large

semiprime N. The algorithm starts by guessing a prime factor of N, represented as g. If g is not a factor of N, then $g^{(p/2)}\pm1$, where p is a positive integer less than N, are two numbers very likely to share factors with N (though they themselves may not be factors of N). A quantum computer can find p by making and manipulating a superposition of all pairs of p and their respective remainders, r, when N is divided by g^p. The goal is to find a value x in the set of all p so that $g^x=mN+1$, where m is some positive integer. This is equivalent to saying that $g^{(x/2)}\pm1$ share factors with N.

To find x, the r component of the (p,r) superposition is measured, leaving the quantum computer with all pairs (p,r) with the same r value. This occurs because measuring the superposition collapses it into one value. The trick is to notice that if $g^x=mN+1$ and $g^p=mN+r$, then $g^{(p+qx)}$ also equals $mN+r$, where q is any positive integer. Therefore, all p in this superposition are sequentially x apart from each other. A QFT can extract x from superposition, and thus the quantum computer would be able to find two numbers $g^{(x/2)}\pm1$ which are very likely to share a factor with N. After 10 guesses, there is a 99% chance that those numbers share a factor with N. Once the numbers are found, the Euclidean algorithm can be used to break up $g^{(x/2)}\pm1$ and N into their factors and find two prime factors of N, which are the private keys. See Appendix B for the full code.

SHA-256: A brute-force collision was simulated as follows. Instead of creating random messages and hashing them into keys, a random 256-bit hash was chosen to be the starting key. Then, random strings of 256 bits were generated until a string matched with the key. The assumption is that the random hashes can all be created from unique messages. This was implemented by calculating the number of iterations it would take 64 qubits to test random hashes of 256-bits until a match with the key was found. A quantum computer can test many possibilities at once then use the QFT to pick out the right one (Sahu, 2021). See Appendix C for the full code.

For every encryption algorithm, 10-20 random keys were tested. Instead of using the full-length keys (e.g. 128 bits), they were scaled down empirically to have a reasonable run time. For example, in the AES-128-bit encryption, a 32-bit key was tested with a 16-qubit computer and then the resulted were scaled up by four to get to 128 bits with a 64-qubit computer. The results would scale linearly because the exponential increase in key possibilities is balanced by the exponential increase in computing. In the results, this is documented as "average simulated iterations," which are the values that were used in the actual program and "average calculated iterations," which are the values that are scaled back up to the full key lengths and full number of qubits. See Table 1 for the simulated key sizes and the calculated theoretical key sizes and see Table 2 for the simulated number of qubits and theoretical number of qubits.

RESULTS

Table 1. Length of keys/hashes in each simulation. The "Length of Partial Key" column contains the length of each key tested in the program. The "Theoretical Length of Full Key" column contains the scaled length of each key.

Encryption Method	Length of Partial Key:	Theoretical Length of Full Key:
AES-128	32 bits	128 bits
RSA-1024	10-13 bits	1024 bits
SHA-256	8 bits	256 bits

Table 2. Qubits used in each simulation. The "Number of Simulated Qubits for Partial Key" column contains the number of qubits inputted in the program. The "Theoretical Number of Qubits for Full Key" column contains the scaled number of qubits for a full key length.

Encryption Method	Number of Simulated Qubits for Partial Key:	Theoretical Number of Qubits for Full Key:
AES-128	16	64
RSA-1024	4	64
SHA-256	4	1.2061×10^{50}

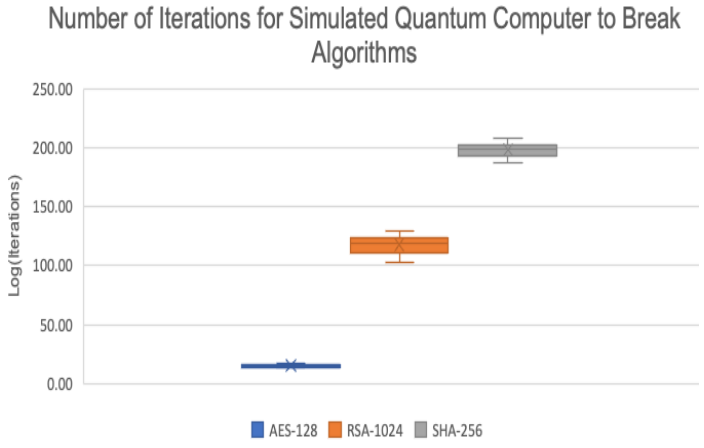

Figure 1. Number of iterations the simulated quantum computer took to break the AES-128, RSA-1024, and SHA-256 algorithms.

Table 3. Iterations of the program needed to hack each simulation. The "average simulated iterations" column contains the exact numbers that came out of the program. The "average calculated iterations" column contains the scaled version for a full key length.

Encryption Method	Mean Simulated Iterations for a Partial Key:	Mean Extrapolated Iterations for a Full Key:
AES-128	44751.6	2.87×10^{19}
RSA-1024	27.35	1.126×10^{164}
SHA-256	1744776.6	3.74051×10^{206}

DISCUSSION

The mean extrapolated iterations to hack the encryptions were 3.74051×10^{206} for the SHA-256 hashing algorithm, 1.126×10^{164} for the RSA-1024 encryption, and 2.87×10^{19} for the AES-128 encryption. However, the mean simulated iterations for a partial key were 1744776.6 for SHA-256, 44751.6 for AES-128, and 27.35 for RSA-1024. SHA-256 consistently took the longest, whereas RSA-1024 and AES-128 flip flipped due to the change in key length. The extrapolated results are interesting as the RSA-1024 is the longest and most memory intensive key, four times that of the SHA-256.

CONCLUSION

After testing all three simulations, it can be concluded that a 64-bit quantum computer would still take an astronomical number of iterations to crack modern encryption. The current largest quantum computer by IBM boasts 65 qubits so theoretically they could crack AES-128 if they tried for a very long time. However, as this technology grows and quantum computers with the same number of qubits as our computers have bits (about 4×10^{12}) are built, quantum hacking could become a real threat.

FUTURE STEPS

If I were to redo this project, I would consider longer testing times since my results were based on testing times of less than a minute. I would also like to physically verify my simulated programs, so they are more accurate and reflective of actual quantum computers and cryptography.

ACKNOWLEDGEMENTS

I would like to recognize everyone who has helped me with this project. My family, who encouraged me to solve hard problems; and my teachers, who taught me how to find solutions. I attended the 2020-2021 Python in Motion math enrichment class taught by Professor Alexey Godin from Carleton University. The insight from that course helped me develop my own programs.

WORKS CITED

Aaronson, S. (2008). The Limits of Quantum. Retrieved from http://www.cs.virginia.edu/~robins/The_Limits_of_Quantum_Computers.pdf

Aaronson, S. (2007). Shor, I'll do it. Retrieved from https://www.scottaaronson.com/blog/?p=208

Arora, M. (2012, May 7). How Secure is AES against brute force attacks? Retrieved from https://www.eetimes.com/how-secure-is-aes-against-brute-force-attacks/

Emerging Technology from the arXiv archive page. (2019, May 30). How a quantum computer could break 2048-bit RSA encryption in 8 hours. Retrevied from https://www.technologyreview.com/2019/05/30/65724/how-a-quantum-computer-could-b reak-2048-bit-rsa-encryption-in-8-hours/

Franklin, R. (2020, March 13). AES vs. RSA Encryptions; What Are the Differences? Retrieved from https://www.precisely.com/blog/data-security/aes-vs-rsa-encryption-differences

Krupansky, J. (2018, September 30). Some Preliminary Questions About Shor's Algorithm for Cracking Strong Encryption Using a Quantum Computer. Retrieved from https://jackkrupansky.medium.com/some-preliminary-questions-about-shors-algorithm-fo r-cracking-strong-encryption-using-a-quantum-b3470546249c

Minutephysics. (2019, May 22). How Shor's algorithm factors 314191. Retrieved from https://www.youtube.com/watch?v=FRZQ-efABeQ

Quantumly. (2017). Where do quantum computers get their speed. Retrieved from http://quantumly.com/quantum-computer-speed.html

QuTech Academy. (2021, January 26). Quantum Fourier Transform by MSc students Elsie Loukiantchenko & Maria Flors Mor Ruiz. Retrieved from https://www.youtube.com/watch?v=MBfyQ5wqmvw

Sahu, M. (2021, January 4). Cryptography in Blockchain: Types & Applications [2021]. Retreived from https://www.upgrad.com/blog/cryptography-in-blockchain/

Schadeck, W. X. (2017, May 27). What is blockchain, really? (An intro for regular people). Retrieved from https://medium.com/@wen_xs/what-is-blockchain-really-an-intro-for-regular-people-e51578d98a96

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y. The first collision for full SHA-1. Retrieved from https://shattered.io/static/shattered.pdf

Stevens, R. (2020, May 12). Quantum computers could crack Bitcoin by 2022. Retrieved from https://decrypt.co/28560/quantum-computers-could-crack-bitcoins-encryption-by-2022

Wood, L. (2011, March 21). The Clock is Ticking for Encryption. Retrieved from https://www.computerworld.com/article/2550008/the-clock-is-ticking-for-encryption.htm l#:~:text=But%20using%20quantum%20technology%20with,crack%20a%20128%2Dbit%20key.

Zimmerman, Paul. (2020, February 28). Factorization of RSA-250. Retrieved from https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html

RIA PATEL

Ria Patel is a high school student in Ottawa, passionate about STEM. She enjoys learning about developments in new technologies like quantum computing. Outside of academics, she can be found running, playing the piano, listening to music, or enjoying a good book.

APPENDICES

```python
#AES-128
import random

def Iterations(key_lenght,qubits):
    qubit_capabilties = 2**qubits
    iterations_needed = key_lenght/qubits
    return qubit_capabilties*iterations_needed

def MatchKey(iterations,key,qubit):
    total_computations = 0
    for i in range(int(iterations)):
        key_try = (random.randint(1,iterations))/iterations
        total_computations += 1
        if key_try == key:
            return total_computations

if __name__ == '__main__' :
    key_lenght = 32
    qubits = 16

    iterations = Iterations(key_lenght,qubits)

    Key = (random.randint(1,iterations))/iterations

    computations = MatchKey(iterations,Key,qubits)
    print("number of iterations to find all possiblities of key lenght", key_lenght, "with", qubits, "qubits :", computations)
    print("but since we are testing AES-128 not AES-32, we can take our answer to the power of 4:", computations**4)
```

Appendix A. AES-128 simulation code.

```python
import random

def Iterations(key_lenght,qubits):
    qubit_capabilties = 36**qubits
    iterations_needed = key_lenght/qubits
    return qubit_capabilties*iterations_needed

def MatchKey(iterations,key,qubit):
    total_computations = 0
    for i in range(int(iterations)):
        key_try = (random.randint(1,iterations))/iterations
        total_computations += 1
        if key_try == key:
            return total_computations

if __name__ == '__main__' :
    key_lenght = 8
    qubits = 4

    iterations = Iterations(key_lenght,qubits)

    Key = (random.randint(1,iterations))/iterations

    computations = MatchKey(iterations,Key,qubits)
    print("number of iterations to find all possiblities of key lenght", key_lenght, "with", qubits, "qubits :", computations)
    print("but since we are testing SHA-256 not SHA-8, we can take our answer to the power of 32:", computations**32)
```

Appendix B. RSA-256 simulation code.

```python
###RSA-1024: Shor's algorithm

import numpy as np
import random
import time
import statistics
from statistics import mode

###Guessing G:
def GuessG(N):
    possible_g = []
    for g in range(2, (N//2)+1):
        if g > 1:
            for i in range(2, g):
                if (g % i) == 0:
                    break
                else:
                    possible_g.append(g)
    pos_g = np.random.choice(possible_g)
    if N%pos_g != 0:
        return pos_g
    else:
        return np.random.choice(possible_g)

###Finding values for R:
def FindR(g,N):
    superposition = []
    for i in range(N):
        g1 = g**(random.randint(1,N))
        g2 = g1%N
        superposition.append(g2)
        if len(superposition) == 4:
            r = superposition[random.randint(0,3)]
            return r

###Euclidean algorithm:
def Euclid(a,b):
    if a == b:
        return a
    elif a == 0 or b == 0:
        return False
    elif a < b:
        q = b
        r = a
    else:
        q = a
        r = b
```

Appendix C. RSA-1024 simulation code.

```
    for i in range(q):
        rem = q%r
        if rem == 0:
            return r
        else:
            q = r
            r = rem

def Test(counter):
    counter+=1
    N = 5891 #public key
    for i in range(20):
        g = GuessG(N)

        for i in range(5):
            r1 = FindR(g,N)
            r2 = FindR(g,N)
            r3 = FindR(g,N)
            r4 = FindR(g,N)

        e1 = Euclid(r1,r2)
        e2 = Euclid(r1,r3)
        e3 = Euclid(r1,r4)
        e4 = Euclid(r2,r3)
        e5 = Euclid(r2,r4)
        e6 = Euclid(r3,r4)

        p_options = [e1,e2,e3,e4,e5,e6]
        one = 1
        p_options = [i for i in p_options if i != one]
        if p_options == []:
            Test(counter)
        else:
            p = statistics.mode(p_options)

            if p%2 != 0: #if the p for p/2 is odd
                Test(counter)
            else:
                p1 = g**(p//2) + 1
                p2 = g**(p//2) - 1

                factor1 = Euclid(p1, N)
                factor2 = Euclid(p2, N)
                if factor1 != 1 or factor2 != 1:
                    if N%factor1 != 0:
                        Test(counter)
                    if N%factor2 != 0:
                        Test(counter)
                    if factor1 == 1:
                        factor1 = N/factor2
                    elif factor2 == 1:
                        factor2 = N/factor1
                    else :
                        Test(counter)
                    print("Prime factors of", N, "are", factor1, "and", factor2)
                    print("Iterations:", counter)
                    exit()
                else:
                    Test(counter)

if __name__ == '__main__' :
    counter = 0
    Test(counter)
```

Appendix C continued. RSA-1024 simulation code.