# How would quantum computing impact the security of Bitcoin by enhancing our ability to solve the elliptic curve discrete logarithm problem?

Neeraj J. Samtani

**Abstract:** Bitcoin, being the most widely used cryptocurrency, should have no security vulnerabilities. When users transfer Bitcoin, they must 'sign off' on the transaction using a private key generated by the elliptic curve digital signature algorithm (ECDSA). Calculating a user's private key from their public key is known as the elliptic curve discrete logarithm problem (ECDLP). The only known method to solve this problem on classical computers is through brute-force, which takes exponential time. However, quantum computers can run a modified version of Shor's algorithm to solve the ECDLP in polynomial time, thus posing a threat to the security of ECDSA. In this paper I explain what makes the ECDLP intractable and run an experiment to estimate the time taken to solve the ECDLP on a classical computer. I then describe the modified version of Shor's algorithm which can solve the ECDLP and compare it to brute forcing a solution on a classical computer. My research has shown that in the advent of quantum computers with sufficient qubits, the signature algorithm used in Bitcoin needs an update. Finally, I suggest a quantum-resistant alternative to ECDSA – Lamport Signatures.

**Key Words:** Bitcoin, quantum computing, elliptic curve discrete logarithm problem, Safety/security in digital systems

## 1. Introduction

### 1.1 What is Bitcoin?

With a rise in online trading and mobile payments, digital currencies are becoming more relevant than ever before. The most used is Bitcoin, a decentralized, distributed, peer-to-peer cryptocurrency created in 2009 by an unknown programmer. They aren't issued by any government nor managed by any bank. Many companies such as Dell and Reddit now accept Bitcoin as a form of payment for their goods and services [1]. There are several advantages to using Bitcoin, such as low transaction fees, the ability to be used in any country, and the lack of capital control.

### 1.2 Explanation of the Question

The code that Bitcoin runs on uses several algorithms and if just one of them has a security flaw, the whole system could be compromised. Bitcoin uses the Elliptic Curve Digital Signature Algorithm (ECDSA) to generate public keys for their users from a randomly selected private key. Proposed by Scott Vanstone, ECDSA is accepted by the International Standards Organization and is used by many programmers for digital key generation and verification. "The mathematical basis for the security of elliptic curve cryptosystems is the computational intractability of the elliptic curve discrete logarithm problem (ECDLP)" [2]. This essay examines the security of the ECDSA by exploring the possibility of solving the ECDLP with modern technology, and, due to the possible advent of quantum computing, aims to theoretically answer the more important question **"How would quantum computing impact the security of Bitcoin by enhancing our ability to solve the elliptic curve discrete logarithm problem?"**

## 2. Working of Bitcoin

Bitcoin uses several common techniques to keep their users' data secure and to enable transactions. Three of these techniques are hash functions, digital signatures, and the blockchain.

### 2.1 Hash Functions

Hash functions are mathematical functions that accept an input string of any size and produce a fixed-size output comprising seemingly random characters [3]. These functions are efficiently computable. If one character is changed in the input string, the output is completely changed. However, if a hash function is run multiple times on the same input, it will always produce the same output. For SHA-256, the output string is always 256 bits long. These two properties allow hash functions to serve as fixed-length summaries for a given input. Hash functions are quantum resistant [4].

### 2.2 Digital Signatures

A digital signature is comparable to a handwritten signature on a piece of paper – it ensures the validity of a document. In Bitcoin, the digital signature changes depending on which document is being signed. This prevents the signature from being copied and pasted onto other documents.

Every node on the Bitcoin network generates its own public and private key. The private key is kept secret and the public key is known by all nodes. The public key acts as an identity for a node and nodes can "speak" for the identity using the private key. Both the message on a document and the private key are used to create the signature, thus making it unique for each document. This prevents the signature from being forged onto other documents. To verify that the signature is valid, other nodes use the message, the public key, and the signature left by the node.

The algorithm used to generate public keys from private keys in Bitcoin is the Elliptic Curve Digital Signature Algorithm (ECDSA). The private key is any random number that can be generated by an algorithm or chosen by the user. In Bitcoin, "each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership" [5]. This is done with the help of the blockchain.

### 2.3 Blockchain

"The block chain is a shared public ledger on which the entire Bitcoin network relies. All confirmed transactions are included in the block chain. This way, Bitcoin wallets can calculate their spendable balance and new transactions can be verified to be spending bitcoins that are actually owned by the spender" [6].

Every Bitcoin transaction is recorded in the blockchain, a copy of which is stored on millions of computers around the world [7]. The blockchain can be viewed by anyone on the network, thus allowing anyone to check the validity of transactions. This makes Bitcoin very secure because any tampering by a malicious user to the blockchain can be detected by other computers and the actions of this malicious user will be disregarded in other copies of the blockchain.

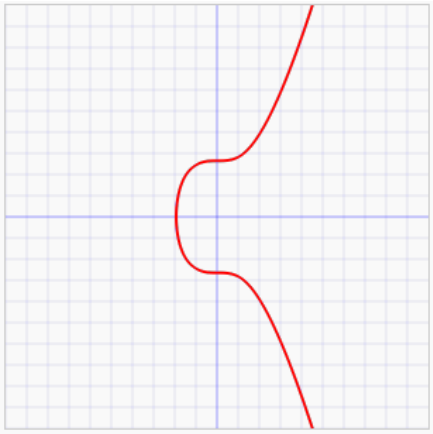
## 3. Elliptic Curve Digital Signature Algorithm (ECDSA)

Since the blockchain and hash functions aren't vulnerable to quantum computing, the only possible weakness in Bitcoin is its digital signature algorithm.

### 3.1 Introduction to ECDSA

The Elliptic Curve Digital Signature Algorithm is used to generate pairs of private and public keys and is based on a polynomial equation which can be plotted on a cartesian plane. The specific equation used by Bitcoin is named Secp256k1 and has the following equation as defined by Federal Information Processing Standards.

$$y^2 = x^3 + 7$$



The Secp256k1 curve plotted on a cartesian plane

However, the actual implementation of ECDSA in Bitcoin doesn't resemble the graph above because it is defined over another field called $Z_p$ (defined for only prime numbers). Plotting that graph would result in seemingly scattered points.

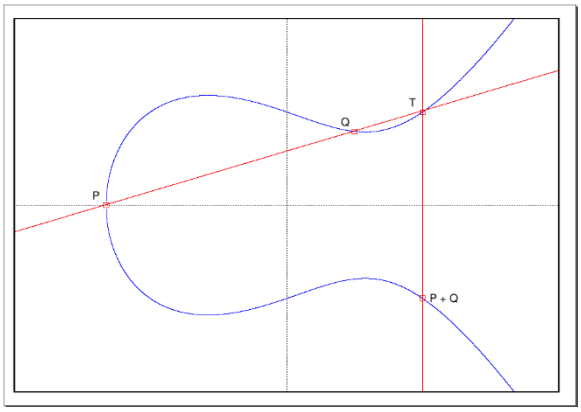
The ECDSA generates and verifies keys according to certain parameters. These parameters (outlined in the appendix) are described for each curve in a document titled "Recommended Elliptic Curve Domain Parameters" written by Certicom Research. ECDSA randomly generates an integer private key from a selection with high entropy or allows a user to choose their own value. It then calculates a public key by multiplying the private key and the generation point – a constant value for that curve. Instead of normal multiplication, the algorithm employs invented elliptic curve mathematics to multiply the two values and calculate the public key.

### 3.2 Invented Mathematical Operations for Elliptic Curves

Elliptic curve arithmetic is complicated. However, it has an intuitively understandable geometric interpretation which is expressed below.

#### 3.2.1 Addition

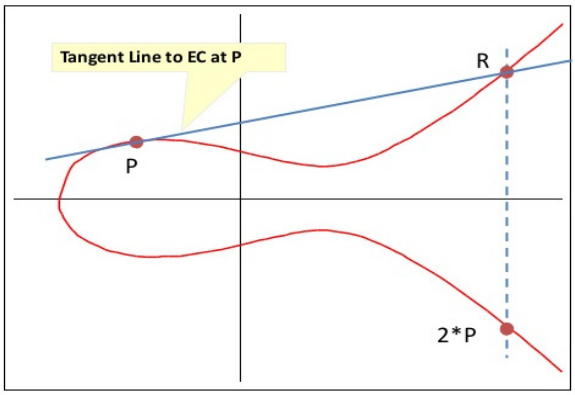
The slope of any two points on an elliptic curve intersect at a third point as well. Adding two points on an elliptic curve involves finding this third point and reflecting it in the x-axis. Elliptic Curve addition employs modular arithmetic, meaning that if a calculated value crosses a set maximum value (officially called Pcurve in the Bitcoin code, outlined in the appendix), the value wraps around and starts from 0 again. This value is the largest possible value that a private key can take.




Elliptic Curve Addition

### 3.2.2 Point Doubling

When trying to add a point to itself, there is no slope that can be found. Hence, the tangent of the curve at that point is taken and the second point on the graph that the tangent intersects is reflected in the x-axis to find the sum. This is known as point doubling and makes use of modular arithmetic as well.




Elliptic Curve Point Doubling

### 3.2.3 Multiplication

When computing with elliptic curves, computers employ the 'double and add' algorithm to increase the efficiency of Elliptic Curve Point Multiplication. Point Multiplication involves repeatedly adding a point to itself using a combination of elliptic point addition and elliptic point doubling. For example, a point A would first be doubled to find 2A, and then 2A would be added to A to find 3A.

## 3.3 Generating Public Keys

The following piece of code written by the Congressional Research Institute represents a Python implementation of ECDSA Multiplication. It is used to generate a public key from a user's private key. The original Bitcoin version is written in C and has many dependencies; hence it is easier to refer to this code. Multiplication is repeatedly adding a number to itself, so that is exactly what the algorithm is doing.

In the code, the function EccMultiply accepts the generation point (GenPoint) and the users private key (ScalarHex). Before generating the public key, the function checks whether the Private Key is valid, then moves on the convert it into binary form. A loop is initiated to traverse each bit of the Private key.

For each bit of the private key, the function doubles the generation point. However, if the bit's value is 1, the algorithm adds it to the generation point as well. This is the efficient algorithm for elliptic curve multiplication known as 'double and add' which was mentioned earlier. The algorithm uses number theory which is outside the scope of this paper. It is important to note that this method is much more efficient than normal elliptic curve multiplication.

The result of this function is the user's public key which is known to all other nodes on the network when a transaction occurs.

```python
def EccMultiply(GenPoint,ScalarHex): #Double & add algorithm
    if ScalarHex == 0 or ScalarHex >= N: raise Exception("Invalid Scalar/Private Key")
    ScalarBin = str(bin(ScalarHex))[2:] #Converting to binary
    Q=GenPoint # Start from the common generation point
    for i in range (1, len(ScalarBin)): # This is invented EC multiplication.
        Q=ECdouble(Q); # Add each point to itself
        if ScalarBin[i] == "1":
            Q=ECadd(Q,GenPoint);
    return (Q)
```

Elliptic Curve Multiplication in Python

### 3.4 Elliptic Curve Discrete Logarithm Problem

Since the generation point and a user's public key are known by all other nodes on the network, one might assume that it would be easy to generate a user's private key by simply dividing the generation point by the public key. However, due to the complexity of the invented mathematics of elliptic curves and the extensive use of modular arithmetic, this isn't the case. The ECDSA is a trapdoor function – it is easy to perform in one way, but reversing it is nearly impossible. Trying to reverse the function leads to the elliptic curve discrete logarithm problem (ECDLP).

"ECDLP is the following problem: given two points $P$ and $Q$ on an elliptic curve $E$ defined over a field $F_q$, where $q$ is prime or a prime power, if $P = [m]Q$ for some $m \in Z$, determine $m$" [8]. Simply put, there is no easy way to reverse Elliptic Curve Multiplication, that is, to find the scalar value (private key) that the generation point was multiplied by to calculate the public key. Attempting to do so would involve trying every possible number that the private key could be - brute forcing - as there is no efficient way to reverse the elliptic multiplication function. This is very inconvenient because Bitcoin private keys are 256 bits long.

## 4. Estimating the time taken to break the ECDSA

Since the only way we can break elliptic curve cryptography is through brute force, this experiment aims at exploring how long it would take to find a Bitcoin private key from a given public key.

### 4.1 Setting up the experiment

Brute forcing a 256-bit private key is impossible; if it weren't, we wouldn't be using it for modern-day cryptography. Hence, this experiment was conducted with private keys of smaller bit lengths and the data collected from this experiment was used to predict how long it would take to brute force a full 256-bit private key.

The code obtained from the Congressional Research Institute provided efficient functions for elliptic curve addition, point doubling, and multiplication. The values from the standard domain parameters for Secp256k1 were altered to scale down the problem to bit sizes that modern computers can manage (4-bit, 8-bit, 12-bit, 16-bit, 20-bit, and 24-bit) by changing the number of points in the curve, the length of the private key, and the coordinates of the generator point. The private key was set to half of its maximum possible value in order to find the average time it would take to brute force a key of that bit length. Next, a new Generator Point must be calculated.

### 4.2 Finding the Generator Point

Since the whole problem had to be scaled down, a new generator point must be found. The $x$ and $y$ value for this generator point had to be integers and finding these values manually would take a lot of time. The following segment of code was used to find the smallest possible generator point, allowing the same point to be used for all trials. The code used, and the results obtained, can be seen below.

```
print()
print("******* Finding the Generator Point *********")
Pcurve = 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 -1
for x in range (1, Pcurve):
    y = ((x**3) + 7)**0.5
    if( y == int(y)):
        print("Gx: ",x)
        print ("Gy: ",y)
        break
```

Code used to calculate the Generator Point

```
******* Finding the Generator Point *********
Gx:  380689
Gy:  234885113.0
```

The calculated Generator Point

The output shows that each coordinate of the smallest possible generator point has a 32-bit value (binary signed 2's complement). This value cannot be used because it is bigger than the bit size limit placed on the number of points on the curve. Hence, a generator point with a decimal value must be used. This is not ideal but will still help predict the time it would take to break the ECDSA.

### 4.3 Finding the private key through brute force

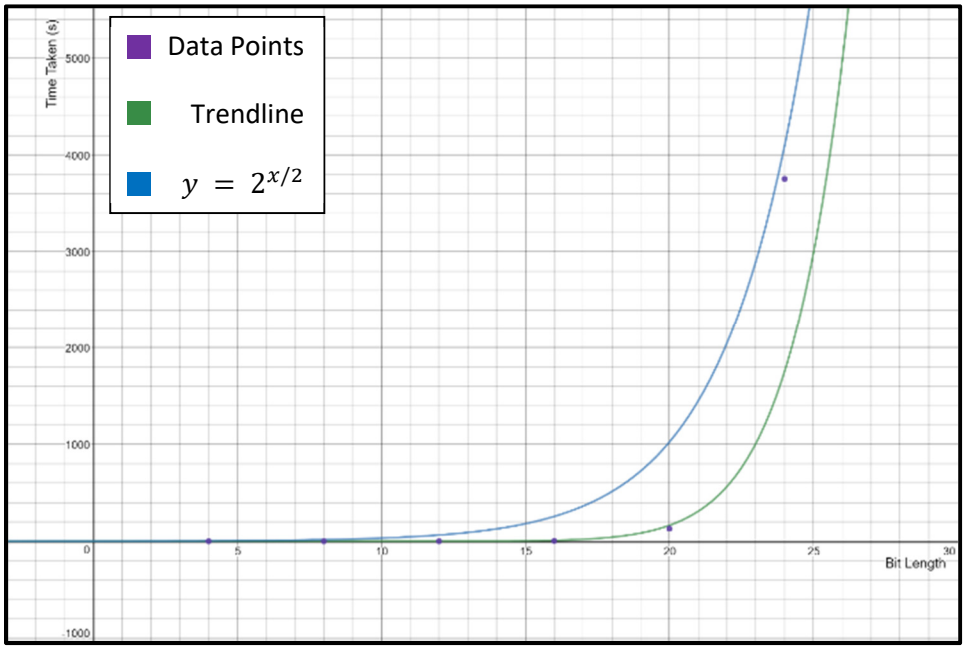
The code written had complexity $O(2^n)$ for the worst case and $O(2^{n/2})$ for the average case. Since a private key that would occur at the average case every time was chosen, the graph should resemble that of $y = 2^{x/2}$. The difference between the trendline of the data and this curve is caused by the processing power of the computer used.

Changing the values of the elliptic curve domain parameters and running the experiment several times - to reduce random errors from experimentation - gave the following results. (Screenshots can be found in the appendix)

| Brute Forcing a Private Key | |
|---|---|
| Bit Length | Time taken (s) |
| 4 | 0.000001 |
| 8 | 0.004015 |
| 12 | 0.105894 |
| 16 | 3.547512 |
| 20 | 129.584898 |
| 24 | 3752.341034 |

This data was then plotted to find a trendline. The equation $y = 2^{x/2}$ was also plotted to see how similar it is to the trendline. The trendline has an equation $y = (2 \times 10^{-15})x^{12.972}$ where $x$ is the bit length of the private key and $y$ is the time taken in seconds. Substituting $x = 256$ into the equation gives the time it would take to break the ECDSA. This gives 34,731,122,970,038,200 seconds, which is equal to 1,614,658,846 years.

This number is huge, and we cannot begin to comprehend its magnitude. To put this number in perspective, if the first human on earth had today's computing power and started trying to find a 256-bit private key using brute force, the probability of them finding the correct one by now would be less than 1%.




Graph of Bit Length and Time Taken

So, is the ECDSA in Bitcoin completely secure? Maybe not. Quantum computing could be the downfall of this algorithm.

# 5. Quantum Computing

## 5.1 Introduction to Quantum Computing

Classical computers work with classical bits which can exist as either 0 or 1. "However, a single quantum bit, or qubit, has the luxury of an infinite choice of so-called superposition states. Nature allows it to have a part corresponding to 0 and a part corresponding to 1 at the same time" [9]. This is known as superposition and these parts are the probabilities of being found in either state. One qubit can exist in a superposition of two states, two qubits can exist in four states, three qubits in eight states, and so on. As long as the qubit remains unobserved, it exists in a superposition and its value cannot be predicted. However, when its value is measured the superposition collapses into either 0 or 1, depending on the probability of being found in that state. "A collection of $n$ qubits is called a quantum register of size $n$" [10].

Quantum gates are similar to logic gates but are used on qubits. They manipulate a qubit's probabilities and give another superposition as an output. When a quantum gate operates on a qubit, it operates on all possible superpositions simultaneously. This is known as quantum parallelism. Hence, if a quantum gate is applied on a two-qubit system, it effectively performs $2^2$ classical computations, on a three-qubit system it performs $2^3$ classical computations, and so on. This number increases exponentially and is the reason for the effectiveness of quantum computers. However, when finding the output of a quantum computer, the state of a random qubit is measured. Hence, algorithms have been developed that increase the probability of measuring the state of the desired qubit.

## 5.2 Shor's Algorithm

With the development of quantum computing, several algorithms have been written which take advantage of the properties of quantum elements. One such algorithm is Shor's algorithm.

Shor's algorithm was originally written to solve the discrete logarithm problem but has been modified in "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms" to solve the ECDLP. The steps outlined in this paper are as follows

1. Create two registers $k$ and $l$ of length $n + 1$ qubits, where $n$ is the number of bits the elliptic curve is defined over.
2. Apply a Hadamard transform to each qubit, which puts them in a superposition of all possible states where the probability of finding the qubit in any of these states is given by

$$\frac{1}{2^{n+1}}$$

We have the Generator Point $g$ and the public key $P$ of a given user and need to find the private key $Q$ of the user. We also know that $P = gQ$ (using elliptic curve multiplication)

3. We create a third register with the value $kg + lP$. We can then substitute $P = gQ$ to get $kg + lgQ$ and factorize $g$ from the equation to get $g(k + lQ)$.
4. Next, we perform a Quantum Fourier Transform on this equation (a complex mathematical function out of the scope of this paper) and measure the state of the first two registers. Now the value of $Q$, which is the user's private key, can be computed from the measurements.

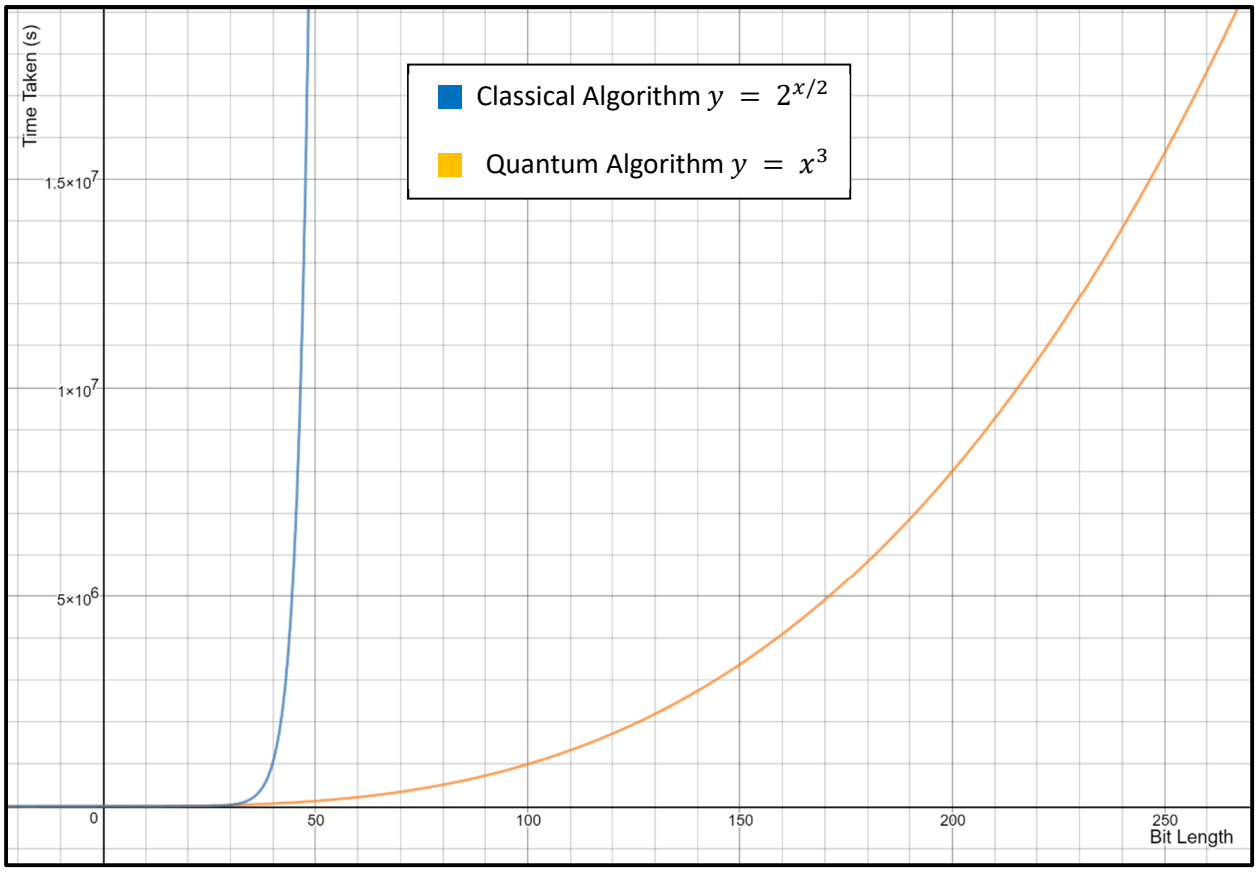
This algorithm can calculate a user's private key from their public key with complexity $O(n^3)$ [11].

### 5.3 Shor's Algorithm vs Classical Brute Force

As compared to the complexity of the classical algorithm $O(2^{n/2})$, Shor's algorithm is a lot faster for larger numbers, as shown in the graph below. We can see that $y = x^3$ grows much slower than $y = 2^{x/2}$. At $x = 256$, the complexity curve for classical computers is nowhere to be found.

For smaller numbers, the classical algorithm is more efficient. However, once the bit length crosses 30, this is no longer the case. The table outlines the average number of steps required for each algorithm.

| Calculating the Private Key | | |
|---|---|---|
| Bit Length (n) | Shor's Algorithm $O(n^3)$ | Classical Algorithm $O(2^{n/2})$ |
| 4 | 64 | 4 |
| 8 | 512 | 16 |
| 12 | 1728 | 64 |
| 16 | 4096 | 256 |
| 20 | 8000 | 1024 |
| 24 | 13824 | 4096 |
| 28 | 21952 | 16384 |
| 32 | 32768 | 65536 |
| 36 | 46656 | 262144 |
| … | … | … |
| 64 | 262144 | 4294967296 |
| 128 | 2097152 | $1.84467 \times 10^{19}$ |
| 256 | 16777216 | $3.40282 \times 10^{38}$ |




Graph of Bit Length and Time Taken for Classical and Quantum Computers

It is evident from the data that Quantum Computers will reduce the security of Bitcoin tremendously because of their enhanced ability to solve the elliptic curve discrete logarithm problem with Shor's Algorithm. Luckily, quantum computers capable of these computations haven't been invented yet.

# 6. Impact of Quantum Computing on Bitcoin

## 6.1 Quantum Attacks

A lot of security mechanisms have been placed in Bitcoin, such as the public blockchain which allows any user to verify the validity of transactions, or the quantum-resistant hash function SHA-256 used by Bitcoin to prevent any tampering of previous transactions. Despite this, if a user - Alice - could guess or calculate another user's - Bob's - private key, then Alice could spend all of Bob's Bitcoins. All the transactions would seem legitimate since Alice would be signing all of them with Bob's public key. This would be a digital form of identity theft and could be performed by cracking the ECDSA. While doing this is infeasible with modern computers, we are making progress in quantum computing and could eventually use Shor's Algorithm to solve the ECDLP.

As found by John Proos and Christof Zalka in their paper "Shor's discrete logarithm quantum algorithm for elliptic curves", the number of qubits required to solve the ECDLP is roughly $6n$, or in Bitcoin's case, roughly 1536 qubits. "IBM Q research has built and tested an operational 50 qubit prototype processor, a huge leap up from its previous record of 17 qubits" [12]. We are far behind the computing requirements for Shor's algorithm, so the open source community that maintains Bitcoin's code doesn't have to worry for now.

However, because of the many benefits that quantum computing would provide, IBM, Microsoft, and other companies are all racing to create quantum computers with many qubits. If these companies are successful, then Bitcoin will have to undergo a lot of change and switch the current ECSDA for another quantum resistant digital signature algorithm.

## 6.2 A Possible Solution

There are a few temporary solutions to this issue, such as using a Bitcoin public key only once. When Alice needs to transfer money to Bob, Bob gives out his Bitcoin address. To produce this address, Bob's public key has been hashed several times over. The address, and hash functions in general, are quantum secure because there is no efficient algorithm (classical or quantum) which can find the input value of a hash function. Hence, even quantum computers will have to resort to brute force.

When Bob wants to send money to someone else, however, then he must give out his public key which will be recorded into the blockchain. A malicious user on the network would then be able to calculate Bob's private key from the public key, provided they have the resources. Therefore, when Bob uses the public key to send out money, he must immediately create a new Bitcoin account and transfer all his money there to keep his money safe.

This is a common solution to this problem and is employed by a few people who suspect that their spending patterns might reveal too much about them, and hence switch their private and public keys every few transactions to protect their privacy and remain anonymous. This practice is even outlined in the 'Protect your privacy' section of the Bitcoin website.

## 7. Conclusion

Bitcoin remains secure against modern-day computing. In the experiment conducted, we saw that it would take approximately 1.6 billion years to calculate a private key from a given public key. To improve the experiment, it could have been performed on multiple computers to reduce the effect of the specific processing power of the computer used. Additionally, the experiment could be run overnight for larger bit lengths to get more data. However, finding the Big O complexity of the algorithm meant that this was no longer required.

Bitcoin remains vulnerable to quantum computers because of Shor's Algorithm, but the advent of these computers seems too far away to worry about. Nevertheless, the development of quantum computing would also mean the development of quantum resistant cryptography. Perhaps in the future, Bitcoin could use Lamport Signatures, a digital signature algorithm which also makes use of hash functions, thus allowing Bitcoin's digital signatures to be quantum resistant.

## 8. Works Cited


[1] "Frequently Asked Questions," [Online]. Available: bitcoin.org/en/faq. [Accessed 2 January 2018].

[2] D. Johnson and A. Menezes, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," 2000.

[3] A. Narayanan, J. Bonneau, E. Felten, A. Miller and S. Goldfeder, Bitcoin and Cryptocurrency Technologies, Princeton University Press, 2016.

[4] M. Amy, O. D. Matteo, V. Gheorghiu, M. Mosca, A. Parent and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3," 2016.

[5] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[6] "How does Bitcoin work?," [Online]. Available: bitcoin.org/en/how-it-works. [Accessed 2 January 2018].

[7] R. Kestenbaum, "Why Bitcoin Is Important For Your Business," 14 March 2017. [Online]. Available: www.forbes.com/sites/richardkestenbaum/2017/03/14/why-bitcoin-is-important-for-your-business/#c86777b41b51. [Accessed 2 January 2018].

[8] M. Musson, "Attacking the Elliptic Curve Discrete Logarithm Problem," 2006.

[9] H.-K. Lo, T. Spiller and S. Popescu, Introduction to Quantum Computation and Information, World Scientific Publishing Co. Re. Ltd., 1998.

[10] A. Ekert, P. Hayden and H. Inamori, "Basic concepts in quantum computation," 2008.

[11] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves," Waterloo, 2008.

[12] S. Dent, "IBM's processor pushes quantum computing closer to 'supremacy'," 10 November 2017. [Online]. Available: www.engadget.com/2017/11/10/ibm-50-qubit-quantum-computer/. [Accessed 2 January 2018].

[13] "Doubling of Point P," 6 November 2013. [Online]. Available: image.slidesharecdn.com/ellipticcurvecryptographyandzeroknowledgeproof-131105012551-phpapp02/95/elliptic-curve-cryptography-and-zero-knowledge-proof-28-638.jpg?cb=1383614911. [Accessed 2 January 2018].

[14] "Elliptic Curve Add," [Online]. Available: 3.bp.blogspot.com/-kbpvaob3pPA/VEeyIPx3abI/AAAAAAAAAAk/3e2Z7Xl4Rms/s1600/elliptic-curve-add.png. [Accessed 2 January 2018].

[15] M. Hughes, "What Are Bitcoins Actually Used For Now in 2016?," 30 March 2016. [Online]. Available: www.makeuseof.com/tag/bitcoins-actually-used-now-2016/. [Accessed 2 January 2018].

[16] Wobine, "PrivateKeyToPublicKey.py," Congresional Research Institute, 13 March 2015. [Online]. Available: github.com/wobine/blackboard101/blob/master/EllipticCurvesPart4-PrivateKeyToPublicKey.py. [Accessed 15 December 2017].

[17] Certicom Research, "Standards for Efficient Cryptography 2 (SEC 2)," 2010.

# 9. Appendix

## 9.1 Domain Parameters for Secp256k1 (Certicom Research)

As excerpted from *Standards*:

The elliptic curve domain parameters over $F_p$ associated with a Koblitz curve secp256k1 are specified by the sextuple T = $(p,a,b,G,n,h)$ where the finite field $F_p$ is defined by:

- $p$ = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
- = $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

The curve $E$: $y^2 = x^3+ax+b$ over $F_p$ is defined by:

- $a$ = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
- $b$ = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

The base point G in compressed form is:

- $G$ = 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798

and in uncompressed form is:

- $G$ = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

Finally the order $n$ of $G$ and the cofactor are:

- $n$ = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
- $h$ = 01

### 9.2 The code used and results of the experiment

The following code was adapted from what was provided by the Congressional Research Institute. This is the experiment performed for 4-bit key sizes. Other experiments were similar to this one.

```
from datetime import datetime

import sys

import os

Pcurve = 2**4 -1 # The proven prime

N=0xF # Number of points in the field - It is the hexadecimal representation of Pcurve

Acurve = 0; Bcurve = 7 # These two defines the elliptic curve. y^2 = x^3 + Acurve * x +
Bcurve

Gx = 0

Gy = 2.646

GPoint = (Gx,Gy) # This is our generator point.

#Individual Transaction/Personal Information

privKey = 0x7

def modinv(a,n=Pcurve): #Extended Euclidean Algorithm/'division' in elliptic curves

    lm, hm = 1,0

    low, high = a%n,n

    while low > 1:

        ratio = int(high/low)

        nm, new = hm-lm*ratio, high-low*ratio

        lm, low, hm, high = nm, new, lm, low

    return lm % n
```

```
def ECadd(a,b): # Not true addition, invented for EC. Could have been called anything.
    LamAdd = ((b[1]-a[1]) * modinv(b[0]-a[0],Pcurve)) % Pcurve
    x = (LamAdd*LamAdd-a[0]-b[0]) % Pcurve
    y = (LamAdd*(a[0]-x)-a[1]) % Pcurve
    return (x,y)
def ECdouble(a): # This is called point doubling, also invented for EC.
    Lam = ((3*a[0]*a[0]+Acurve) * modinv((2*a[1]),Pcurve)) % Pcurve
    x = (Lam*Lam-2*a[0]) % Pcurve
    y = (Lam*(a[0]-x)-a[1]) % Pcurve
    return (x,y)
def EccMultiply(GenPoint,ScalarHex): #Double & add. Not true multiplication
    if ScalarHex == 0 or ScalarHex >= N: raise Exception("Invalid Scalar/Private Key")
    ScalarBin = str(bin(ScalarHex))[2:]
    Q=GenPoint
    for i in range (1, len(ScalarBin)): # This is invented EC multiplication.
        Q=ECdouble(Q);
        if ScalarBin[i] == "1":
            Q=ECadd(Q,GenPoint);
    return (Q)
print(); print( "  ******* Public Key Generation *********");
print()
PublicKey = EccMultiply(GPoint,privKey)
print( "  the private key:");
print( "  ", privKey); print()
print( "  the calculated public key:");
print( "  ",PublicKey); print()
#The experiment starts here
startTime = datetime.now() #Starting the timer
for tempPrivateKey  in range (1,Pcurve): #Checking all possible values
    tempPublicKey = EccMultiply(GPoint,tempPrivateKey)
```

```
    if tempPublicKey == PublicKey:
        print()
        print ("  ******* Brute Force Complete *********")
        print()
        print("  Public Key Input: ")
        print("  ",tempPublicKey)
        print()
        print ("  Private Key Found: ")
        print ("  ",tempPrivateKey)
        print()
        print ("  Total time taken = " , datetime.now() - startTime, " seconds")
         break
         exit()
#End of code
```

Below are screenshots of the results obtained from the experiment

```
******* Public Key Generation *********

the private key:
 7

the calculated public key:
 (13.213194274812679, 5.074667140656167)


******* Brute Force Complete *********

Public Key Input:
 (13.213194274812679, 5.074667140656167)

Private Key Found:
 7

Total time taken =  0:00:00  seconds
```

4-bit key

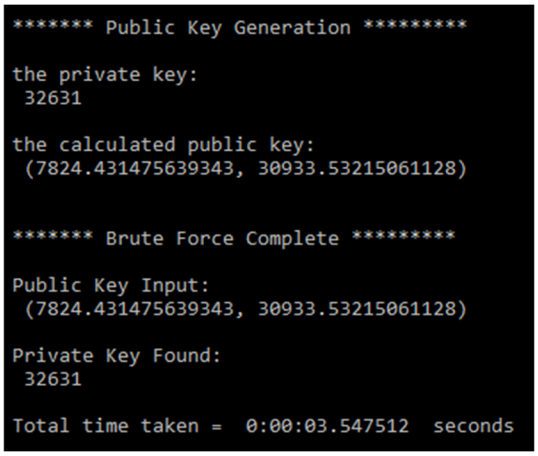
```
******* Public Key Generation *********

the private key:
 119

the calculated public key:
 (152.5414810860093, 126.55067221809804)


******* Brute Force Complete *********

Public Key Input:
 (152.5414810860093, 126.55067221809804)

Private Key Found:
 119

Total time taken =  0:00:00.004015  seconds
```

8-bit key

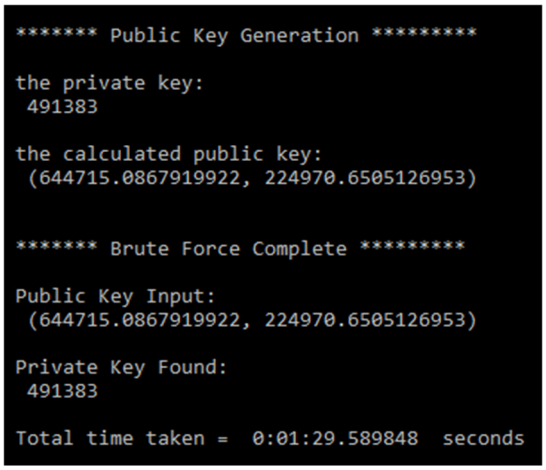
.

```
******* Public Key Generation *********

the private key:
 1911

the calculated public key:
 (3203.79941130057, 1946.3612213241868)


******* Brute Force Complete *********

Public Key Input:
 (3203.79941130057, 1946.3612213241868)

Private Key Found:
 1911

Total time taken =  0:00:00.105894  seconds
```

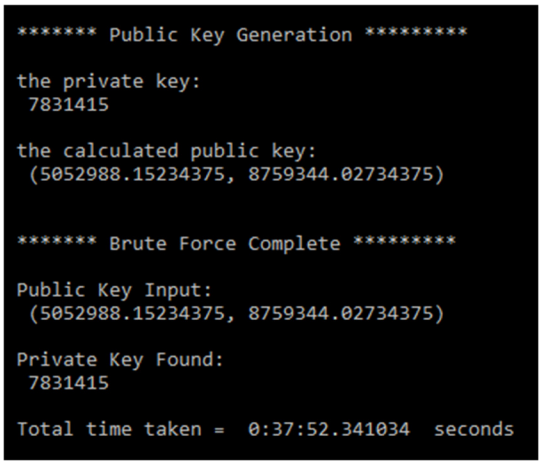
12-bit key




16-bit key




20-bit key




24-bit key