Quantum Information Systems:
The State of Post-Quantum Cryptography as a Means to Combat Shor's Algorithm Run on a Scalable Quantum Computer

Andrew Erickson
Dennis Guster
Leena Radeke
Erich Rice
Information Systems Department
St. Cloud State University

ABSTRACT

A true quantum computer will require quantum algorithms to be implemented, and a major concern related to cryptography is the exponential speed increase that quantum computing could provide. Two quantum algorithms that are creating these concerns are Grover's and Shor's. Both are related to speeding up fundamental computing problems, but the ramifications of the speed-up related to Shor's algorithm could cause a much greater danger than Grover's. These algorithms are unfamiliar to many, and tend to use some form of a stream cipher, which is not sensitive to the use of factoring large numbers. The results found in the subsequent examples show there are post-quantum encryption options that can be easily integrated within a standard LINUX framework. It is the quantum algorithms, designed to compromise the encrypted data that would use quantum algorithms such as Shor's, that will need to run on a quantum computer to be effective. It does appear to be a matter of time before deploying that level of protection becomes critical. A cloud architect should be proactive to this threat, rather than reactive. Therefore, devising proactive procedures for quantum attacks need to be an ongoing process.

Key words: quantum computer, qubit, QCL, Shor's algorithm, Grover's algorithm, post quantum cryptography.

1 INTRODUCTION

Quantum computing has been around for some time, but began receiving attention in the early eighties [6, 9]. While a scalable quantum computer has remained out of reach, special purpose quantum computers have generated interest in the topic [23]. Quantum computing certainly has numerous problems to overcome, but its potential, coupled with quantum algorithms such as Shor's, has created quite a stir from a cryptography standpoint. Without a doubt, there are strengths and weakness related to quantum computing [2]. This takes us to a major concern related to cryptography: the exponential speed increase that quantum computing could provide. A true quantum computer will require quantum algorithms to be implemented. Two quantum algorithms that are creating concerns from a cryptography perspective are Grover's [27] and Shor's [8]. Both are related to speeding up fundamental computing problems.

Specifically, Grover's algorithm has the potential to perform searches more quickly than classical algorithms run on classical computers, when implemented on a quantum. It is expectedly could perform an N-sized search in approximately $\sqrt{N}$ time, which is a quadratic speed-up over classical methods. If Grover's algorithm can reach its full potential, the field of "Big Data" would benefit greatly. There is some thought that Grover's algorithm could be used to compromise symmetric keys utilized in such algorithms as AES-256, and the NIST report on post quantum computing asserts that this problem can be easily combated by increasing the key size [22].

The ramifications of the speed-up related to Shor's algorithm could cause a much greater danger than Grover's. Because Shor's algorithm could factor an n-bit integer in polynomial time, the speed-up over classical methods would be enormous. Currently, a quantum computer with enough accessible qubit space is not available. If available, it would cause several encryption methods relying on factoring large numbers to become obsolete. This may be a genuine problem for common encryption algorithms like RSA, ECDSA and DSA. These algorithms devise keys based on factors become vulnerable [18]. Therefore, Shor's algorithm could provide an exponential speed-up over current methods, allowing the exploitation of many of public-key cryptography methods, including RSA. This is verified in the NIST report [22], which classifies RSA as no longer being secure in a post quantum world.

Subsequently, the primary question is this: how soon might a largescale general purpose become available? There has been considerable progress in the last 5 years, but the field still has a long way to go. A representative example of the challenges in realizing a true quantum computer is reported by Novet [23]. One of the challenges is related to supporting an adequate number of qubits. A qubit is the way that data is represented in a quantum computer. Instead of bits, a quantum computer uses qubits. A qubit is a vast improvement in comparison to the classical bit, and is denser due to its ability to represent multiple states simultaneously. One of the major challenges to the development of a large scale general purpose quantum computer is the problem of de-coherence [10]. This difficulty is related to the fact that the state of the underlying infrastructure (such as a spinning photon) can be easily and inadvertently modified. The quantum elements need to be isolated, or they could be inadvertently measured which would change their state. Providing this de-coherence often requires distinctive design constraints including, for example, an ultra-cold environment.

To some extent, the quantum computing revolution has fallen under the radar of many people. In fact, if one doesn't keep up on the state of encryption methods, one might think that the quantum revolution wasn't even happening. However, there are organizations such as PQCrypto that have understood the danger and have been in existence for over 10 years [24]. This work is often classified as post-quantum encryption. The core of their work focuses on four families of algorithms: hash-based cryptography, code-based cryptography, lattice-based cryptography and multivariate-quadratic-equations cryptography. In all four cases the encryption methodology has been deemed at least quantum computing resistant. For the short term, perhaps as a stop gap measure, there have been efforts to shore up the existing classical methods, such as AES, in which just increasing the key size can be effective.

A good example can be found in the LINUX package “gpg” (gnu pretty good privacy). To improve security, key sizes of up to 4096 bits are supported with existing algorithms. If a full-scale quantum computer is developed, the code encrypted even with this key will still be vulnerable. However, it still provides some protection while qubit sizes of today's computers are limited. It also offers additional protection against attacks by hackers using larger clusters of computers/processors. If a key is compromised, it gives the hackers a short cut the sensitive data.

That being said, the threat created by using Shor's algorithm on a true quantum computer is very concerning. While increasing the key sizes is supported in existing cryptography software suites such as PGP (or the GNU version GPG), support for algorithms that are quantum resistant is not a feature of PGP. There is, however, a software suite entitled codecrypt [25] providing a number of quantum resistant options. Like GPG, CodeCrypt is an open source cryptography suite but it uses quantum resistant algorithms. These algorithms differ in that they are based on the McEliece cryptosystem, which is an asymmetric encryption algorithm specifically using cryptographic primitives deemed unbreakable by quantum computers. For GPG users familiar with LINUX systems, the structure is similar, and the basic learning curve should be minimal. Therefore, the purpose of this paper is to use both GPG and Codecrypt to illustrate how common LINUX-based open source software could be used to provide reasonable protection against quantum based attacks.

2 REVIEW OF LITERATURE

2.1A Brief History of Quantum Computing

The concept of quantum computing is older than one might expect. In fact, its origins begin in the early 1970s. It wasn't even considered practical on any level until the 1980s, when breakthrough research done by Paul Benioff [1], Richard Feynman [9], and Yuri Manin [17] appeared. A key contribution was the work of David Deutsch (1985) which put forward the idea that a universal (or general purpose) quantum computer was theoretically possible. In classical computing terms, this would be analogous to the universal Turing machine [28]. In other words, Deutsch's model suggested that a universal quantum computer would have the ability to simulate any other quantum computer [8].

The interest in quantum computing picked up when Peter Shor [20] devised an integer factorizing algorithm. This caught the attention of many people, as it could be used to break some classical encryption algorithms. Much of the interest was due to the speed potential: Shor's algorithm operates much quicker than classical methods, and can solve complex factors in polynomial time frame [7]. It is expected that when run on a quantum computer with adequate qubit space, the algorithm would be able to crack public-key cryptography in minutes [3]. There are several encryption algorithms effected, but RSA is the most notable.

Subsequently, Lov Grover came up with a very effective quantum database search algorithm [11]. Interestingly, Grover's algorithm can serve as a subroutine to existing classical algorithms. This is an architecture that many expect will facilitate the addition of quantum computing into the mainstream, where classical computing remains dominant and quantum computing serves as a co-processor to aid in speeding up time consuming processes. As early as the 2000s, the basics to illustrate an underlying quantum architecture began to appear. A good example of this is the use of linear optical quantum computing, as implemented with KLM protocol [13]. Specifically, this system uses linear optical elements, photon sources, and photon detectors. Since that time, many advances within the field of quantum computing have taken place. At the forefront is the development of a special purpose quantum computer by the D-wave company [23].

2.2 State of Quantum Computers

While the hype associated with quantum computing continues to grow, a true general purpose physical quantum computer has yet to emerge. Therefore, the full effect of Shor's algorithm for now still appears to be a theoretical construct. Judging from recent postings on the internet, the topic has gained much attention from both a theoretical and practical perspective. The IBM Quantum Experience (QX) project is representative of this attention. The goal of this project is to facilitate knowledge of quantum computing by allowing anyone with an internet connection to gain access to IBM'S quantum processor. From a basic development perspective, this provides a terrific opportunity for anyone connected to the IBM Cloud to run their own experiments on a pseudo-quantum computer. For ease of use, a quantum composer GUI is provided, or of course the quantum computer can be programmed by using the quantum assembly language.

Although they are not true general-purpose quantum computers, special purpose quantum computers are available for commercial use. Perhaps the most recognizable has been developed by D-Wave Systems. Their first attempt, the D-Wave One, could only run discrete optimization math problems. Their second generation, the D-Wave Two, followed and contains a total of 512 qubits. A successful benchmark test run in 2013 showed that the D-Wave was quite functional, and could solve two mathematical problems consisting of 100+ variables in less than second [20]. This roughly translates to about 3,600 times faster than the CPLEX super computer, which performed the same test in about half an hour. The latest version, the D-Wave 2X, contains an impressive 1,000 qubit architecture. The logic behind the 2X was to enact hardware changes dealing with both reliability and speed improvements.

Search giant Google is also involved in quantum computing, and recently has produced a 49-qubit chip. The minimum threshold is fifty qubits, as a quantum computer chip with 50+ qubits would be able to surpass classical computers on selected tasks [5]. Microsoft is also involved in quantum computing, and is in the process of creating a qubit architecture in addition to a quantum programming language that will run on both classical simulators and quantum computers. To ease the learning curve, the language is based on their Visual Studio structure [16].

3 METHODOLOGY AND RESULTS

The goal of this paper is to illustrate how data can be made safe against quantum-based attacks, such as Shor's algorithm. To do this, a series of examples will be presented and delineated. The first example examines a well-known encryption suite PGP (pretty good privacy), and shows there are some things that can be done to provide quantum-resistant encryption without much additional work or training. Because the examples are presented within a LINUX framework, the GNU equivalent of PGP, called GPG, is utilized.

3.1Example 1: Use of classical encryption suites

Perhaps the easiest way to gain some degree of protect again quantum attacks would be to increase the key size. In some cases, the quantum algorithms only offer a quadratic speed-up, and increasing the key size is expected to provide adequate protection in such scenarios [22]. The tried and true GPG command can be used to invoke classical encryption with a large key of 4096 bits, which would obviously be more difficult to factor than smaller keys. In the example below, RSA is selected as the encryption algorithm. This key will be valid for three months, although shorter intervals are certainly possible. The key is protected with a passphrase, which is probably the most vulnerable part of the process. When the key is exported, it is clear we are dealing with a significantly large key!

Of course, when the key size is increased, it will require more resources for the encryption/de-encryption process to occur in a timely manner. This is further complicated by such operational safeguards as padding the key to make it harder to find. If one looks at the key as it is stored in the file ttkey, its size is 3036 bytes and not the expected 512 bytes (or 4096 bits). There may also be a concern with the process used to create random bytes. Computer systems have been known to use device-related seeds that do not produce truly random values. The stop gap measures GPG does offer will not be enough, necessitating the discussion of other options currently available in the next sections.

```
ThomasTortoise@debianTestHost:~$ gpg --gen-key
gpg (GnuPG) 1.4.18; Copyright (C) 2014 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.

What keysize do you want? (2048) 4096
Requested keysize is 4096 bits

Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0) 3m
Key expires at Fri 05 Jan 2018 02:20:36 PM CST
Is this correct? (y/N) y

You need a user ID to identify your key; the software constructs the user ID
from the Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name: Thomas Tortoise
Email address: thetortoiseofdoom@gmail.com
Comment: hey
You selected this USER-ID:
    "Thomas Tortoise (hey) <thetortoiseofdoom@gmail.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
You need a Passphrase to protect your secret key.

We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.

......+++++
.............+++++
gpg: /home/ThomasTortoise/.gnupg/trustdb.gpg: trustdb created
gpg: key 9CC38F7D marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2018-01-05
pub   4096R/9CC38F7D 2017-10-07 [expires: 2018-01-05]
      Key fingerprint = B49B C842 0E62 F353 B5A7  EB15 4A6B 1FA7 9CC3 8F7D
uid                  Thomas Tortoise (hey) <thetortoiseofdoom@gmail.com>
sub   4096R/D1B69BE2 2017-10-07 [expires: 2018-01-05]
```

```
ThomasTortoise@debianTestHost:~$ gpg -a --export 9CC38F7D
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1

mQINBFnZN1cBEAC9xFNrX88xMDb9TENc5/WLVVF3E2OMjALqEcrogD6o8vZ8yCU0
9lWCmZGkamrWVHI91rgH72v+48QlqxBjAtIj6fIewjGED1vToqJXtvtqBShIfqsZ
Middle deleted for space
gDGEI15ZUcs16aqVABR2KFqdXVcpi65Pw2Owkp9PjaCgCiv+Rd4DFdvDQmoCXZdi
60pesnkHRImK6L+91+TzSmGxYiWWCH9ALjIvtknc
=XAk9
-----END PGP PUBLIC KEY BLOCK-----

BCRL\dennis.guster@eros:~$ cat > ttkey
BCRL\dennis.guster@eros:~$ ls -l ttkey
-rw-r--r-- 1 BCRL\dennis.guster BCRL\domain^users 3036 Oct  9 13:23 ttkey
```

3.2Example 2: Use of Post-Quantum Encryption Suites

Although many IT professionals are not aware of it, there already is a suite of encryption algorithms widely available (so far only on UNIX systems) that are quantum computing resistant. This software can be installed on a LINUX VM where the executable is typically stored in /usr/bin. As one would expect, key size is important in assuring robustness. However, there are other options in this suite that go beyond just extending the key size. Although this software is still in a development phase, it certainly is useful from an educational perspective to illustrate how post-quantum algorithms could be deployed, in addition to planning future encryption strategies taking advantage of these options. Below, one can see the version installed is 1.7.6.

```
BCRL\ dennis.guster@eros:~$ whereis ccr
ccr: /usr/bin/ccr /usr/share/man/manl/ccr.l.gs
BCRL\ dennis.guster@eros:~$ ccr -V
codecrypt 1.7.6
```

One of the options that might be considered is a stream cypher. In the examples below, a stream cipher from the ccr suite is used to illustrate how a post-quantum algorithm could be implemented. ChaCha20 is billed as a stream cipher, and is often the algorithm of choice within this category. It is also available in several other security software suites beyond just ccr.

Let's quickly review the difference between a stream cipher and its more popular cousin, the block ciphers. In a block cipher, the algorithm used is designed to work with data units of a fixed size. This is typically a multiple of 8 bytes, such as 64 bytes. A stream cipher can work with just a single, or any size, but each byte is dependent on the prior byte. This means the encryption/decryption cannot start in the middle of the data because of this dependency issue. In block ciphers, the encryption/decryption process can be done in any order, but must adhere to block boundaries based on block size. It is possible for block ciphers to use data of variable size; however, the data will need to be padded so a consistent block size is maintained.

There is a way to allow variable size and dependency, which adds to robustness, called counter mode. When counter mode is employed, it does not directly encrypt the plaintext data. The next step is to take the resulting encryption and XOR it with the data in question. This takes us to ChaCha20, which is classified by the software suites as a stream cipher despite it technically being a block cipher in counter mode. The encryption granularity of the byte level is preserved, but the jump ahead capability on the block level is still maintained. Consequently, the dependence on large factorable keys is not needed, and ChaCha20 can be viewed as a quantum-resistant algorithm. It is still in the development state, and there problems with it being deployed incorrectly. A summary of the problems follows:

- It is implemented as just a block cipher. This does not allow any data size, only the defined block size is allowed.
- It is implemented as a typical stream cipher, but with no jump ahead mechanism.
- It is not designed to work on *Big-Endian* systems (the order bits are stored least or most significant bit) [14].

To get a better understanding of ccr, we can begin by delineating a localized example and then follow with the stream implementation across a network. First, we create our private and public keys for signing and encrypting data with ccr. Note that ChaCha20 is not used directly, but is used as part of the padding (adding extra data to confuse a hacker) process.

3.2.i Localized Example

The first ccr command calls the sig parameter, which uses the following signature algorithm:

SIG = FMTSEQ128C-CUBE256-CUBE128

in which the FMTSEQ series are Merkle-tree signature algorithms. Specifically, the FMTSEQ128C-CUBE256-CUBE128 scheme will provide an attack complexity of approximately 2^{128}. Furthermore, CUBE256 is used as the message digest algorithm, and CUBE128 is used to build the Merkle tree.

The second ccr command calls the enc parameter, which uses the following encryption algorithm:

ENC = MCEQCMDPC128FO-CUBE256-CHACHA20

This algorithm uses a McEliece-based encryption scheme, which is formed using a McEliece trapdoor running on quasi-dyadic Goppa codes (this is the MCEQD- algorithm) and with a quasi-cyclis medium-density parity-check (this is the QCMDPC). The Fujisaki-Okamoto algorithm provides encryption padding. Specifically, within the algorithm MCEQCMDPC128FO, the trapdoor is designed to provide attack complexity around 2^{128}, and CUBE256 and CHACHA20 are the hash and symmetric cipher functions used to support the Fujisaki-Okamoto padding scheme. According to the ccr manual page, algorithms with 2^{128} complexity would require around 1 the trapdoor is designed to provide attack complexity around 10^{22} years of CPU time to break on today's high-speed CPUs.

Note the online documentation for ccr is available, and three examples to call it are provided. The sig and enc examples are shown below.

```
ThomasTortoise@debianTestHost:~/ccr2$ ./ccr -g sig --name "Thomas Tortoise"
Gathering random seed bits from kernel...
If nothing happens, move mouse, type random stuff on keyboard,
or just wait longer.
Seeding done, generating the key...
ThomasTortoise@debianTestHost:~/ccr2$ ./ccr -g enc --name "Thomas Tortoise"
Gathering random seed bits from kernel...
If nothing happens, move mouse, type random stuff on keyboard,
or just wait longer.
Seeding done, generating the key...
BCRL\dennis.guster@eros:~$ ccr –h //basic listing of the switches
BCRL\dennis.guster@eros:~$ ccr -g help //list of encryption schemes
BCRL\dennis.guster@eros:~$ man ccr //detailed manual for ccr
```

In this case, the public key is being exported to the file TomKey.asc, which will be transferred to a trusted user Griff Gryphon. In turn, this user has provided access to his public key in the file GrifKey.asc. Hence, this public key can be uploaded to our keyring with the alias "Good Guy Griff".

```
ThomasTortoise@debianTestHost:~/ccr2$ ./ccr -p -a -o TomKey.asc -F Thomas
ThomasTortoise@debianTestHost:~/ccr2$ ./ccr -ia -R GrifKey.asc --name "Good Guy Griff"
ThomasTortoise@debianTestHost:~/ccr2$ ./ccr -k
pubkey FMTSEQ128C-CUBE256-CUBE128 @701d96a7ee05db7c4fbab0... Thomas Tortoise
pubkey MCEQCMDPC128FO-CUBE256-CHACHA20 @73fbad959b7d6da3803a1f... Thomas Tortoise
pubkey FMTSEQ128C-CUBE256-CUBE128 @0f4f94d10e3ed518aeb734... Good Guy Griff
pubkey MCEQCMDPC128FO-CUBE256-CHACHA20 @331aa2ea4d6c4301dd1919... Good Guy Griff
```

Now the public key for Griff from our keyring can be used to send him an encrypted message. The message in test.doc is encrypted and put into the file MsgForGriff.ccr.

```
ThomasTortoise@debianTestHost:~/ccr2$ ./ccr -se -r Griff < test.doc >
MsgForGriff.ccr
fmtseq notice: 65535 signatures remaining
ThomasTortoise@debianTestHost:~/ccr2$ ls
aclocal.m4 autom4te.cache compile config.status configure.ac depcomp install-sh
m4 Makefile.in MsgForGriff.ccr src test.doc
```

Next, an encrypted message from Griff is sent. This message was encrypted using our public key, and is copied to our ccr directory for decrypting.

```
ThomasTortoise@debianTestHost:/home/ae123/ccr2$ cp MsgForTom.ccr ~/ccr2
ThomasTortoise@debianTestHost:/home/ae123/ccr2$ cd
ThomasTortoise@debianTestHost:~$ cd ccr2
ThomasTortoise@debianTestHost:~/ccr2$ ls
aclocal.m4 autom4te.cache compile config.status configure.ac depcomp install-sh m4
Makefile.in MsgForGriff.ccr README test2.txt TomKey.asc AUTHORS ccr config.guess
config.sub COPYING GrifKey.asc libtool Makefile man MsgForTom.ccr src test.doc
```

Now the decrypt and verify commands can be used to decrypt the message Griff sent.
The output provides the details of its encryption methods, our own public key, and the verification status of the sender's key. You can see its contents below.

```
ThomasTortoise@debianTestHost:~/ccr2$ ./ccr -dv -o DecryptedMessage.doc
<MsgForTom.ccr
incoming encrypted message details:
algorithm: MCEQCMDPC128FO-CUBE256-CHACHA20
recipient: @73fbad959b7d6da3803a1f377253c7eb3aaca51293cd02b308f241bf471f4181
```

```
recipient local name: `Thomas Tortoise'
incoming signed message details:
algorithm: FMTSEQ128C-CUBE256-CUBE128
signed by: @0f4f94d10e3ed518aeb734d583fadc7dfd9507e0c1693cf90e3fc2cad163763a
signed local name: `Good Guy Griff'
verification status: GOOD signature ;-)
ThomasTortoise@debianTestHost:~/ccr2$ ls
aclocal.m4 autom4te.cache compile config.status configure.ac DecryptedMessage.doc
INSTALL ltmain.sh Makefile.am missing NEWS test2enc.txt test.txt
AUTHORS ccr config.guess config.sub COPYING depcomp install-sh m4 Makefile.in
MsgForGriff.ccr README test2.txt TomKey.asc
autogen.sh ChangeLog config.log configure COPYING.LESSER GrifKey.asc libtool
Makefile man MsgForTom.ccr src test.doc
ThomasTortoise@debianTestHost:~/ccr2$ cat DecryptedMessage.doc
hey how's it going?
```

3.2.ii Networked Example

To successfully implement ChaCha20 on a network connection, the tried and true gnu openssl command was used. Unfortunately, the version of this software on our LINUX host does not have ChaCha20 included as part of its default release; we therefore added it manually.

The current version on the LINUX host

```
BCRL\dennis.guster@eros:~$ openssl version
OpenSSL 1.0.1t  3 May 2016
```

The command to search for a particular cipher

```
BCRL\dennis.guster@eros:~$  openssl ciphers | grep ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
```

The search for chacha20 reveals it is not supported

```
BCRL\dennis.guster@eros:~$ openssl ciphers | grep chacha -i
BCRL\dennis.guster@eros:~$
```

The server side of this client/server connection is started with the s_server parameter. In this case, an instance of a server is initialized on the default port of 4433 on any available interface on the current host for testing and debugging. The –rev parameter allows the server to return whatever the client types in reverse format. Note that the protocol cipher list is configured to only support one cipher suite, which includes the ChaCha20 cipher. Therefore, when the client negotiates the connection he/she will have no choice but to select this quantum-resistant cipher. Specifically, the DHE portion of the suite is the authentication method, the RSA portion is for the key exchange, CHACHA20 is the stream cipher, and POLY1305 provides message authentication code. The message authentication code provides a way to verify the fidelity of the ciphered text.

```
Cae123@debianTestHost:~/sslcert$ sudo openssl s_server –rev
[sudo] password for ae123:
Using default temp DH parameters
ACCEPT
CONNECTION ESTABLISHED
Protocol version: TLSv1.2
Client cipher list: DHE-RSA-CHACHA20-POLY1305:SCSV
Ciphersuite: DHE-RSA-CHACHA20-POLY1305
Signature Algorithms:
```

```
RSA+SHA512:DSA+SHA512:ECDSA+SHA512:RSA+SHA384:DSA+SHA384:ECDSA+SHA384:RSA+SHA2
56:DSA+SHA256:ECDSA+SHA256:RSA+SHA224:DSA+SHA224:ECDSA+SHA224:RSA+SHA1:DSA+SHA
1:ECDSA+SHA1
No peer certificate
```

Next, the client can be connected to the server using the s_client command built into openssl. The CHACHA20 encryption suite is specified on the command line and will be used to encrypt traffic. Note this matches the encryption suite listed on the server side. The client will connect to the server port 4433 on the default network, the loopback (127.0.0.1), and use a randomly generated client port for the duration of the session. As part of the process, a certificate is issued to verify the authenticity of the server to the client. This process has limited value in this test scenario because the certificate is simply self-signed. A third trusted party, such as Verisign, would normally sign the certificate.

This process also supports the use of tickets, which like secure shell allow many people to use the same algorithm but customizes it for one individual. Too often, keys don't have a predetermined expiration period, giving hackers a potentially long-time frame to compromise them. Although keys are still used in this process, the session level is managed by tickets. Tickets typically have relatively short lifetimes; that timeframe is 7200 seconds (2 hours) in the example below. At the end, we can see the messages that were exchanged in regular order, and then the reverse of those messages.

```
ae123@debianTestHost:~$ sudo openssl s_client -cipher DHE-RSA-CHACHA20-POLY1305
CONNECTED(00000003)
depth=0 C = te, ST = test, L = test, O = test, OU = 222, CN = test, emailAddress
= test
verify error:num=18:self signed certificate
verify return:1
depth=0 C = te, ST = test, L = test, O = test, OU = 222, CN = test, emailAddress
= test
verify return:1
---
Certificate chain
 0 s:/C=te/ST=test/L=test/O=test/OU=222/CN=test/emailAddress=test
   i:/C=te/ST=test/L=test/O=test/OU=222/CN=test/emailAddress=test
---

Server certificate
-----BEGIN CERTIFICATE-----
MIIDrjCCApagAwIBAgIJAIWhz/HWR57YMA0GCSqGSIb3DQEBCwUAMGwxCzAJBgNV
BAYTAnRlMQ0wCwYDVQQIDAR0ZXN0MQ0wCwYDVQQHDAR0ZXN0MQ0wCwYDVQQKDAR0
ZXN0MQwwCgYDVQQLDAMyMjIxDTALBgNVBAMMBHRlc3QxEzARBgkqhkiG9w0BCQEW
Middle deleted for space
ZMDB3xVyJHD8udTMOQR/t73za7Q59bMsv2iUq5S175zNBQ==
-----END CERTIFICATE-----
subject=/C=te/ST=test/L=test/O=test/OU=222/CN=test/emailAddress=test
issuer=/C=te/ST=test/L=test/O=test/OU=222/CN=test/emailAddress=test
---
No client certificate CA names sent
Peer signing digest: SHA512
Server Temp Key: DH, 2048 bits
---
SSL handshake has read 2293 bytes and written 412 bytes
Verification error: self signed certificate
---
New, TLSv1.2, Cipher is DHE-RSA-CHACHA20-POLY1305
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
```

```
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : DHE-RSA-CHACHA20-POLY1305
    Session-ID: 22F29A11B9A88381535040BFD80EF94C3798C4DB4303E8D2B2B505EE961331FB
    Session-ID-ctx:
    Master-Key:
4F9AF4B4D8103578C60231F5874C4D4C9570EE4BA526F1533E5E2A3E6DA7B898B0952F431C5DBC99B
5C9899CF5B3E8DF
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 7200 (seconds)
    TLS session ticket:
    0000 - a0 e7 94 f4 cc 3b b7 b9-b8 23 08 a1 d8 df d4 58   .....;...#.....X
    0010 - d7 05 92 2f 9d c7 1b 24-a4 9c d0 ff 6f d2 a8 6f   .../...$....o..o
    0020 - 20 46 2f 2e b2 04 09 11-4f cd 06 ec 46 2f 1e 27    F/.....O...F/.'
    0030 - f8 2c e2 22 9e 39 dd f8-42 11 58 1f 39 d7 48 67   .,.".9..B.X.9.Hg
    0040 - 64 5b 88 dc c7 99 03 b7-99 ee 89 16 e4 38 ba 27   d[..........8.'
    0050 - cd 74 a3 aa 31 57 5d c9-a1 08 0b e0 d9 1b 1b a5   .t..1W].........
    0060 - bc bb fc 13 04 5f 47 6b-f7 2b fb 22 9e 16 72 4f   ....._Gk.+."..rO
    0070 - f1 2c 39 aa 41 6c a8 58-c0 c1 eb 20 c8 17 36 9f   .,9.Al.X... ..6.
    0080 - 7c 38 5e 6c b2 bb e4 7e-43 69 7f b0 fe fd 60 0e   |8^l...~Ci....`.
    0090 - 8d 93 6d e5 0e 7d 6e a0-59 b3 07 2f 6f ec 52 45   ..m..}n.Y../o.RE

    Start Time: 1508218038
    Timeout   : 7200 (sec)
    Verify return code: 18 (self signed certificate)
    Extended master secret: yes
---
Let's hope this gets encrypted by the server!
!revres eht yb detpyrcne steg siht epoh s'teL

Wow looks like it was!
!saw ti ekil skool woW
^Cae123@debianTestHost:~$
```

While from the feedback from running the commands above appears to indicate everything has worked so far, it is prudent to verify that everything is operational. To do this, it is logical to observe the data as it crosses the network using a common packet sniffer, such as tcpdump. The data exchange in the previously described server and client session is depicted below. The first packet comes from the client on port 41617 and is directed to the server on port 4433. Its payload is 67 bytes, even though the message itself is only 45 bytes. Clearly, the latter part of the packet containing the payload is not readable, and the encryption appears to be working.

```
ae123@debianTestHost:/usr/bin$ sudo /usr/sbin/tcpdump -i lo 'port 4433' -vvv -X
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
00:27:33.377961 IP (tos 0x0, ttl 64, id 31207, offset 0, flags [DF], proto TCP (6),
length 119)
    localhost.41617 > localhost.4433: Flags [P.], cksum 0xfe6b (incorrect ->
0x9ac0), seq 3026303449:3026303516, ack 4154923002, win 1397, options [nop,nop,TS
val 230346898 ecr 230343167], length 67
        0x0000:  4500 0077 79e7 4000 4006 c297 7f00 0001  E..wy.@.@.......
        0x0010:  7f00 0001 a291 1151 b461 b9d9 f7a7 17fa  .......Q.a......
        0x0020:  8018 0575 fe6b 0000 0101 080a 0dba d092  ...u.k..........
        0x0030:  0dba c1ff 1703 0300 3e66 542d 82ff a2ad  ........>fT-....
        0x0040:  e8ef 3aec 9187 f8e7 0237 075a 6492 6eb7  ..:......7.Zd.n.
        0x0050:  78ed a408 2eff c261 95c1 002c deb3 d0fd  x......a...,....
        0x0060:  143b e3b8 ea2a 5734 5b93 a0f5 490b d594  .;...*W4[...I...
        0x0070:  5966 ad4e afe1 3c                        Yf.N..<
```

4 DISCUSSION AND CONCLUSIONS

The results obtained were interesting in the sense that they showed there are post-quantum encryption options that can be easily integrated within a standard LINUX framework. One would expect overtime that this algorithm may be integrated into a Windows and Apple framework as well. It should be noted that these algorithms are running on a classical architecture. It is the quantum algorithms, designed to compromise the encrypted data that would use quantum algorithms such as Shor's, that will need to run on a quantum computer to be effective. For the most part, the algorithms delineated herein are new to most people. Moreover, they tend to use some form of a stream cipher, which is not sensitive to the use of factoring large numbers. To further ensure fidelity, the algorithms featured multiple layers of encryption. For example, there were four layers in the case of DHE-RSA-CHACHA20-POLY1305. Expressly, the DHE portion of the suite is the authentication method, the RSA portion is for the key exchange, CHACHA20 is the stream cipher and POLY1305 provides message authentication code.

There are certainly better ways to deploy encryption using only the software currently available. In this paper, increasing the key size by utilizing the widely used GPG software suite was delineated. A great guide to making the decision on what encryption algorithms to use is reference [22]. This NIST document proves an excellent quick reference table that can be used to help in the selection process. This begs the question: will companies embrace the post-quantum algorithms, use them for future endeavors, and retrofit them to historical data? Judging from the huge amount of legacy code still in use, it is doubtful the upgrades will take place in a timely manner. One would suspect, after some of the legacy encrypted data is compromised, that action will then begin to occur.

It is clear many people are dreading the arrival of quantum computing, but there is no denying the fact that the advent of this field is on the horizon. There are already numerous commercial products currently available related to quantum computing and quantum data communication. The learning curve for this technology is undoubtedly steeper than it is for classical technology, and only a limited amount of classical information technology transfers to the quantum world. Regardless, quantum computing is the wave of the future, and the time to begin gaining operational knowledge of the quantum world is now. A simple first step would be to use the NIST guide and create a conversion plan so all data in the future is protected by post-quantum encryption algorithms.

As with any type of computing, it is important to verify results. In the case of any type of encryption algorithms, it is vital to observe the data as it is transferred within the internet or intranet. In the above experiment, this was done on the network level using the packet sniffer tcpdump. Viewing the dump data obtained, it certainly was not readable, and the size of the encrypted block was different from the raw data. This indicated that there is not a one-to-one relationship of the encrypted to the decrypted characters, unlike a cypher method. The encrypted data in no way appears as a common ASCII character set, which would stifle unsophisticated hackers. Particularly for the sophisticated hackers that will be in the forefront of utilizing quantum algorithms, it is important that post-quantum algorithms employing multiple layers are used. The fact that the algorithm employed uses padding adds to the fidelity, which could be verified by the larger block size of the encrypted data.

Based on the examples herein, at least for the short term, it appears that there are solutions to protect against quantum-based attacks. However, it does appear to be a matter of time before deploying that level of protection becomes critical. A cloud architect should be proactive to this threat, rather than reactive. Therefore, devising proactive procedures for quantum attacks need to be an ongoing process. It is clear the quantum computing revolution will change the landscape in Information Technology. Based on the authors' observation, spending even a little time evaluating post-quantum encryption techniques can provide a useful foundation in understanding the technology, and thereby prepare for the sophisticated attacks that Shor's algorithm run on a true quantum could impose.

5 REFERENCES

[1] Benioff, P. (1982). Quantum mechanical Hamiltonian models of Turing machines. *Journal of Statistical Physics*, *29*(3), 515-546.

[2] Bennett, C. H., Bernstein, E., Brassard, G., & Vazirani, U. (1997). Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, *26*(5), 1510-1523.

[3] Bernstein, D. J. (2010, May). Grover vs. McEliece. In *International Workshop on Post-Quantum Cryptography* (pp. 73-80). Springer, Berlin, Heidelberg.

[4] Byrne, C. (2015, May). The Golden Age of Quantum Computing is Upon Us (Once We Solve These Tiny Problems). Retrieved from https://www.fastcompany.com/3045708/big-tiny-problems-for-quantum-computing.

[5] Courtland, R. (2017, May 24). Google aims for quantum computing supremacy [News]. *IEEE Spectrum*,*54*(6), 9-10. doi:10.1109/mspec.2017.7934217

[6] Deutsch, D. (1985, July). Quantum theory, the Church-Turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* (Vol. 400, No. 1818, pp. 97-117). The Royal Society.

[7] Deutsch, D., & Jozsa, R. (1992, December). Rapid solution of problems by quantum computation. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* (Vol. 439, No. 1907, pp. 553-558). The Royal Society.

[8] De Wolf, R. (2013). Quantum Computing: Lecture Notes. Retrieved November 2017, from http://homepages.cwi.nl/~rdewolf/qcnotes.pdf.

[9] Feynman, R. P. (1982). Simulating physics with computers. *International journal of theoretical physics*, *21*(6), 467-488.

[10] Gottesman, D. (2009, April). An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics* (Vol. 68, pp. 13-58).

[11] Grover, L. K. (1996, July). A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (pp. 212-219). ACM.

[12] Hardesty, L. (2015). Qubits with staying power. http://news.mit.edu/2015/extended-qubits-quantum-computers-0129.

[13] Knill, E., Laflamme, R., & Milburn, G. J. (2001). A scheme for efficient quantum computation with linear optics. *Nature*. Nature Publishing Group. 409(6816), 46-52.

[14] Insane Coding. (2014). http://insanecoding.blogspot.com/2014/06/avoid-incorrect-chacha20-implementations.html.

[15] Lenstra, H. W., & Pomerance, C. (1992). A rigorous time bound for factoring integers. *Journal of the American Mathematical Society*, *5*(3), 483-516.

[16] Linn, A. (2017, September 25). With new Microsoft breakthroughs, general purpose quantum computing moves closer to reality. https://news.microsoft.com/features/new-microsoft-breakthroughs-general-purpose-quantum-computing-moves-closer-reality/.

[17] Manin, Y. (1980). Vychislimoe i nevychislimoe (computable and noncomputable). *Soviet Radio* (pp. 13–15). In Russian.

[18] Marchenkova, A. (2015, August 15). Break RSA encryption with this one weird trick. Retrieved November 2017, from https://medium.com/quantum-bits/break-rsa-encryption-with-this-one-weird-trick-d955e3394870.

[19] Martinis, J. (2017, 16 May). People of ACM - John Martinis. https://www.acm.org/ articles/people-of-acm/2017/john-martinis.

[20] McGeoch, C. C., & Wang, C. (2013, May). Experimental evaluation of an adiabiatic quantum system for combinatorial optimization. In *Proceedings of the ACM International Conference on Computing Frontiers* (p. 23). ACM.

[21] Miszczak, J. A. (2012). High-level structures for quantum computing. *Synthesis Lectures on Quantum Computing*, *4*(1), 1-129.

[22] NIST Repot on Post-Quantum Computing. (2016). https://csrc.nist.gov/csrc/media/publications/nistir/8105/final/documents/nistir_8105_draft.pdf.

[23] Novet, J. (2015, December 2). Google says its quantum computer is more than 100 million times faster than a regular computer chip. Retrieved November 2017, from http://venturebeat. com/2015/12/08/google-says-its-quantum-computer-is-more-than- 100-million-times-faster-than-a-regular-computer-chip/.

[24] Pqcrypto (2017). https://pqcrypto.org/.

[25] Quantum Resistent Encryption: CodeCrypte. (2013). http://www.privacydusk.com/unix-privacy-tricks/quantum-resistant-encryption-codecrypt/.

[26] Rahmi, M., Shamanta, D., & Tasnim, A. (2012). Basic Quantum Algorithms and Applications. *International Journal of Computer Applications*, *56*(4).

[27] Shor, P. W. (1994, November). Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on* (pp. 124-134). IEEE.

[28] Turing, A.M. (1950). Computing machinery and intelligence. Mind 59 (Oct. 1950), 433-460.