CLOUD QUANTUM COMPUTER

RANDOM NUMBER GENERATION

FOR CRYPTOGRAPHIC PURPOSES

by

Andrew Pham

B.S. Computer Science (California State University, Fullerton) 2019

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Liberal Arts

in Extension Studies

Field of Software Engineering

Cambridge, MA

2021

HARVARD UNIVERSITY

ABSTRACT

CLOUD QUANTUM COMPUTER

RANDOM NUMBER GENERATION

FOR CRYPTOGRAPHIC PURPOSES

By Andrew Pham

Cryptographic techniques used in computers today rely on a key seeded by random number generators with cryptographic strength depending on the randomness of these numbers. Algorithmic methods have been used to generate pseudorandom numbers but unpredictable sensor data from hardware random number generators has been the preferred method of generating random numbers for cryptography. Quantum processes have theoretical non-deterministic fundamental unpredictability possibly making them the ultimate iteration of hardware random number generators. This paper presents a method for utilizing cloud quantum computers newly made available to the public to generate random numbers. Random sequences generated from these computers will be run through a statistical test suite and show that this method [produces / does not produce] cryptographically strong random bits.

Author’s Biographical Sketch

Andrew Pham graduated with a Bachelor of Science in Computer Science from California State University at Fullerton. Directly following his graduation Andrew came to Harvard University to study Software Engineering part-time. During the day he works as a software developer and holds a commission in the US Space Force.

Dedication

This thesis is dedicated to my parents; without their support, none of my studies would have been possible.

Acknowledgments

I’d like to thank all the brilliant professors, lecturers, and fellows at Harvard whose teachings have become the foundation of this this thesis.

# Table of Contents

List of Figures ii

List of Tables iii

Definition of Terms iv

Introduction 1

Chapter I: Random Number Generation in Cryptography 2

Statement of Problem 3

Purpose of Study 3

Chapter II: Cloud Quantum RNG Methodology 12

Rigetti Quantum Computer 13

Hadamard Initialization 21

Data 21

Chapter III: Statistical Testing 40

NIST 41

Tools and Testing Environment 41

Sample Size and Parameters 41

Test Results 43

Chapter IV: Findings and Discussion 60

Summary of Findings 63

Possible Causes of Failure 41

Practicality 71

Future Recommendations 71

Glossary 73

Bibliography 75

Appendix A: Raw FTS Output 77

Appendix B: Truncated Righetti Output 78

Appendix C: asdf 79

List of Figures

List of Tables

Definition of Terms

*CSPRNG*: Cryptographically Secure Pseudorandom Number Generator

*Hadamard Gate:* A quantum logic gate that acts on a single qubit, creating a superposition. The gate maps the qubit basis states  and  to two superposition states with equal weight of the computational basis states.

The Hadamard gate maps the state  to  and  to 

The Hadamard Matrix: 

*HRNG*: Hardware Random Number Generators, a device that generates random numbers from a physical process. Also called True Random Number Generators.

*Input Based RNG Attack:* modify the input of the RNG to put it in a known state allowing the prediction of future output. This can be done by seeding a PRNG or feeding signal to a HRNG.

*NIST*: National Institute of Standards and Technology, a United States government agency that publishes cryptographic standards and guidelines.

*PRNG*: Pseudo-Random Number Generator, a process that generates numbers that looks random but are actually deterministic.

*QRNG*: Quantum Random Number Generator, a HRNG that utilizes the fundamental randomness of quantum mechanics to produce random numbers.

*Qubit:* A quantum analogue of a bit encoding quantum information. A qubit is a two-state quantum mechanical system allowing the bit to be in the two classical states or in superposition of both states simultaneously.

*Randomness Extractor:* A function applied to the output of a weak entropy source to generate a uniform, highly random output. Also known as a unbiasing algorithm.

*RNG*: Random Number Generator, a process that generates random numbers that cannot be predicted better than random chance.

*SDK:* Software Development Kit, a collection of tools in a package that facilities the creation of applications.

*State Compromise Extension RNG Attack*: when the internal secret state of the RNG is known and can be used to predict future output.

Introduction

Random numbers have a wide variety of applications from simulations to sampling but one of the most impactful and every day of uses is encryption. The foundation of modern cryptographic security relies on random number generation. In cryptography, a key is used to encrypt and decrypt information (E. Barker, 2020a). It works analogously to the combination of a lock. On a computer, keys look like a string of random characters. Guessing a key will allow the attacker to decrypt information. Keys must be random to be secure because if the attacker has information on which keys are more or less likely, it makes guessing the key easier (E. Barker, 2020b). Bad RNGs can and have been the demise of many encrypted systems. Here we propose a method of random number generation that theoretically produces random numbers through quantum physical processes utilizing cloud technology recently introduced to the public.

As theorized by… Quantum Random Number Generation by Ma, Xiongfeng, Xiao yuan, Zhu Cao, Bing Qu, and Zhen Zhang (Ma et al., 2016): Details general approaches to creating QRNGs incorporating aspects such as trustworthiness and random number generation speed.

Chapter 1

TODO problem statement

TODO Purpose of study

The process of generating random numbers has been divided into two main approaches, PRNGs and HRNGs. PRNGs utilize an algorithm to generate “random” numbers, or simply a series of mathematical formulas. However, by nature of an algorithm, PRNGs are deterministic in nature and can be predicted if the state of the PRNG is known. Thus, PRNGs require inputs called seeds which adds unpredictability. The seed itself must be random and unpredictable. Therefore, PRNGs are often seeded with a HRNG. Many times, PRNGs have better statistical properties for randomness and produce random numbers faster than pure HRNGs.

HRNGs produce random numbers by taking data from a physical process. Usually, sensors are trained on statistically random signals such as thermal noise. HRNG often relies on processes that are difficult to simulate and model but may not inherently be random, such as camera data pointed at an entropic source (Noll et al., 1998). It’s worth noting that random data has a high level of entropy, but data that has high entropy is not necessarily very random. There are requirements other than entropy that are recommended for cryptographic RNGs (E. B. Barker & Kelsey, 2015). One can also subvert a HRNG by inducing signals from its supposedly random source, for example shining in a light in our previously mentioned HRNG camera sensor. Other methods of HRNG come from weakly random sources such as keyboard delays or disk drive timing information. These methods need to be run through a randomness extractor to pass for use in cryptographic standards (Trevisan & Vadhan, 2000).

An ideal HRNG cannot be controlled, calculated, or predicted. The only processes known to be fundamentally random are those of quantum measurements and observation. The generation of genuine randomness is generally considered impossible with only classical means. Multiple measurements made on quantum processes in identical states will not always give the same result. This means that theoretically QRNGs are impervious to known or forced state attacks such as state compromise extension attacks and input-based attacks because the state cannot be used to predict or dictate future behavior. Because of this property, QRNGs have been said to be the final iteration of random number generators. Today, many companies sell QRNG hardware that utilizes quantum phenomena; however, secure QRNG hardware is not widespread for the average consumer.

As described in… Implications of Quantum Superposition in Cryptography: A True Random Number Generation Algorithm by Dhananjay S. Deshpande, Aman Kumar Nirala, and Ayodeji Olalekan Salau (Deshpande et al., 2020, p. 421): Used QUISKET to create a QRNG on IBM’s cloud quantum computer, IBM-Q experience. Paper also explains the physical and mathematical aspects of superposition and RNG. The paper shows in theory that RNG for cryptography is possible on quantum computers. The paper utilizes the same algorithm as proposed in this thesis for RNG by applying the Hadamard gate on a qubit. However, the paper does not run statistical testing on the output and the IBM-Q computer was a 15 qubit system, limiting their output.

The release of quantum computers available in the cloud could bring this technology to everyday users. To test if these new methods are suitable for cryptographic purposes, we will run them through the NIST Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. If the numbers generated pass the tests in the suite, we deem the technique ready for widespread application and allow for the adoption of quantum RNG on a universal basis making random number generation one of the first real world applications of quantum computers.

Other attempts at creating cryptographically secure random numbers with quantum computers have failed due to noise skewing the output, or lack of sample size. See prior works in section IX for more details. Now with the release of Righetti’s quantum computer, we hope to overcome these obstacles.

Chapter II: Cloud Quantum RNG Methodology

Random numbers will be created inside of a quantum computer by measuring a qubit in superposition. The quantum computer will be from Amazon Braket, a platform from Amazon Web Services providing on demand cloud computing. The specific computer for the implementation will be D-Wave's Advantage system. After random number data is collected, the sequences will be evaluated via the NIST Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.

The random numbers will be generated inside of the quantum computer by measuring qubits after Hadamard initialization. Quantum computers encode information with qubits, basic units of quantum information. Using qubits, we can utilize quantum properties such as entanglement and superposition. We are most interested in the property of superposition which puts the qubits into a probabilistic state. To put a qubit into superposition we operate on them with a Hadamard gate. This is commonly called Hadamard initialization. The Hadamard gate puts the qubit into equal superposition between the 0 state and the 1 state (Brylinski & Chen, 2019). In other words, when measured the qubit has an equal chance of collapsing into the 0 state and the 1 state.

The Amazon Braket SDK has built in Hadamard circuits that we can apply to each numbered qubit. Once each qubit is in superposition, we measure the state, collapsing the state into either 0 or 1. These measurements will be stored in sequences for our testing.

Each sequence will be stored as an ASCII sequence of 0s and 1s. We will store 100 sequences of 10,000 random bits to meet our criteria for significance level and testing requirements described later. Specifically, these sequences will be stored in the Amazon Bucket service once the task is completed by the quantum computer.

Chapter III: Statistical Testing

NIST

The National Institute of Standards and Technology is a physical science lab and an agency of the US Department of Commerce. Their mission is to promote innovation and publish science in a wide variety of fields to further that mission. The NIST has recognized expertise in the field of cryptography and publishes a standard for testing random number generators along with a recommendation of cryptographically secure random number generators for use. These papers are in the public domain and the process, source code, and standards are transparent for public scrutiny.

Tools and Testing Environment

The NIST Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications (STS) is a testing suite created by the NIST for determining whether a random number generator is suitable for cryptographic applications (Bassham et al., 2010). While others have devised tests and test suites for randomness, we have chosen this test suite because the NIST is a US government agency, has strong documentation, and provides tests specifically for cryptographic applications.

Faster Randomness Testing (FST) is a project from the Faculty of Informatics at Masaryk University optimizing the NIST statistical tests for randomness (Sýs & Říha, 2014). In our testing we will be using the Visual Studio solution of their Faster Randomness Testing Suite based off STS 2.1.2. FST functionally runs the same tests as STS.

Testing Environment Table

|  |  |
| --- | --- |
| Faster Randomness Testing | 2.1.2 |
| Visual Studio | Microsoft Visual Studio Enterprise 2019  Version 16.11.1 |
| Windows | Microsoft Windows 10 Pro |
| System | AMD Ryzen 3600 6-core Processor, 3593 Mhz  16 GB physical memory |

NIST Test Quick Reference

|  |  |
| --- | --- |
| Test | Purpose |
| Frequency (Monobits) Test | To determine whether that number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence. The test assesses the closeness of the proportion of ones to 0.5, that is, the number of ones and zeroes in a sequence should be about the same. |
| Test for Frequency within a Block | To determine whether the frequency of ones is an M-bit block is approximately M/2. |
| Runs Test | To determine whether the oscillation between such substrings is too fast or too slow. |
| Test for the Longest Run of Ones in a Block | To determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence. |
| Random Binary Matrix Rank Test | To check for linear dependence among fixed length substrings of the original sequence. |
| Discrete Fourier Transform (Spectral) Test | To detect periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness. |
| Non-overlapping Template Matching | To reject sequences that exhibit too many occurrences of a given non-periodic (aperiodic) pattern. |
| Overlapping Template Matching Test | To reject sequences that show deviations from the expected number of runs of ones of a given length. |
| Maurer's Universal Statistical Test | To detect whether the sequence can be significantly compressed without loss of information. An overly compressible sequence is non-random. |
| Lempel-Ziv Complexity Test | To determine how far the tested sequence can be compressed. The sequence is non-random if it can be significantly compressed |
| Linear Complexity Test | To determine whether the sequence is complex enough to be considered random. |
| Serial Test | To determine whether the number of occurrences of the 2m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. The pattern can overlap. |
| Approximate Entropy Test | To compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and m+1) against the expected result for a random sequence. |
| Cumulative Sum (Cusum) Test | To determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. |

Sample Size and Other Testing Parameters

The significance level for these tests will be set to 0.01, the minimum recommended by the NIST. “The sample should be on the order of the inverse of the significance level”, following this rule, we will have 100 sequences in our sample size. This means each test will be run on 100 different sequences. For our RNG to be suitable for cryptographic applications, the random sequences must pass the entire suite with an approximately 96% pass rate.

Each test in SP 800-22 dictates a minimum number of bits required for evaluation. The overlapping template matching test, linear complexity test, and random excursion tests require the most bits, 1,000,000 at minimum.

|  |  |
| --- | --- |
| Test | Minimum Input Size Recommendation for Each Sequence in bits |
| Frequency Test | 100 |
| Frequency Test within a Block | 100 |
| Runs Test | 100 |
| Test for the Longest Run of Ones in a Block | 128, for the smallest preset value of M=8 |
| Binary Matrix Rank Test | 38,912 bits  n>= 38MQ (M and Q are coded as 32 rn) |
| Discrete Fourier Transform Test | 1000 bits |
| Non-overlapping Template Matching Test | For the default setting N=8 (number of independent blocks), n=8M (M is block length). Since we cannot have a fractional number of bits as the block length, n must be at least 8 |
| Overlapping Template Matching Test | TODO,,, I think 100? |
| Maurer’s Universal Statistical Test | This test requires a long sequence of bits (n ≥ (Q + K) L) which are divided into two segments of L-bit blocks, where L should at least 6. For an L of 6, n should be 387,840 as prescribed by the table in NIST-SP 800-22 2.9.7 |
| Linear Complexity Test | 1,000,000 |
| Serial Test | m < floor(log2 n) -2  If m=1, we have the same test as the frequency test so we must use at least m=2 for any new results. For m=2, n (our input size) must be greater than 16 |
| Approximate Entropy Test | m < floor(log2 n) – 5  No minimum input recommendation |
| Cumulative Sums Test | 100 |
| Random Excursions Test | 1,000,000 |
| Random Excursions Variant Test | 1,000,000 |

Grouping the tests by minimum input size, we find 9 tests can be run with less than 1,000 bits per sequence, 11 tests can be run with less than 500,000 bits per sequence, and all 15 tests can be run with 1,000,000 bits per sequence. We are going to first test the 9 that can be run with less than 1,000 bits per sequence because of cost limitations. If most of those 9 tests pass, we can continue the analysis; if not, there would be no need to continue with more tests because the sequences are not random.

The Rigetti Quantum Computer charges $0.30 per task and $0.00035 per shot. A shot is a single execution of the quantum circuit. We pay to execute a task, our quantum circuit, then for repetitions of that task or shots. Since the task is only factored in once, task cost is negligible compared to the final cost so we will exclude it from our cost calculations. The Rigetti Quantum Computer currently has 32 qubits, meaning we can generate 32 random bits at a time. To calculate our cost per random bit generated we divide the shot cost ($0.00035) by 32, giving us a cost of $0.0000109375 per bit. Generating 1,000,000 bits would cost $10.94. If we wanted to run every test in the test suite for the minimum significance level recommended (0.01), we would need 100 sequences of 1,000,000 bits, which would cost $1,093.75

As mentioned before we will run the 9 tests with the lowest bitstream length requirements. We’ve generated 1,000,000 bits by running 31,250 shots of the circuit. Those 1,000,000 bits are divided into 100 sequences of length 1,000 bits. We will use this sample size for the following tests: TODO. TODO TESTS THAT REQUIRE SAMPLE SIZE MODIFICATION

Testing Parameters Full Calculations

|  |  |
| --- | --- |
| Test | Parameters |
| Frequency Test |  |
| Frequency Test within a Block |  |
| Runs Test |  |
| Test for the Longest Run of Ones in a Block |  |
| Binary Matrix Rank Test |  |
| Discrete Fourier Transform Test |  |
| Non-overlapping Template Matching Test |  |
| Overlapping Template Matching Test |  |
| Maurer’s Universal Statistical Test |  |
| Linear Complexity Test |  |
| Serial Test |  |
| Approximate Entropy Test |  |
| Cumulative Sums Test |  |
| Random Excursions Test |  |
| Random Excursions Variant Test |  |

Testing Parameters Quick Reference

|  |  |
| --- | --- |
| Test | Parameters |
| Frequency Test |  |
| Frequency Test within a Block |  |
| Runs Test |  |
| Test for the Longest Run of Ones in a Block |  |
| Binary Matrix Rank Test |  |
| Discrete Fourier Transform Test |  |
| Non-overlapping Template Matching Test |  |
| Overlapping Template Matching Test |  |
| Maurer’s Universal Statistical Test |  |
| Linear Complexity Test |  |
| Serial Test |  |
| Approximate Entropy Test |  |
| Cumulative Sums Test |  |
| Random Excursions Test |  |
| Random Excursions Variant Test |  |

The tests in 800-22 are formulated to test the null hypothesis, “the sequence is random”. The alternate hypothesis is that “the sequence is not random”. If every test accepts the null hypothesis, cloud quantum computers can generate random numbers for cryptographic applications.

Chapter IV: Findings and Discussion

The minimum pass rate for each statistical test included here is approximately 96.0150% for a sample size of 100 binary sequences.

Results Table

|  |  |
| --- | --- |
| Test | Proportion of Passing Sequences |
| Frequency Test | 49%\* |
| Frequency Test within a Block | 89%\* |
| Runs Test | 96.25% |
| Test for the Longest Run of Ones in a Block | 95%\* |
| Binary Matrix Rank Test |  |
| Discrete Fourier Transform Test | 92%\* |
| Non-overlapping Template Matching Test | In the 148 templates tested, 108 had a passing proportion of passing sequences (40 failing) |
| Overlapping Template Matching Test |  |
| Maurer’s Universal Statistical Test | x |
| Linear Complexity Test | x |
| Serial Test | 99% |
| Approximate Entropy Test | 94%\* |
| Cumulative Sums Test | 53%\* |
| Random Excursions Test | x |
| Random Excursions Variant Test | x |

\*Failure, proportion of passing sequences under 96.0150%

Description of Findings

Test name, what does it test for, what operations does the test carry out, what parameters did we use, did it pass or fail, by how much

Frequency test, our findings were similar to…. True Random Number Generator using Superconducting Qubits by Abdullah Ash-Saki, Mahabubul Alam, and Swaroop Ghosh (Ash-Saki et al., 2019). Generated random numbers using superconducting qubits from IBM. Ran statistical testing on the samples, which failed, then gave detailed analysis. Their experiments showed a deviation of 35% from ideal 1/0 ratio due to noise. They also propose techniques to improve the 1/0 ratio and pass the NIST statistical tests.

Further research

Findings matched… Quantum Random Number Generation with the Superconducting Quantum Computer IBM 20Q Tokyo by Kentaro Tamura and Yutaka Shikano (Tamura et al., 2020): Utilizes the IBM 20Q Tokyo for QRNG with Hadamard initialization then runs the NIST tests to analyze the output, an identical approach to this thesis proposal. Using the 20 qubit quantum computer they obtained a sample length of 43,560 bits. Their statistical analysis showed that the sample was biased and correlated. They observed that their sample was not uniform and failed at least 4 of the NIST Test Suite’s tests; however, they only applied the first 6 of 15 tests to the sample. They also revealed that passing the tests required both the von Neumann and Samuelson randomness extractors, though the effectiveness of this method is unclear. The primary limitation of this study was a limited sample size which we can overcome with quantum computers realizing more qubits for computation…. So maybe this could be done with a randomness extractor

Summary

Bibliography

Ash-Saki, A., Alam, M., & Ghosh, S. (2019). True Random Number Generator using Superconducting Qubits. *2019 Device Research Conference (DRC)*. <https://doi.org/10.1109/drc46940.2019.9046456>

Barker, E. (2020a). Guideline for using cryptographic standards in the federal government: *NIST Special Publication*. <https://doi.org/10.6028/nist.sp.800-175br1>

Barker, E. (2020b). Recommendation for key management: *NIST Special Publication*. <https://doi.org/10.6028/nist.sp.800-57pt1r5>

Barker, E. B., & Kelsey, J. M. (2015). Recommendation for Random Number Generation Using Deterministic Random Bit Generators. *NIST Special Publication*. <https://doi.org/10.6028/nist.sp.800-90ar1>

Bassham, L. E., Rukhin, A. L., Soto, J., Nechvatal, J. R., Smid, M. E., Barker, E. B., Leigh, S. D., Levenson, M., Vangel, M., Banks, D. L., Heckert, N. A., Dray, J. F., & Vo, S. (2010). A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST Special Publication.* <https://doi.org/10.6028/nist.sp.800-22r1a>

Brylinski, R. K., & Chen, G. (2019). *Mathematics of Quantum Computation (Computational Mathematics)* (1st ed.). Chapman and Hall/CRC.

Deshpande, D. S., Nirala, A. K., & Salau, A. O. (2020). Implications of Quantum Superposition in Cryptography: A True Random Number Generation Algorithm. *Information and Communication Technology for Intelligent Systems*, 419–431. <https://doi.org/10.1007/978-981-15-7062-9_41>

Haw, J., Assad, S., Lance, A., Ng, N., Sharma, V., Lam, P., & Symul, T. (2015). Maximization of Extractable Randomness in a Quantum Random-Number Generator. *Physical Review Applied*, *3*(5), 1. <https://doi.org/10.1103/physrevapplied.3.054004>

Ma, X., Yuan, X., Cao, Z., Qi, B., & Zhang, Z. (2016). Quantum random number generation. *Npj Quantum Information*, *2*(1). <https://doi.org/10.1038/npjqi.2016.21>

Noll, L. C. N., Mende, R. G. M., & Sisodiya, S. S. (1998). *Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system* (U.S. Patent 5,732,138). U.S. Patent and Trademark Office.

Sýs, M., & Říha, Z. (2014). Faster Randomness Testing with the NIST Statistical Test Suite. *Security, Privacy, and Applied Cryptography Engineering*, 272–284. <https://doi.org/10.1007/978-3-319-12060-7_18>

Tamura, K., & Shikano, Y. (2020). Quantum Random Number Generation with the Superconducting Quantum Computer IBM 20Q Tokyo. *IACR Cryptol. ePrint Arch., 2020*, 78.

Trevisan, L., & Vadhan, S. (2000). Extracting randomness from samplable distributions. *Proceedings 41st Annual Symposium on Foundations of Computer Science*. https://doi.org/10.1109/sfcs.2000.892063