A CONSOLIDATED SOLUTION FOR BROADER STATISTICAL TESTING OF RANDOM NUMBER GENERATORS IN A POST-QUANTUM ERA

John Edward Naizer

A THESIS

Presented to the Faculty of Miami University in partial
fulfillment of the requirements
for the degree of

Master of Science

Department of Computer Science & Software Engineering

The Graduate School
Miami University
Oxford, OH

2024

Dr. Khodakhast Bibak, Advisor
Dr. James Kiper, Reader
Dr. Suman Bhunia, Reader
Dr. Honglu Jiang, Reader

ABSTRACT

Pseudo-random number generators (PRNGs) and, more recently, quantum random number generators (QRNGs) play critical roles in ensuring the security of systems. Ensuring that the generated randomness from these devices meets the benchmark of cryptographically secure randomness is of utmost importance. To meet this benchmark, the National Institute of Standards and Technology (NIST) has standardized a series of rigorous statistical tests to determine if a random output meets the criteria for use in cryptographically secure applications. However, this benchmark has been continuously evolving with recent advancements in the quantum realm, demanding increased coverage and rigor in statistical testing. We utilize prior work that has contributed to this demand to explore more efficient and rigorous ways to test contemporary random number generators and analyze how they perform when tested beyond what is required by standardization bodies like NIST. We introduce a Command Line Interface (CLI) tool for users to easily apply a myriad of different testing suites on randomly generated binary data. Additionally, we use this tool to expand statistical testing analysis on a few modern PRNGs in tandem with comparative analysis against existing statistical analysis data on certain QRNGs.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the past decade, companies and organizations have been taking advantage of a increasingly popular way to store and compute data which is known today as the cloud [5]. At a surface level, the cloud is essentially a virtual environment that one can imagine being up in the clouds compared to on one's actual machine which provides a safe space to store and manage your data [5]. There are many benefits to using the cloud including larger storage options for one's data as well as faster download and upload speeds [5]. However, a recently emerging technology known as quantum computing has already begun to threaten the very infrastructure of the cloud including the security and integrity of user data [6].

Quantum computing allows nefarious users to break conventional security protocols and algorithms that were previously sound compared to classical computing [6]. They do this by taking advantage of the many benefits that quantum computing provides over classical computing including solving considerably complex problems exceedingly faster and being able to process larger sets of data with ease [6]. The conventional security protocols and algorithms used to protect sensitive data rely on these complex problems to absolutely prevent nefarious users from accessing the data, and the fact that quantum computing can solve these complex problems exceedingly faster presents a very clear problem to data security [6]. In order to combat the ever-growing threats to data security in the cloud, new and improved protocols and algorithms backed by even more complex problems must be explored and implemented so that not even quantum computing techniques can be exploited to expose the sensitive data [6].

The solution to providing unbreakable data security resides in the realm of cryptography. Cryptography handles the masking or encryption of sensitive data so that a nefarious user cannot reveal or decrypt the contents of the data for their own use [7]. The considerably complex problems that security protocols and algorithms use are that of different encryption techniques [7]. Many encryption techniques are used specifically for various purposes depending on the applications of where the data is stored and how it is accessed [7]. Researchers have been actively modeling and testing new encryption techniques to deter quantum computing attacks as well as using quantum computing to deploy attacks on different encryption techniques to determine how sound they are [7].

Exploring the robustness of pseudo-random number generators (PRNGs) and quantum random number generators (QRNGs) stems from the increasing threat posed by quantum computing to cloud infrastructure and data security [8]. Quantum computing's ability to break conventional security protocols and algorithms at a rapid pace presents a significant challenge to safeguarding

sensitive data in the cloud [8]. To counter these threats, it is essential to ensure that the randomness generated by PRNGs and QRNGs meets the highest standards of cryptographic security. By delving into the realm of statistical testing, the goal is to apply modern testing suites that rigorously evaluate the quality of randomness produced by these generators. This research focuses on expanding the coverage and rigor of statistical tests to fortify data security, ensuring that even with the advancements in quantum technologies, the integrity and quality of data storage and management in cloud services remain robust and secure.

The motivation behind this type of research is to ensure the protection and quality of sensitive user data in a post-quantum world which is exactly where this paper's purpose lies.

1.2 Contributions

The contributions of this proposed thesis project are as follows:

- A Command Line Interface (CLI) tool that enables users to seamlessly test random binary data across multiple contemporary statistical testing suites.
- Statistical and computational analysis on selected RNGs using selected statistical testing suites utilizing the CLI tool.
- Comparative analysis between statistical testing analysis from selected RNGs and existing statistical testing analysis on QRNGs.

Chapter 2

Background & Related Work

2.1 The Cloud

The cloud is a ubiquitous technology that has become increasingly prominent in today's world. At a high level the cloud is a network of remote servers that are used to store, manage, and process data and applications, rather than relying on local hardware and infrastructure [9]. Thus, this technology provides a scalable, flexible, and cost-effective way to access resources and services remotely with little overhead cost for the customer [9]. From social media and business platforms to enterprise software and government systems, the cloud has become an important part of our digital landscape, enabling us to store and access data and collaborate on resources in real-time. With the rise of remote work and digital transformation initiatives, the cloud has become even more essential, allowing individuals and organizations to access and collaborate on data and resources from anywhere in the world [9].

2.1.1 Cloud Computing

Cloud computing is a model of computing where computing resources and computing power are provided over the internet as a service. There are three core services that cloud computing provides:

Software as a Service (SaaS): A cloud-based software delivery model that enables users to access and run applications through internet-connected devices without the need for purchasing or installing physical software on-premises upfront [9].

Platform as a Service (PaaS): A cloud computing model where a third-party cloud service provider (CSP) provides an environment on a pay-as-you-go basis, allowing customers to build, develop, run, and manage their applications [9].

Infrastructure as a Service (IaaS): A cloud computing model where a third-party cloud service provider (CSP) offers virtualized computing resources, including servers, data storage, and network equipment, to customers on demand over the internet [9].

The cloud computing market in the United States has seen tremendous growth over the past few years and is only projected to grow at an increasing pace. Customer spending on public cloud services is expected to grow 20.7% to a total spending of around $600 billion in 2023 [10]. Out of the three cloud services Infrastructure-as-a-service is expected to observe the highest customer spending growth in 2023 at 29.8%, although all services are expected to see growth [10].

2.1.2 Shared Responsibility Model

There are many shared responsibilities between the CSP and the customer, and the *Shared Responsibility Model* is a framework that was created in order to differentiate which of these responsibilities fall on them, respectively. The model requires that the CSP must be responsible for the security and infrastructure of the cloud environment while requiring the customer to be responsible for protecting the sensitive data and other assets that they store in the cloud [9]. One common misunderstanding between the customers and the CSP is that cloud workloads, typically applications that utilize data, are protected under the CSP [9]. Under this false assumption, customers are at risk for running cloud workloads with sensitive data in the public cloud which puts the sensitive data in a vulnerable state [9]. It is important to note that each of the three core services that cloud computing provides are subject to the concept of shared responsibility [9].

Some of the areas that a user are responsible for include [9]:

- Managing user credentials and user security
- Network and endpoint security
- Any code that might be used for cloud workloads

Some of the areas that a CSP are responsible for include [9]:

- All hardware and cloud infrastructure
- Data centers and facilities that utilize cloud resources
- Cloud virtualization layer between the hardware and software

Although the *Shared Responsibility Model* requires careful consideration and analysis for the CSP and customer, there are many immediate benefits that it provides [9]:

Efficiency: While customers hold a considerable amount of responsibility in the *Shared Responsibility Model*, certain important aspects of security such as hardware, infrastructure, and virtualization layer are mainly handled by the CSP [9]. In a conventional on-premises model, the customer is primarily responsible for managing these aspects. However, by moving to the cloud, IT staff can now concentrate on other tasks and requirements while allocating resources and investments to areas that fall under their responsibility [9].

Enhanced Protection: Cloud service providers are highly attentive to the safety of their cloud environment and usually allocate substantial resources to guarantee complete protection for their clients [9]. As a part of the service agreement, CSPs have much more time to perform comprehensive monitoring and testing, along with prompt patching and updating [9].

Expertise: CSPs frequently possess greater knowledge and proficiency in the evolving domain of cloud security and by partnering with a cloud vendor, customers can take advantage of the organization's experience, resources, and assets [9].

2.1.3 Microsoft Azure

One of the modern and very influential companies of today, Microsoft, has employed their own solution, Microsoft Azure, to provide solutions over all services of the cloud: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [11]. At its core, Microsoft Azure is a centralized sandbox for creating all different types of resources that can work individually or together to provide companies with cloud-based solutions [11]. Microsoft Azure not only provides several advantages over on-premises solutions but is efficient, flexible, and very reliable [11].

Microsoft Azure specializes in disaster recovery and data backup [11]:

- Azure offers exceptional flexibility for data backup, supporting multiple languages, operating systems, and locations.
- Azure site recovery enhances data backup by providing offsite replication, extended data retention, and cost savings without significant capital investment.

Microsoft Azure provides an easy solution to modern day web and mobile applications [11]:

- Azure offers automatic patch management and continuous deployment support for streamlined app management.
- Microsoft AutoScale adjusts resource allocation based on web traffic, optimizing performance and cost efficiency.

In the field of quantum computing, Microsoft has created an advanced quantum cloud service known as Azure Quantum [12]. Within Azure Quantum, users have access to state-of-the-art cloud tools to help them utilize and refine quantum algorithms using accessible hardware and software [12]:

- Utilize the Azure Quantum resource estimator tool to project the required number of logical and physical qubits, runtime, and differences across qubit technologies for executing and refining quantum applications on future quantum computers.
- Combine classical and quantum computation to innovate hybrid algorithms like adaptive phase estimation.
- Azure Quantum features quantum Software Development Kits (SDKs) like Q#, Qiskit, and Cirq, supporting cross-platform deployment, direct access to Quantum Processing Units (QPUs) for native circuit execution, and a repository of high-quality samples and educational resources.

There are many more advantages to Microsoft Azure including its plethora of different tailor-made resources that provide the tools to construct any unique cloud solution. Provided below are some examples of these resources:

Virtual Machine: An Azure Virtual Machine has the capabilities to run virtually any operating system distribution with up to 416 vCPUs and 12 TB of memory, making them exceedingly customizable [13]. They also feature per-second billing allowing for only paying what computing power one uses, scaling which allows customers flexibility with the type of hardware they would like to utilize, and comprehensive data encryption features that meet regulatory requirements [13].

Storage Account: Azure Storage Accounts offer plenty of different options to store data within the cloud including blobs, tables, and queues [14]. This flexibility allows for auto scaling to fit customer needs, and the storage accounts can be seamlessly woven into different standing projects within Azure cloud as well as outside via REST API [14].

Function App: Azure Function Apps allow for various pieces of code to be deployed via a serverless architecture, reaping the benefits of no designated server infrastructure and configurations [15]. Function apps are designed to carry out smaller tasks that can work completely independent from your project via accessible endpoints [15].

2.2 Quantum Computing

Quantum computing is a rapidly emerging technology that is on track to revolutionize the way we process information. Unlike classical computers that rely on binary bits to store and process information, quantum computers use quantum bits or qubits, which can exist in multiple states simultaneously, allowing for more complex calculations and exceedingly faster processing times [16]. Quantum computing has the potential to solve complex problems in a wide range of fields, from materials science and cryptography to artificial intelligence and machine learning. As a result, many companies and research institutions as well as governments are investing heavily in quantum computing research and development, and the technology will inevitably have a colossal impact on many areas of science and industry [16].

2.2.1 Properties of Quantum Mechanics

Quantum computers rely on four basic properties found in the realm of quantum mechanics to provide the inherent ability to compute complex calculations:

Superposition: Superposition is a fundamental concept in quantum mechanics that describes the ability of a quantum system to exist in multiple states simultaneously [17]. This means that until a measurement is made, the system is not confined to a single state but rather exists in a combination of all possible states [17]. For instance, an unmeasured electron can exist in superposition, where it occupies multiple energy levels or positions simultaneously. The act of measurement causes the superposition to collapse into a single state, with the resulting state being one of the possible states that the system could be in [17].

Entanglement: Entanglement is a quantum property that enables the linkage of objects, specifically qubits, in a way that their properties become interconnected, regardless of distance [17]. By creating entanglement between qubits in a quantum computer, the system can process information in a fundamentally different way than classical computers, allowing it to examine multiple states at once [17]. This results in a significant computational advantage, enabling quantum computers to solve complex problems and find solutions much faster and more efficiently than classical computers [17].

Interference: The phenomenon of interference is a crucial tool for manipulating quantum states [17]. The two types of interference are called constructive interference which can enhance signals and destructive interference which can cancel out signals [17]. By exploiting interference, quantum systems can be designed to amplify certain states while suppressing others, which can improve the accuracy of measurements and increase the probability of obtaining a correct outcome [17].

Coherence/Decoherence: The performance of quantum computers can be significantly impacted by noise and environmental disturbances, making them extremely sensitive to external effects [17]. Moreover, the inherent quantum nature of the information makes it prone to decay over time towards the state of decoherence, limiting the number of operations that can be performed before the information is lost [17].

2.2.2 Quantum Algorithms

The inability to solve exceedingly complex calculations is a classical computer's most prominent shortcoming in today's world, but quantum computing revolutionizes this by proving impressive calculation times in comparison to classical computers [16]. According to an IBM internal analysis of quantum computing's potential for significant speed improvements over classical computers, a classical algorithm with exponential run time that takes roughly 3300 years would only take a quantum algorithm with polynomial time 11 minutes [16]. There have been many revolutionary quantum algorithms that have been discovered that have shaken the foundations of mathematics and physics. Two significant algorithms that have been discovered thus far are [18]:

Shor's Algorithm: Every integer contains a unique decomposition of itself into other smaller factors [18]. One very difficult problem that was born out of the factoring of large integers was the problem of finding the prime factors of the integer [18]. Suppose we would like to factor an arbitrary integer N with d decimal places [18]. The classical brute-force algorithm requires the need to search through all prime numbers $p_i \in \{p_1, p_2, ..., p_k\}$ up to $\sqrt{N}$ and check where p_i divides N [18]. At worst case, this classical brute-force algorithm would take roughly $\sqrt{N}$ which increases exponentially as d increases [18]. Immediately, the drawbacks of this classical algorithm can be made known which motivated researchers to seek out a better more efficient algorithm. In 1995, a mathematician by the name of Peter Shor proposed a polynomial-time quantum algorithm addressing this factoring problem [18]. The backbone of this algorithm relies on being able to solve another problem known as period finding: Given integers N and a, find the smallest positive integer r such that $a^r - 1$ is a

multiple of N [18]. The number r is referred to as the period of a modulo N. Provided below is an algorithmic representation of period finding [18]:

$$\begin{aligned} a^2 &= j_1 \pmod N \\ a^3 &= j_2 \pmod N \\ &\vdots \\ a^r &= 1 \pmod N, \end{aligned}$$

where $j_i \in \{j_1, j_2, ..., j_{r-2}\}$ is the result of carrying out the modular exponentiation [18]. Once the period r is found for an arbitrary number N, let us then use the identity [18]

$$a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1).$$

We can first observe that $a^{r/2} - 1$ is not a multiple of N since otherwise $r/2$ would then be the smallest period for N [18]. This then separates the problem into two cases [18]:

Case 1: $a^{r/2} + 1$ is not a multiple of N. So, neither integers $a^{r/2} \pm 1$ are factors of N, meaning p_1 is a prime factor of $a^{r/2} - 1$ and p_2 is a prime factor of $a^{r/2} + 1$ (or visa versa) [18]. Thus, the two prime factors p_1 and p_2 can be computed by finding the greatest common denominator of N and $a^{r/2} \pm 1$ denoted by $\gcd(N, a^{r/2} \pm 1)$ [18].

Case 2: $a^{r/2} + 1$ is a multiple of N. In this case, it is wise just to stop and find a different integer a since the occurrence of this case is not significantly common [18].

Shor's becomes advantageous when implemented on a quantum computer due to its efficient simulation of the period-finding machine [18]. The primary reason that Shor's algorithm does not work on classical computers is due to the computational complexity of calculating modular exponentiation in the period-finding problem [18]. In terms of performance, Shor's algorithm runs in *polynomial* time on a quantum computer compared to *exponential* time on a classical computer, providing a substantial speedup for being able to find prime factors for large integers [18].

Grover's Algorithm: Suppose you have a collection of N items stored in an unsorted database and you would like to find an item with some desired property. Let's label this desired item as w among the collection of items. Using a classical computer, one would need to go through on average $N/2$ of these items before finding w [18]. Furthermore, worst case one would need to go through all N to find w [18]. Fortunately, by using Grover's algorithm we can reduce the amount of time to find the desired item w to around $\sqrt{N}$ steps [18]. Grover's algorithm even applies to generic list structures which makes it very versatile [18]. The key idea that is used in Grover's algorithm is amplitude amplification [18]. Suppose our database is constructed of all possible basis states that the qubits can be in [18]. For example, suppose we have 2 qubits, so our list of possible computational basis states is comprised of:

$$|00\rangle, |01\rangle, |10\rangle, |11\rangle.$$

Since the chance of finding w is uniform compared to the other items, we can represent this as a quantum state of *uniform superposition* [18]:

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle.$$

Based on the properties of a quantum system being in superposition, observing a state of this quantum system would collapse it into one of the basis states, where each state has exactly $\frac{1}{N}$ chance of collapsing into [18]. So, on average we would need to collapse the quantum system $\frac{1}{N}$ times to find w which is an exact representation of our problem [18]. Let's now use amplitude amplification to enhance our chances of finding w. At a high level, amplitude amplification extends the amplitude of the probability of collapsing into w while concurrently shrinking the amplitudes of the other states [18]. Carrying out repeated operations on the quantum system using amplitude amplification will return w with exceedingly high confidence [18].

2.3 Cryptography

Cryptography is a field of study that deals with developing secure communication techniques that enable authorized parties to access and interpret sensitive information [19]. It involves using various algorithms and methods to encrypt data and messages to ensure confidentiality, integrity, and authenticity [19]. Cryptography plays a critical role in the digital age, where secure communication and online transactions have become so prevalent [19].

2.3.1 Pseudo-Random Number Generators

In order to achieve strong levels encryption within cryptography one must use robust sources of randomness. Randomness (entropy) is fundamental to cryptography because it is essential for generating keys which are the main ingredient for encrypting [20]. Higher randomness equates to a more secure cryptographic system [20]. Thus, the main challenge is creating true randomness. To create randomness, Random Number Generators (RNGs) are used:

Random Number Generators (RNGs): An RNG relies on a nondeterministic source, referred to as the entropy source, along with an entropy distillation process to produce randomness [1]. The distillation process is essential to address any weaknesses in the entropy source that might result in non-random sequences, such as long strings of zeros or ones [1]. Typically, the entropy source derives from a physical quantity like electrical circuit noise, user process timing (e.g., keystrokes or mouse movements), or quantum effects in semiconductors, often using various combinations of these inputs [1].
The outputs of an RNG can be used directly as random numbers or can be fed into a pseudorandom number generator (PRNG) [1]. To be used directly, the RNG output must satisfy strict randomness criteria, verified by statistical tests to ensure that the physical sources of the RNG inputs appear random [1]. For example, electronic noise might seem random but

could contain regular structures like waves or periodic phenomena that statistical tests would identify as non-random [1].

For cryptographic applications, RNG outputs must be unpredictable. However, some physical sources, such as date/time vectors, can be quite predictable. Combining outputs from different types of sources can mitigate this issue, but the resulting RNG outputs might still fail statistical tests for randomness [1]. Additionally, producing high-quality random numbers can be time-consuming, which is impractical when large quantities are needed [1]. In such cases, pseudorandom number generators may be more suitable.

Pseudo Random Number Generators (PRNGs): A pseudorandom number generator (PRNG) generates multiple "pseudorandom" numbers using one or more inputs known as seeds [1]. In applications requiring unpredictability, the seed must be random and unpredictable, typically sourced from a random number generator (RNG), making an RNG essential for a PRNG [1]. The PRNG's outputs are deterministic functions of the seed, meaning the only true randomness comes from the seed generation [1]. This deterministic nature is why the term "pseudorandom" is used. Since each value in a pseudorandom sequence can be recreated from its seed, saving the seed is sufficient for reproducing or verifying the sequence [1]. Ironically, pseudorandom numbers can appear more random than those from physical sources [1]. A well-constructed pseudorandom sequence transforms each value from the previous one, which seems to add extra randomness [1]. Multiple transformations can remove statistical auto-correlations between inputs and outputs, resulting in PRNG outputs with better statistical properties and faster production than those from an RNG [1].

2.3.2 Encryption Techniques

Some of the common encryption techniques used in cryptography include symmetric-key encryption and public-key encryption [19]:

Symmetric-Key Encryption: A type of encryption where a single key is used in correspondence with both the encryption and decryption of sensitive plaintext [19]. This type of encryption is widely used in major industries such as aerospace, business, and health care. It is important to note that this type of encryption is not limited to sharing sensitive information between just two parties [19]. Any person who has possession of the key is able to access the sensitive information which therein lies a problem of data access by unauthorized parties [19]. Symmetric-key encryption is categorized into two types of ciphers: block ciphers and stream ciphers.

Block Ciphers: Block ciphers are known for converting arbitrary plaintext into fixed-sized blocks of data, usually either 64 bits or 128 bits (8 bytes or 16 bytes respectively) in length [21]. Some modes of block ciphers also require an initialization vector (IV) for encryption [21]. An IV is a pseudo-random or random sequence of characters used to encrypt the first block of plaintext which then results in a unique initialization ciphertext block for the subsequent blocks [21].

Block ciphers are based on a type of mathematical construct known as Pseudo Random

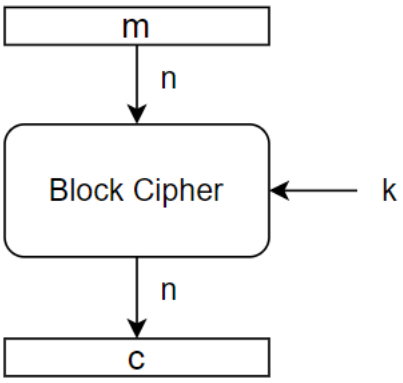

Figure 2.1: Block diagram representing a block cipher [2]

Permutations (PRPs) [2]. PRPs are invertible functions that accept an n-bit input m along with a secret key k, and produce an n-bit output c [2]. A PRP is secure if it is indistinguishable from a random bijective function mapping n-bit inputs to n-bit outputs [2]. Block ciphers use secure PRPs to:

1. Encrypt a block of plaintext into a block of ciphertext using a secret key [2]
2. Decrypt the ciphertext block back into the original plaintext block [2]

Block cipher operations ensure that the plaintext is thoroughly mixed and dispersed into the ciphertext [2]. This means that even small changes in the plaintext result in drastically different ciphertexts [2]. Figure 2.1 shows the high-level architecture of a typical block cipher. In block cipher encryption, each plaintext block corresponds to a ciphertext block [2]. As a result, changing one bit in the plaintext can alter the entire ciphertext block, demonstrating the cipher's sensitivity to changes in the plaintext [2].

Below are some common block cipher modes [21]:

- **Cipher Block Chaining (CBC)** - In this mode, plaintext is broken up into blocks of data where the first block is initially encrypted with the IV via bitwise exclusive-OR and then encrypted again with the key [21]. Thereafter, each subsequent block is encrypted by the previous block's resulting ciphertext and then encrypted again with the key [21].
- **Counter Mode (CTR)** - In this mode, plaintext is broken up into blocks of data where the blocks are encrypted separately from one another using a counter as the initialization vector. After every subsequent block encryption the counter is incremented [22]. One primary advantage over many other block ciphers is that counter mode supports parallel computing [22].

One advantage that block ciphers provide over stream ciphers is that, in general, there is a high diffusion rate since most modes incorporate previous ciphertext blocks in the encryption of subsequent plaintext blocks [21]. However, this high diffusion rate also

correlates to a high error propagation rate which implies that a minor change to the plaintext will significantly change the subsequent ciphertext blocks [21].

Stream Ciphers: Stream ciphers are known for encrypting a continuous stream of bits of data using keystreams [21]. A keystream is essentially a combination of a key and a nonce (number used only once) to form a pseudo-random number which is then used to encrypt the plaintext by bitwise exclusive-OR [21]. Stream ciphers are grounded in

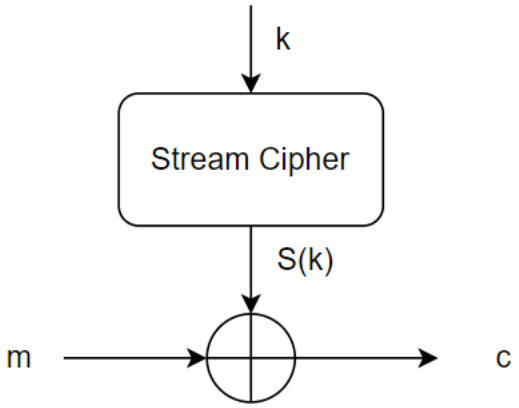

Figure 2.2: Block diagram representing a stream cipher [2]

a theoretical cipher known as the One Time Pad (OTP) [2]. In an OTP, the secret key must be the same length as the message m [2]. The encryption function is defined as $E(k,m) = m \oplus k$, and the decryption function is $D(k,c) = c \oplus k$ [2]. Typically, both plaintext and ciphertext are processed as bitstreams [2]. The key k is referred to as the keystream, and it is combined bit by bit with the plaintext to generate the ciphertext stream [2].

If the keystream is perfectly random (i.e., follows a uniform distribution), the OTP achieves perfect secrecy [2]. This means the resulting ciphertext is indistinguishable from a random sequence, due to the properties of the XOR operation [2] [2]. Consequently, an attacker intercepting the ciphertext cannot gain any information about either the message or the key [2]. However, the OTP is impractical because the sender and receiver must share a secret key that is as long as the message [2].

Stream ciphers implement the OTP concept. In stream ciphers, the keystream is generated by a Pseudo-Random Generator (PRG) [2]. The PRG takes a value k (the seed of the stream cipher) as input and outputs the keystream $S(k)$ [2]. Thus, the encryption and decryption functions are defined as $E(k,m) = m \oplus S(k)$ and $D(k,c) = c \oplus S(k)$ [2]. Therefore, the secret key exchanged between the sender and receiver is the seed of the PRG, which is much shorter than the resulting keystream [2]. As long as the PRG produces an unpredictable keystream, the stream cipher remains secure [2].

In Figure 2.2, the high-level architecture of a typical stream cipher is depicted. It is

important to note that in stream cipher encryption, there is always a bit-to-bit correspondence between plaintext and ciphertext. Specifically, a one-bit difference in the plaintext results in the same one-bit difference in the ciphertext [2].

Stream ciphers can be categorized into synchronous and asynchronous stream ciphers [21]:

- **Synchronous** - These types of stream ciphers generate keystream blocks that are independent from past plaintext and ciphertext blocks [21].
- **Asynchronous** - These types of stream ciphers generate keystream blocks as a function of the symmetric key as well as a fixed size of the previous ciphertext block [21].

Below is a common stream cipher mode [21]:

- **Salsa20** - Salsa20 is a lightweight efficient cipher that uses an expansion function to create a keystream [21]. The keystream is created via a key, nonce, and constant vectors using various add-rotate-XOR (ARX) operations [21].

Public-Key Encryption: Also known as asymmetric encryption, public-key encryption uses two mathematically-linked keys known as a public-private key pair in order to encrypt and decrypt sensitive information [19]. In order to send sensitive information one must encrypt the data with their destination's public key and then the destination may decrypt that data with their own private key [19]. This process ensures that the parties' respective private keys are never to be shared with any unauthorized parties [19].

RSA Asymmetric Encryption System: The RSA Asymmetric Encryption System forms the basis for most public-key encryption schemes [23]. RSA was first introduced in 1977 by its founders Rivest, Shamir, and Adleman, hence the name [23]. Suppose we select two large prime numbers x and y [23]. We then calculate n as a product of x and y denoted as $n = x \cdot y$ [23]. We then compute Euler's *totient* function $\phi(n) = (x-1)(y-1)$ [23]. We then select an integer e that is relatively co-prime to $\phi(n)$ where $1 < e < \phi(n)$ [23]. The pair (n, e) now makes up the public key which can be considered as a physical lock to encrypt your sensitive data [23]. To calculate the private key we use the public key and the extended euclidean algorithm to find d such that $d = 1 \pmod{\phi(n)}$ [23]. The pair (n, d) then make up the private key [23].

2.3.3 Message Authentication Codes

The main goal of data encryption within cryptography is to ensure that sensitive data remains confidential. However, there is also another side of cryptography that handles the integrity of sensitive data. In order to ensure that sensitive data does not get tampered with in any way by an adversary, Message Authentication Codes (MACs) are used [24]. MACs can be considered as a digital signature for data since they tell you who created or sent the data and whether the data has been tampered with [24]. A MAC is usually in the form of a tag or small string of bits that are concatenated to the message. Two important properties that MACs provide are [24]:

- **Completeness** - When a message is sent to a receiver, the receiver can use the tag to then verify the message and confirm the message integrity [24]
- **Security** - Tags are derived from the message before it has been sent, which means that a receiver can determine if the message has been altered by verifying the message with the tag [24]

Below are some common variations of MACs:

- **Hash-Based MAC (HMAC)** - Hash-Based MAC essentially uses a cryptographic hashing algorithm such as MD5 or SHA-256 to hash the data into a fixed bit size tag [25]. This tag along with the file data is sent over the server to the end user, who then hashes the retrieved file themselves and compares their generated hash to the shared, server generated hash value [25]. Furthermore, HMAC also employs the authenticity check of the message by providing exchanging parties a way to verify that the message is actually from their interacting parties [25]. HMAC implements this authenticity check by carrying out a preliminary process of key exchanging where both parties exchange a shared secret key [25]. HMAC is excellent for ensuring file transfer data integrity due to its efficiency with handling large files [25].
- **Cipher-Based Mac (CMAC)** - Cipher-Based Mac creates message authentication tags by using a block cipher and a private key similar to Cipher Block Chaining (CBC) encryption [26]. However, instead of using a standardized hashing function like in HMAC, CMAC uses a block symmetric key method [26]. CMAC would be more beneficial to use when there is embedded hardware that offers hardware acceleration [26]. Generally however, HMAC provides a faster authentication method over CMAC [26].
- **Parallelizable Mac (PMAC)** - Many other conventional MAC algorithms are sequential, meaning that in order to move forward in the authentication algorithm one must encrypt the previous message blocks [27]. PMAC has a similar block cipher structure to CMAC but provides a parallelizable solution to this problem by allowing a MAC to be calculated independently for each message block [27]. An immediate benefit to this is a decrease in bottleneck for network speeds which increases the speed of cryptographic hardware [27].

2.3.4 Authenticated Encryption

In order to provide a complete spectrum of protection over sensitive user data, different ciphers can be used in part with different MACs. Using both ensures that sensitive user data is protected both in terms of confidentiality and integrity to maximize data protection which is otherwise known as authenticated encryption [28]. A select few of these authenticated encryption protocols have been standardized by the National Institute of Standards and Technology (NIST) [28]:

- **Galois/Counter Mode (AES-GCM)** - This mode of operation uses AES-CTR for the encryption and then uses CW-MAC for the authentication [29]. This system requires the use of a nonce, or initialization vector, which is unique to every instance of the protocol, a key for the encryption operation, and associated data that will be authenticated but not encrypted [29].

- **Counter with Cipher Block Chaining Message Authentication Code (AES-CCM)** - This mode of operation uses CBC-MAC for the authentication and then uses AES-CTR for the encryption [28]. This system has similar requirements to AES-GCM in that it requires the use of a nonce, or initialization vector, which is unique to every instance of the protocol, a key for the encryption operation, and associated data that will be authenticated but not encrypted [28].

2.4 Cloud Security

The use of cryptographic techniques in cloud security refers to the implementation of methods to safeguard the privacy and security of data stored and processed in the cloud. Cryptography offers several tools to ensure the confidentiality, integrity, and authenticity of data, all of which are essential requirements for cloud security [30]. Encryption is a popular cryptographic technique used to protect data both at rest and in transit, making it unreadable to unauthorized parties. Furthermore, cryptographic protocols, such as secure key exchange mechanisms and digital signatures, can be used to authenticate users and secure cloud-based transactions. However, implementing cryptography in cloud security requires careful consideration of factors such as key management, protection, and distribution [30].

2.4.1 Cloud Security Challenges

Provided below are some challenges that cryptography faces within the cloud:

- **Secure Key Management** - Since cloud-based transactions require encryption and decryption methods to protect sensitive user data, the keys that are used in these methods require protection from adversaries in some way [31]. CSPs provide this key protection by implementing key management protocols [31]. Some of these cloud key management protocols include hardware security modules and other tools which are necessary to meet compliance requirements from different reference standards such as the United States National Institute of Standards and Technology (NIST) [31].

- **Multitenancy** - CSPs provide their services to many different customers using computing resources and power [32]. Multitenancy is the idea that the customers of the CSP all use the same computing resources and power from the same source, although the data remains completely separate to sustain data confidentiality and integrity [32]. Multitenancy also helps CSPs conform to many compliance requirements for cryptography that require CSPs to ensure user data is handled in a conformed and uniform way [32]. Multitenancy also allows better use of resources since cloud computing servers or machines can be shared amongst users which in turn lowers the user costs [32].

- **Scalability** - The cloud is an ever-fluctuating network of storage and computing resources, and in order for the cloud to be an integral role in customers and organizations it must be able to scale to meet the ever-increasing demands of computing power, memory, and

communication speeds [33]. With these demands calls for the continuing preservation of data confidentiality and integrity [33].

2.4.2 Cloud Computing Resources

Among the cloud, there are many cloud computing resources that can function as random number generators (RNGs) and are publicly available for use. These resources may be used for cryptographic applications as long as they produce randomness deemed secure by statistical testing suites. Provided below are two modern cloud computing resources that can be used as RNGs:

1. **Sparse Simulator**: High-performance simulation methods for quantum programs on classical hardware often use large vectors to represent quantum states, but these states are frequently sparse due to algorithmic structures [34]. A novel simulation technique was introduced to exploit this sparsity, reducing memory usage and runtime [34]. Optimizations such as gate rescheduling were also implemented to minimize data structure accesses [34]. The technique was benchmarked with quantum algorithms for factoring, integer and elliptic curve discrete logarithms, and chemistry applications [34]. The Sparse Simulator can be utilized through Microsoft Azure Quantum by using their open-source programming language called Q# within their Azure Quantum Development Kit [35]. Q# can be used to deploy code solutions as complex as recreating quantum algorithms or as simple as creating an RNG.

2. **Wolfram Cloud**: The Wolfram Cloud integrates a cutting-edge notebook interface with the Wolfram Language, allowing for scalable programming and immediate access to a wide array of built-in algorithms and knowledge [36]. It enables seamless code and content deployment, offering instant APIs, mobile apps, and interactive document embedding [36]. Utilizing the extensive Wolfram Knowledgebase, the Wolfram Cloud ensures continuous data updates and robust automation, along with flexible storage solutions and programmable permissions [36]. Wolfram Language can be used to deploy code that utilizes built-in algorithms to construct RNGs, taking advantage of the computing power Wolfram Cloud has to offer.

Despite these challenges, cryptography remains a crucial tool for cloud security, providing a means for organizations to safeguard their data against unauthorized access through what the tools of cloud computing resources have to offer.

2.5 Quantum Cryptography

Quantum cryptography is a sub-field of cryptography that aims to create secure communication channels that are impossible to eavesdrop on. Unlike traditional cryptographic systems that rely on mathematical algorithms, quantum cryptography uses the principles of quantum mechanics to protect data [37]. The central idea behind quantum cryptography is that a quantum system cannot be observed without disturbing it, making it impossible for an eavesdropper to intercept messages without leaving a trace. For instance, a popular technique used in quantum cryptography

is Quantum Key Distribution (QKD), which involves the exchange of quantum states between two parties to establish a secret key. Since any attempt to observe the quantum states by a third party would disrupt the transmission, the two parties can detect the presence of an eavesdropper and abort the key exchange [37]. Another example of utilizing quantum mechanics is what are known as quantum random number generators (QRNGs). QRNGs play a vital role in generating random numbers by exploiting properties of subatomic particles such as electrons and photons [3].

2.5.1 Quantum Random Number Generators

In order to generate random numbers, different device constructions are developed such as the ones provided below [3]:

- **Radioactive Decay QRNGs** - Radioactive decay-based quantum random number generators utilize the inherent randomness of radioactive decay processes to generate random numbers [3]. These QRNGs rely on the unpredictability of the timing or energy levels of radioactive decay events to produce truly random outcomes [3]. The detection of radioactive decay events by devices such as Geiger counters serves as a source of entropy, which is converted into random numbers through appropriate measurement and processing techniques [3].

- **Noise-based QRNGs** - Noise-based quantum random number generators leverage the intrinsic randomness present in electronic or quantum noise phenomena present in electrical circuits to generate random numbers [3]. Specifically, these QRNGs exploit the unpredictable fluctuations or variations in electrical signals such as shot noise or vacuum fluctuations [3]. The randomness is captured through specialized sensors or circuits that amplify and extract the noise signals [3].

- **Optical QRNGs** - Optical quantum random number generators utilize the principles of quantum optics to generate random numbers such as the nature of light or the uncertainty principle, to produce unpredictable outcomes [3]. Optical QRNGs typically involve the detection of single photons or the measurement of quantum properties of light, such as its polarization or phase [3]. Furthermore, some QRNGs utilize the same unpredictable emittance of light similar to how radioactive decay QRNGs operate [3].

2.5.2 Post-Processing

QRNGs are very often used to construct a random number generator protocol which involves more than just the generators themselves [3]. In Figure 3.1, a simple block diagram is shown visually representing a quantum random number generator with post-processing. Post-processing is done on the outputted numbers to first extract more randomness out of them and then validate that they are in fact random enough to meet regulatory standards, such as the National Institute of Standards and Technology (NIST) standards for random numbers [3]. Post-processing can be broken down into two key components [3] [38]:

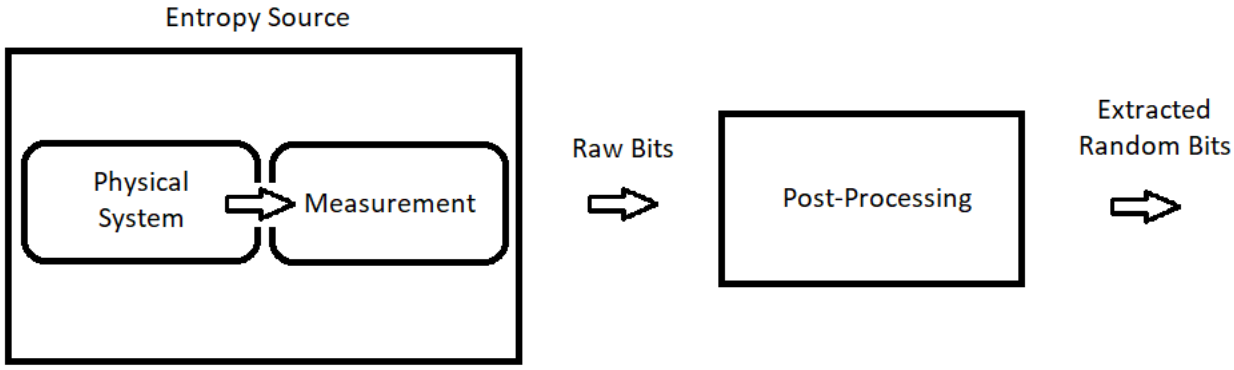

Figure 2.3: Block diagram representing a quantum random number generator [3]

- **Randomness Extractors** - Randomness extractors are algorithms or techniques designed to distill true randomness from potentially imperfect or biased sources of data, in this case QRNGs [3]. These extractors aim to eliminate any biases or correlations in the input data and produce high-quality random numbers by applying mathematical and statistical operations to transform the input data into a more uniform and unbiased distribution [3].

- **Verification Testing** - Statistical testing for random numbers involves examining the characteristics of the data to determine its level of randomness [38]. Some examples of these characteristics are the number of ones and zeros or patterns in m-bit blocks of the data [38]. They are organized into test suites that provide more comprehensive randomness analysis such as the NIST Statistical Test Suite, an efficient implementation of these tests that provides fast throughput of testing random numbers [38].

2.5.3 Statistical Testing Suites

Numerous testing suites have been developed before and after NIST released its standardized testing suite, and most continue to challenge the benchmark of how rigorous the statistical testing should be. Quite a few are very popular today, and they are:

- **ENT statistical testing suite**: The ENT statistical test suite is a small testing suite comprised of 6 statistical tests which are entropy, ideal compression rate, Chi-square test, arithmetic mean, Monte Carlo estimation of π and serial correlation [39].

- **PractRand statistical testing suite**: PractRand includes a comprehensive battery of tests to detect bias in RNGs efficiently, outperforming other test suites in detecting biases quickly [40]. It offers flexible testing options, including command line tools for easy integration and multithreaded capabilities for higher performance [40]. PractRand supports very long sequence lengths, features original tests, and allows for preliminary result evaluations at any time during testing. However, it requires more random bits than most test suites, which may not be suitable for very slow PRNGs [40].

- **TestU01 statistical testing suite**: TestU01 is an ANSI C software library providing a suite of tools for the empirical statistical testing of uniform random number generators (RNGs) [41]. The library features predefined test suites for uniform random numbers and bit sequences and tools for studying the interaction between tests and the point sets from various RNG families, helping determine necessary sample sizes before an RNG fails [41]. TestU01 also offers various generic and specific RNGs, both from literature and widely used software [41]. Tests can be applied to predefined generators, user-defined generators, or random number streams from any device or file [41]. Additionally, the article surveys and classifies statistical tests for RNGs and applies these test batteries to many commonly used RNGs [41].
- **Diehard(er) statistical testing suite**: Dieharder is a tool designed for timing and testing both software and hardware random number generators (RNGs) for research and cryptographic purposes [42]. Unlike its predecessor Diehard, which uses file-based sources of random numbers limited to about ten million, Dieharder supports testing of generators that can produce unbounded streams of random numbers, essential for modern applications needing 10^{18} or more random numbers [42]. Dieharder supports three types of file-based input, including piping bit streams and direct file input of binary or ASCII formatted numbers, and can handle large files [42]. Additionally, Dieharder is extensible, incorporating all Diehard tests and eventually all NIST STS tests, aiming to include various user-contributed tests and those from the literature, making it a comprehensive tool for RNG testing and validation [42].

Various statistical tests can be used to evaluate and compare a sequence to a truly random one. Randomness is characterized by probabilistic properties, which can be described and predicted using probability [1]. The outcomes of these tests on a truly random sequence are known in advance and can be expressed in probabilistic terms [1]. There are countless possible statistical tests, each designed to detect patterns that would suggest the sequence is not random. Because there are so many tests, no single set of tests is considered exhaustive. In fact, there will never be a complete list of statistical tests, which implies that *completeness* will never be achieved [1]. Additionally, the results of these tests must be interpreted carefully to avoid incorrect conclusions about a generator [1].

A statistical test is designed to evaluate a specific null hypothesis (H_0). In this context, the null hypothesis is that the sequence being tested is actually random [1]. The alternative hypothesis (H_a) is that the sequence is not random [1]. Each test results in a decision to accept or reject the null hypothesis, determining whether the generator is producing random values based on the tested sequence [1]. Each test uses a relevant randomness statistic to decide on the null hypothesis. Assuming the sequence is random, this statistic follows a certain distribution. A theoretical reference distribution of this statistic under the null hypothesis is determined mathematically, from which a critical value is derived (typically far out in the tails of the graphed distribution, such as at the 99% point) [1]. During the test, a test statistic value is calculated for the sequence, and if this value exceeds the critical value, the null hypothesis of randomness is rejected (otherwise, it is accepted) [1].

Statistical hypothesis testing works because the reference distribution and critical value are generated under the assumption of randomness [1]. If the data is truly random, the test statistic

value should have a very low probability (around perhaps 0.01%) of exceeding the critical value [1]. If the test statistic value does exceed the critical value, it suggests that the assumption of randomness is incorrect [1]. Therefore, exceeding the critical value leads to rejecting H_0 (randomness) and accepting H_a (non-randomness).

Statistical hypothesis testing produces one of two conclusions: either accept H_0 (the data is random) or accept H_a (the data is non-random) [1]. The following table relates the true but unknown status of the data to the conclusion derived from the testing procedure.

Null Hypothesis (H_0) Is:	True	False
Rejected	Type I Error	No Error
Not Rejected	No Error	Type II Error

Table 2.1: Type I and Type II Errors [1]

If the data is genuinely random, rejecting the null hypothesis (concluding that the data is non-random) will happen a small percentage of the time, known as a Type I error [1]. On the other hand, if the data is truly non-random, accepting the null hypothesis (concluding that the data is random) results in a Type II error [1]. Correct conclusions occur when H_0 is accepted if the data is random and when H_0 is rejected if the data is non-random (resulting in a "No Error" conclusion) [1].

The probability of a Type I error is often referred to as the level of significance of the test and is denoted as α. This is the probability that the test will indicate a sequence is not random when it actually is [1]. In cryptography, common values for α are around 0.01, meaning there is a 1% chance of concluding non-randomness for a truly random sequence [1].

The probability of a Type II error is denoted as β. This is the probability that the test will indicate a sequence is random when it is not, meaning a "bad" generator produced a sequence that seems random [1]. Unlike α, β is not fixed and can vary because there are many ways a data stream can be non-random, each yielding a different β [1]. Calculating β is more complex than calculating α due to these many possible forms of non-randomness [1].

A key goal of the tests is to minimize the probability of a Type II error, for example, to reduce the chance of accepting a sequence as random when the generator is actually faulty [1]. The probabilities α, β, and the size n of the tested sequence are interrelated such that specifying any two determines the third [1]. Typically, one selects a sample size n and a value for α (the probability of a Type I error) [1]. They then choose a critical point for the statistic that minimizes β (the probability of a Type II error) [1]. This involves selecting an appropriate sample size and an acceptable level of significance to ensure the smallest possible probability of incorrectly accepting a sequence as random [1].

Each test relies on a calculated test statistic value, which is derived from the data [1]. If the test statistic value is S and the critical value is t, then the probability of a Type I error is expressed as $P(S > t \mid\mid H_0 \text{ is true}) = P(\text{reject } H_0 \mid H_0 \text{ is true})$, and the probability of a Type II error is expressed as $P(S \leq t \mid\mid H_0 \text{ is false}) = P(\text{accept } H_0 \mid H_0 \text{ is false})$ [1]. The test statistic is used to calculate a P-value, summarizing the strength of the evidence against the null hypothesis [1]. For these tests, each P-value represents the probability that a perfect random number generator would produce a

sequence less random than the one tested, given the type of non-randomness the test assesses [1].

A P-value of 1 indicates perfect randomness, while a P-value of 0 indicates complete non-randomness [1]. A significance level (α) can be chosen for the tests. If P-value $\geq \alpha$, the null hypothesis is accepted, suggesting the sequence appears random [1]. If P-value $< \alpha$, the null hypothesis is rejected, suggesting the sequence appears non-random [1]. The parameter α denotes the probability of a Type I error, typically chosen in the range [0.001, 0.01] [1].

Provided below are two eaxmples where a P-value is chosen, and the resulting conclusion can be made:

- Example 1: $\alpha = 0.01$: Implies that 1 out of every 100 sequences would be rejected if it were random. A P-value of 0.01 or higher indicates that the sequence is random with 99% confidence, while a P-value below 0.01 indicates non-randomness with 99% confidence [1].
- Example 2: $\alpha = 0.001$: Implies that out of 1000 random sequences, we would expect one sequence to be rejected by the test. If the P-value is greater than or equal to 0.001, the sequence is deemed random with 99.9% confidence. Conversely, if the P-value is less than 0.001, the sequence is considered non-random with the same level of confidence [1].

In the research conducted within this paper, a P-value of 0.01 is used, which is also the same P-value used within NIST's publication of their standardized testing suite. Overall, Quantum Cryptography holds great promise for the future of secure communication.

2.6 Cloud Computing with Quantum Cryptography

Cryptography, cloud computing, and quantum mechanics all come together in the realm of cloud computing with quantum cryptography. Quantum cryptography leverages the principles of quantum mechanics to create secure communication channels, which are essential for cloud computing, where sensitive data is stored and processed. With the emergence of quantum computers, traditional cryptographic systems are becoming increasingly vulnerable to attacks, and quantum cryptography offers a promising solution for ensuring the security of cloud-based applications. As cloud computing and quantum cryptography continue to evolve, the future of secure cloud-based systems will likely depend on a combination of traditional cryptographic techniques and quantum-enabled security measures.

2.7 Related Work

Currently, there is developing research into the new paradigm of statistical testing of random number generators, both pseudo and quantum alike. This developing research is challenging the adhesiveness of RNGs to the ever-changing benchmark of cryptographically secure randomness by exploring broader statistical testing suites as well as their own protocol implementations. Several of these research papers within this realm will be unpacked in order to build upon existing knowledge and provide additional insight in this area.

2.7.1 Cloud-based Quantum Random Number Generator System

With the birth of quantum computing comes both benefits and drawbacks in the realm of cloud computing, cloud storage, and data security. Specifically, quantum computing has provided new ways to compromise classical Psuedorandom Number Generators (PRNGs) [43]. PRNGs are used at a great extent to output seemingly random numbers which are then used in cryptosystems to hide sensitive information. Quantum computing has opened up new security loopholes and algorithm attacks on these classical generators which presences a huge concern, especially when these classical cryptosystems are used by companies who require the utmost security for their customers' data [43]. Quantum Random Generators (QRNGs) have become increasingly sought after since they provide an excellent solution to the security loopholes. Alibaba Cloud servers currently use these QRNGs to produce random numbers by leveraging the properties of quantum mechanics. There are four unique QRNGs that Alibaba Cloud servers use [43]:

- single-photon detection
- photon-counting detection
- phase-fluctuations
- vacuum-fluctuations

At a high level, these four methods involve observing, counting, or detecting photons of light which act in a random manner based on the principles of quantum mechanics. The outputs of the QRNGs are based on the measurements of the photons and are then sent to customers requiring high security which use the random numbers for their cryptosystems [43]. In some cases where the client requires exceedingly high security, the numbers generated from each QRNG will be combined by bitwise exclusive-OR to ensure a random number is outputted with high confidence. Regarding the methods of utilizing QRNGs, there are currently no universal QRNG standards or verification methods that have been enforced which makes the evaluation of the performance of QRNGs difficult [43].

Each random number generated from the QRNGs follow a QRNG Platform Protocol [43]:

1. Data Import - Standard interfaces are used for the importing of data from the QRNGs to the end-users along with different data formats available.
2. Randomness Extraction from Entropy Source - Random numbers pass a randomness extractor which enhances the randomness per bit of data.
3. Bitwise XOR (Optional) - Outputs are combined by bitwise exclusive-OR.
4. Randomness Testing - The system performs real-time entropy tests such as National Institute of Standards and Technology (NIST) Randomness Tests on the random numbers to verify quality of randomness.

5. Identity Authentication for Dissemination - System authenticates end-users who request random numbers using pre-shared keys (PSKs).

6. Data Download of Random Numbers - Customers download the random numbers using various encryption protocols.

QRNGs will play an increasingly important role in the realm of cryptography since random numbers are a vital ingredient in a vast majority of encryption algorithms. There are immediate applications of QRNGs including the distribution of quantum random numbers known as QKD as well as post-quantum algorithms which will help strengthen encryption techniques [43].

2.7.2 Comparative Study of Quantum Random Number Generators

Random number generation is a critical component in many areas of information processing, including cryptography, mathematical modeling, Monte Carlo methods, and gambling [44]. The quality of randomness and the efficiency of the generation process are essential for these applications. While software-produced pseudorandom sequences are fast, they often fall short of the required randomness quality, requiring the development of hardware-based methods to generate truly random numbers [44].

Quantum random number generators (QRNGs) utilize the inherent unpredictability of quantum mechanical phenomena to produce random numbers [44]. These devices draw from physical sources such as quantum fluctuations, ensuring a high degree of randomness [44]. For instance, the PQ4000KSI by ComScire and the JUR01 developed at Wroclaw University of Science and Technology are two such QRNGs that have undergone rigorous testing [44].

A notable advancement in this field is the JUR02, a new miniaturized QRNG prototype that achieves a bit generation rate of 1 Mb/s [44]. This device has successfully passed standard randomness tests, highlighting its potential for practical applications [44]. Additionally, research in this area includes exploring entanglement-based QRNG protocols, which offer unconditionally secure public randomness verification [44].

Evaluating the quality of randomness involves statistical tests designed to detect any deviations from true randomness [44]. Common tests include the NIST and Dieharder suites, which assess various properties of the bit sequences generated by QRNGs [44]. The tests ensure that the sequences meet stringent standards required for secure and reliable applications [44]. Within this research, the PQ4000KSI and the JUR01 were put to the test using NIST's standardized statistical testing suite. The testing results were then compared to a pseudo-random number generator by Wolfram and shown to be very similar in terms of the randomness metrics [44].

These developments in QRNGs address the limitations of pseudo-random number generators by providing truly random sequences that enhance the security and reliability of various information processing tasks [44]. This research also emphasizes the critical distinction between quantum random number generators (QRNGs) and classical pseudo-random generators. QRNGs rely on quantum phenomena like quantum measurements and Fermi transitions, which provide unconditional randomness [44]. Despite inherent classical components introducing bias, methods like the von Neumann algorithm are employed to mitigate this bias effectively [44]. Current randomness

tests are noted for their limitations in distinguishing true randomness from pseudo-randomness, and the statistical testing results show this lack of distinction [44]. To address this, the research proposes a novel QRNG algorithm using multi-qubit entanglement for non-destructive public randomness testing. This approach allows for high-accuracy testing without compromising the confidentiality of the random bit sequence, thereby enhancing the efficiency and security of randomness testing in cryptographic applications [44].

2.7.3 Enhancing Random Number Generator Verification

A Statistical Testing Environment (STE) designed to rigorously evaluate the statistical properties of random number generators (RNGs) was proposed within this research [4]. This environment offers flexibility, allowing for both lightweight assessments and intensive testing that exceeds standard certification requirements [4]. The STE was applied to evaluate three RNGs: a 32-bit Linear Feedback Shift Register (LFSR) pseudo-RNG, Intel's RDSEED hardware RNG based on thermal noise, and IDQuantique's 'Quantis' quantum RNG utilizing photon detection [4].

Key findings from the research include:

- Shortcomings were identified in two RNGs under intense statistical testing, confirming and expanding upon previous research findings [4].
- By implementing a variety of post-processing techniques including deterministic, seeded, two-source, and physical extractors the research successfully improved the statistical quality of RNG outputs. This approach extends beyond previous studies that focused solely on deterministic or seeded extraction methods [4].
- The analysis consistently demonstrated that higher levels of extraction (level 2 and above) effectively produced outputs statistically indistinguishable from uniform randomness [4].

However, the study also highlights the inherent limitations of relying solely on statistical tests to ensure RNG integrity, particularly in cryptographic applications [4]. Ongoing challenges within existing statistical test suites, such as those of NIST and Dieharder, were brought to light, revealing issues such as test interdependencies and inaccurate testing settings that can affect the assessment of RNG quality [4].

Through this comprehensive approach, the research aims to contribute to the advancement and validation of RNG technologies, emphasizing the importance of complementary methods alongside statistical testing to ensure robust randomness in critical applications, especially in cryptography [4].

Chapter 3

Overview

This thesis strives to answer the main question:

> *How can statistical testing for modern PRNGs and QRNGs be enhanced beyond current standards in a post-quantum era?*

In the context of this research question, we aim to find a consolidated solution to achieve great efficiency and usability, while sustaining robustness and reliability in cryptographic applications. In order to arrive at the solution and answer the question, we identify three distinct aims that need completed:

3.1 Aim 1: Select Statistical Testing Suites and Random Number Generators

In order to explore statistical testing beyond what is currently standardized, we will select reputable testing suites as well as the standardized NIST Statistical Testing Suite (STS). To analyze the impacts of broader statistical testing coverage, we will select three reputable RNGs for use in the statistical testing as well as comparative statistical analysis. Two important questions that we answer in the completion of this aim are:

1. Which statistical testing suites are well suited for rigorous statistical testing purposes?
2. Which three contemporary RNGs will elicit the best comparative analysis on existing statistical testing research?

3.2 Aim 2: Develop Consolidated Statistical Testing Tool

Once the statistical testing suites and RNGs have been selected and the random binary data has been generated, we will develop a consolidated statistical testing tool in the form of a Command Line Interface (CLI). This tool will be used to run the statistical testing analysis on the three sets of ten randomly generated data samples. Three important questions that we answer in the completion of this aim are:

1. How can we develop a tool that can be easily integrated into any statistical testing environments?

2. In what ways can we make this tool very user-friendly to increase ease-of-use?

3. What input should a user have to specify in order to test randomly generated data?

Even though these are the three main questions we strove to answer in this aim, there are many more intricate questions we answered when making architectural decisions in the development of this tool.

3.3 Aim 3: Generate Data and Statistical Testing Results

The final aim requires that the random data is generated and statistical testing is completed on each of the ten samples in relation to each of the three RNGs. Once the CLI tool is complete, the statistical testing results will be collected from each of the testing suites we run on the data. Three main questions we answer in the completion of this aim are:

1. In what format should we generate the random data in?

2. What should the generated size of each sample be?

3. How do we consistently generate and test each sample of data?

Similar to Aim 2, we also have many other specific questions we answer in regards to the statistical testing methodology of the samples that we go over in detail in later chapters.

Chapter 4

Selection of Statistical Testing Suites and Random Number Generators

In this chapter we will elaborate on the first aim of this thesis - focusing on the selection of the statistical testing suites and the random number generators. In completing this task, we selected eight different statistical testing suites and three random number generators.

4.1 Statistical Testing Suite Selection

Each statistical testing suite was sourced from their respective originating website or code repository. Specifically, a detailed description, and link to the originating website or code repository will be provided for each of these testing suites. It is important to note that each statistical testing suite is unique in its own regard, but some tests within each suite may overlap. Furthermore, each testing suite has its own recommendation on its usage specifying how long a sample should be as well as how to interpret the results. These selections are also based on Quantinuum's implementation of their Statistical Testing Environment [4], and this research builds on these selections as well.

In this aim we strove to answer the first question:

Which statistical testing suites are well suited for rigorous statistical testing purposes?

The eight selected statistical testing suites are as follows:

- *ENT*
- *PractRand*
- *SmallCrush*
- *Crush*
- *Alphabit*
- *Rabbit*
- *Dieharder*

- *NIST STS*

It is also important to note that the *SmallCrush*, *Crush*, *Alphabit*, and *Rabbit* testing suites all originate from one testing suite package known as *TestU01*. Below, we will break down each statistical testing suite:

4.1.1 ENT

ENT[1] is referred to as a pseudo-random number sequence test [45], however with recent advances in QRNGs, ENT has been an effective and efficient test in the statistical testing realm. ENT performs exactly six statistical tests on a randomly generated stream of either 8-bit bytes or just bits [45]. The six statistical tests are:

1. **Entropy and Compression**: The information density of a file's content, measured in bits per character, provides insight into its randomness [45]. For examples, the results from analyzing an image file reveal a high information density, suggesting the file is almost entirely random and unlikely to be further compressed [45]. This test can be considered two separate tests but in high correlation with each other.

2. **Chi-square Test**: The chi-square test, widely used for assessing data randomness, is particularly sensitive to flaws in pseudo-random number generators [45]. It calculates the chi-square distribution for a file's byte stream and presents it as a number and a percentage, given by the formula

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i},$$

   where χ^2 is chi squared, O_i is the observed value, and E_i is the expected value [45]. This shows how often a truly random sequence would exceed the calculated value [45]. This percentage indicates the suspected non-randomness of the tested sequence. Values over 99% or under 1% suggest the sequence is almost certainly non-random. Percentages between 99% and 95% or between 1% and 5% are suspect, while those between 90% and 95% or between 5% and 10% are "almost suspect." Despite its high information density, that same image file tested is not random, as revealed by the chi-square test [45].

3. **Arithmetic Mean**: This involves summing all the bytes or bits in a file (whichever format is selected) and dividing by the file length [45]. For data that is nearly random, this average should be around 127.5 for bits and 0.5 for bytes [45]. A significant deviation from this value indicates that the data values are consistently high or low [45].

4. **Monte Carlo Value for Pi**: A sequence of six bytes is treated as 24-bit X and Y coordinates inside a square [45]. Each successive one is plotted and if the distance of a randomly generated point falls within the radius of the inscribed circle in the square, it counts as a "mark" [45]. The proportion of marks can be used to estimate the value of Pi. For large datasets (due

[1] https://www.fourmilab.ch/random/

to the slow convergence of this method), the approximation will get closer to the true value of Pi if the sequence is almost random [45].

5. **Serial Correlation Coefficient**: This metric evaluates the dependency of each (byte or bit) in a file on its preceding byte (or bit) [45]. In truly random sequences, this value (which may be positive or negative) will be close to zero [45]. Non-random byte streams, like those found in a C program, will typically have a serial correlation coefficient around 0.5 [45].

The ENT testing suite does not have validation built in to the results. However, when performing analysis on the ENT test results, we referenced ENT test result thresholds as seen in Table 3 of [46], which were also referenced in [4] (primary research we are building on).

4.1.2 PractRand

PractRand[2] is an extensive C++ library of statistical tests as well as pseudo-random number generators [40]. However, we only used the available testing suite pertaining to our research. Provided below are some of the best features of PractRand [40]:

- Provides a standard set of tests, unlike competitors that offer only raw tests without clear parameter choices.
- Quickly detects bias in a wide range of RNGs, outperforming other statistical test suites in variety and speed.
- Supports easy integration via command line tools and allows direct data passing with source code and static libraries for speed and versatility.
- Handles extremely long sequences, tested on over 500 terabytes and expected to work up to several exabytes, unlike competitors with scalability issues.

PractRand is a very versatile testing tool and has implemented a lot of tests that other testing suites offer as well. With the combined speed and ease-of-use, this testing suite contends to be at the top of the list [40].

One unique aspect of PractRand is that it offers a more detailed analysis of tested data [40]. Instead of a ”pass” or ”fail” metric, it offers metrics that may indicate that a sample is “unusual”, “mildly suspicious”, “suspicious”, or “suspect” [40]. All tests contained in the testing suite rely on the ”p-value” statistic for determining the indication of how well the data succeeded at passing [40]. The statistical tests are [40]:

1. **BCFN**: This test examines long-range linear correlations by counting bits, which can often identify Fibonacci-style RNGs that use large lags in data generation to bypass other statistical tests. The first parameter first sets the minimum ”level” for detecting bias, with higher values speeding up the process but potentially missing shorter-range correlations. The second parameter, which is the log-base-2 size of internal tables, helps determine memory and cache usage when testing.

[2] https://pracrand.sourceforge.net/

2. **DC6**: This test examines short-range linear correlations through bit counting. It uses parameters to specify the size of integers it processes internally, the number of adjacent integers it checks for correlations between, and the specific information used for each integer. The test operates as a frequency test on overlapping sets of Hamming weights, which has similarities to other overlapping tests in other testing suites like NIST STS.

3. **Gap16**: The Gap Test is essentially a test that determines the significance of the interval between the recurrence of a certain digit [47]. The Gap16 test provided in PractRand is a very similar variation of this test.

4. **BRank**: This test follows a conventional binary matrix rank assessment method. Its uniqueness lies in the control logic determining when to extract data from the RNG output stream to form matrices and their sizes. However, due to the granularity of its results, obtaining precise p-values for many sub-tests of this test is often very difficult and unfeasible.

5. **FPF (Floating Point Accuracy)**: This test checks for range correlations shorter than that of DC6. This test essentially performs a frequency analysis on the binary representation of floating-point numbers, using the integer values from overlapping segments of the original data stream.

PractRand also appears to have other tests listed, however, these tests are actually in fact wrappers for other tests listed [40]. Since PractRand excels at live testing data streams, the testing suite begins to return testing results at very small intervals starting with a few kilobytes of data. Later on in Chapter 6 when PractRand is used, the reader will have an opportunity to understand how we calibrated PractRand to only return testing results near the size of our sample files, thereby reducing redundant testing data.

4.1.3 TestU01

TestU01[3] is a software library implemented in the C language that offers a collection of testing suite batteries as well as classical implementations of random number generators [41]. TestU01 is different from the other selected testing suites as it has many predefined batteries of statistical testing suites that reside inside TestU01 [41]. These batteries are essentially their own testing suite and have their own orientations towards certain sequences of bits [41]. Some may run more efficiently than others and some may include many more tests than others [41]. For brevity's sake, we will only go into detail about all tests offered by SmallCrush and Alphabit (since they are 10 tests or less), but please refer to [41] for the full descriptions of all tests offered in TestU01 and its statistical testing suites. Provided below are descriptions of each statistical testing suite we selected to use inside of TestU01:

SmallCrush: SmallCrush is one of the smaller but faster statistical testing suites inside of TestU01. In order to run the testing suite, it requires slightly less than 51320000 random numbers [41]. The following tests are applied for this testing suite [41]:

[3] https://simul.iro.umontreal.ca/testu01/tu01.html

1. `smarsa_BirthdaySpacings`: This test simulates throwing n points into $k = d^t$ cells in t dimensions. Imagine each point as a birthday randomly assigned to one of k days. To decide which cell a point falls into, t integers between 0 and $d-1$ are generated and combined differently based on parameter p. The test then calculates the differences between sorted cell numbers and counts collisions among these differences. Under the assumption H_0, the number of collisions follows a Poisson distribution with a specific mean.

2. `sknuth_Collision`: This test checks for collisions among points in a grid, similar to the serial test but focusing on counting how often a point lands in an already occupied cell rather than calculating a chi-square statistic.

3. `sknuth_Gap`: This test focuses on analyzing gaps between successive values generated within the range [0, 1). It counts how many times sequences of exactly s successive values fall outside a specified interval $[\alpha, \beta]$. It then uses a chi-square test to compare the expected and observed counts of these sequences across different values of s.

4. `sknuth_SimpPoker`: This test involves generating n groups of k integers from 0 to $d-1$ using nk calls to the data. For each group, it calculates the number s of distinct integers present. A chi-square test is then applied to compare the expected and observed frequencies of these s values across all groups.

5. `sknuth_CouponCollector`: This test simulates collecting a complete set of d unique items by generating random integers from $\{0, ..., d-1\}$. It tracks how many random draws are needed until each item appears at least once. This process repeats n times. The test then counts how often exactly s draws were required for each s, comparing these counts to expected values using a chi-square test.

6. `sknuth_MaxOft`: This test evaluates the maximum value from groups of t values randomly generated within the interval $[0, 1)$. It creates n such groups, calculates the maximum X for each group, and then compares the empirical distribution of these maximum values against the theoretical distribution $F(x) = x^t$ using both a chi-square test and an Anderson-Darling (AD) test. The chi-square test categorizes the values of X into d groups, where the expected number in each group under the null hypothesis H_0 is n/d. For $N > 1$, it also evaluates the empirical distribution of p-values from the AD test against the AD distribution.

7. `svaria_WeightDistrib`: This test generates k uniform numbers $u_1, ..., u_k$ and computes

$$W = \frac{1}{k} \sum_{j=1}^{k} I[\alpha \leq U_J < \beta],$$

which counts how many u_j's fall within the interval $[\alpha, \beta)$. Under the null hypothesis H_0, W follows a binomial distribution with parameters k and $p = \beta - \alpha$. This process is repeated n time to obtain $W_1, ..., W_n$, and their empirical distribution is compared to the binomial distribution using a chi-square test.

8. `smarsa_MatrixRank`: This test fills an $L \times k$ matrix with random bits using sequences of uniform numbers. Each row of the matrix is filled sequentially by taking k/s bits from a sequence of uniforms. After filling n matrices this way, it counts how many matrices have each rank (number of independent rows). The test then compares this count with the expected distribution of ranks using a chi-square test, adjusting groups if necessary to match theoretical expectations.
9. `sstring_HammingIndep`: This test looks at sequences of bits in blocks of length L. It generates n such blocks and counts the number of 1s in each block. Under the assumption of randomness H_0, the number of 1s in each block follows a binomial distribution with parameters L and 1/2 (meaning, on average, there are L/2 ones per block). It then compares the observed counts of blocks with different numbers of 1s to what would be expected by chance using a chi-square test.
10. `swalk_RandomWalk1`: This test checks random walks on a number line. Starting at 0, each step goes either left or right with equal chance. It looks at walks with different step lengths, starting from L_0 and increasing by 2 steps each time, up to L_1. For example, it first tests walk of length L_0, then $L_0 + 2$, then $L_0 + 4$, and so on. It then computes n values for each of these statistics which represents a value describing the final walk distance from the starting point. It compares the empirical distribution of these values with the theoretical law using a chi-square test.

All tests provided in SmallCrush have default parameters set when run on randomly generated data which can also be found in [41].

Crush: Crush is one of the bigger sized testing suites within TestU01. Not only does Crush include all of the tests offered by SmallCrush, it includes a plethora more including variants of tests [41]. Since the Crush testing suite is so large, the reader may refer to [41] that outlines all 96 tests as well as all of the different default parameters used.

Alphabit: Alphabit is a smaller testing suite than Crush but still larger than SmallCrush. This testing suite has been primarily designed to test hardware random bits generators as well [41]. However, applying Alphabit to any pseudo-random number generator gives one great insight to how random it truly is. Provided below are the tests included in the Alphabit testing suite [41]:

1. `smultin_MultinomialBitsOver`: This test generates n cell numbers using an overlapping approach at the bit level. It arranges n bits in a circular manner, where each block of L consecutive bits determines a cell number. The test doesn't require L and s to be divisible by each other.
2. `sstring_HammingIndep`: Please refer to SmallCrush description of `sstring_HammingIndep`.
3. `sstring_HammingCorr`: The test uses the s most significant bits from each generated random number (excluding the first r bits) to form n blocks of L bits. Each block's

Hamming weight, X_j (number of 1s), is calculated. The test then computes the empirical correlation between successive X_j values using the formula:

$$\hat{\rho} = \frac{4}{(n-1)L} \sum_{j=1}^{n-1} (X_j - \frac{L}{2})(X_{j+1} - \frac{L}{2}).$$

Under the null hypothesis H_0, as n becomes very large, $\hat{\rho}$ divided by $\sqrt{n-1}$ approximates a standard normal distribution. This correlation test is valid only for large values of n.

4. `swalk_RandomWalk1`: Please refer to SmallCrush description of `swalk_RandomWalk1`.

Rabbit: Rabbit is a decently sized testing suite, right between the caliber of the Alphabit and Crush testing suites. Since Rabbit also has quite a large list of tests, please refer to [41] for the complete description of the 26 tests as well as the default parameters used.

4.1.4 Dieharder

Dieharder[4] testing suite is a comprehensive tool that extends George Marsaglia's original Diehard tests by incorporating additional statistical tests and enhancing its ability to detect subtle patterns and correlations in RNG outputs [42]. Key features of Dieharder include its integration with the GNU Scientific Library (GSL), which provides access to a variety of RNG algorithms and statistical functions [42]. This integration allows users to test a wide range of RNGs and evaluate their performance under different conditions [42]. Dieharder specifically provides 18 crafty but extremely effective tests, and they are [42]:

1. **Birthday Spacings**: Tests the distribution of gaps between repeated birthday values assigned by intervals from the data and compares it to the expected Poisson distribution for random numbers.

2. **Overlapping Permutations**: Evaluates the sequence of overlapping 5-tuples of random integers, checking the permutation frequency within the data.

3. **Ranks of 31x31 and 32x32 matrices**: Examines the rank distribution of randomly filled binary matrices using the data, comparing the observed ranks to the expected ranks for random numbers.

4. **Ranks of 6x8 Matrices**: Analyzes the ranks of 6x8 binary matrices generated from the data, assessing whether the rank distribution matches that expected from truly random numbers.

5. **Monkey Tests**: Includes OPSO (Overlapping-Pairs-Sparse-Occupancy), OQSO (Overlapping-Quadruples-Sparse-Occupancy), and DNA (counting certain 4-tuples of bits) to check the frequency of overlapping words of different lengths.

[4] https://rurban.github.io/dieharder/manual/dieharder.pdf

6. **Count the 1s in a Stream of Bytes**: Counts the occurrences of 1s in streams of 8-bit bytes and evaluates if the distribution matches that expected from random data.

7. **Count the 1s in Specific Bytes**: Similar to the previous test but counts the 1s in specific bytes within a larger stream of data.

8. **Parking Lot Test**: Simulates cars parking in a lot and assesses how well the distribution of open spaces matches that expected from randomness.

9. **Minimum Distance Test**: Measures the minimum distance between pairs of points randomly placed in a unit square and compares the observed distribution with the theoretical one for random numbers.

10. **3D Spheres Test**: Evaluates the distribution of distances between random points in a three-dimensional unit cube and compares it to the expected distribution for random points.

11. **Squeeze Test**: Applies a series of transformations to random sequences to determine if the sequences can be compressed as expected for truly random data.

12. **Overlapping Sums Test**: Sums sequences of random numbers and checks if the distribution of these sums matches that expected for true random sequences.

13. **Runs Test**: Assesses the length of runs (sequences of identical bits) in a random sequence and compares it to the expected distribution for random data.

14. **Craps Test**: Simulates a large number of games of craps using the random sequence and checks if the outcomes match the theoretical probabilities for random play.

15. **Serial Correlation (two bytes) Test**: Measures the correlation between successive pairs of bytes in the random sequence, expecting no significant correlation if the sequence is random.

16. **Random Walk (1D) Test**: Simulates a one-dimensional random walk and examines the distribution of the final positions and the number of returns to the origin, comparing it to the expected distribution for a truly random walk.

17. **Random Walk (2D) Test**: Similar to the 1D test but simulates a two-dimensional random walk, evaluating the final positions and path taken for randomness.

18. **Random Walk (3D) Test**: Extends the random walk analysis to three dimensions, checking the distribution of final positions and the paths taken for consistency with random behavior.

Just like PractRand, Dieharder is more unique in that it offers another level of transparency when showing the user how well the data performed on each of the tests. Instead of just a ”pass” and ”fail”, Dieharder also offers a ”weak” metric that only appears if the p-value is suspiciously close to falling into the ”fail” interval [42].

4.1.5 NIST STS

The NIST Statistical Testing Suite (STS)[5] is very unique to that of the other testing suites because it provides a *standardized* and comprehensive set of tests specifically designed to evaluate the randomness of binary sequences [1]. These tests are widely accepted and used in both academic and industrial settings for their rigor and reliability [1]. Its standardization by NIST (National Institute of Standards and Technology) ensures consistency and comparability of results across different applications, making it a trusted tool [1]. The following tests are included in the NIST STS [1]:

1. **Frequency (Monobit) Test**: Determines whether the number of ones and zeros in a sequence are approximately the same. Let n be the length of the bit sequence, and let S_n be the sum of the bits (with zeros counted as -1). The test statistic is

$$S_n = \sum_{i=1}^{n} (2X_i - 1),$$

   where X_i is the i-th bit in the sequence. The sequence is considered random if S_n is close to zero.

2. **Frequency Test within a Block**: Divides the sequence into N non-overlapping blocks of length M. For each block, the proportion of ones π_i is calculated. The test statistic is

$$\chi^2 = 4M \sum_{i=1}^{N} (\pi_i - 0.5)^2.$$

3. **Runs Test**: Evaluates the total number of runs (continuous sequences of the same bit) in the sequence. The number of runs n_r is counted, and the test statistic is

$$\mu = \frac{2n_0n_1}{n} + 1, \quad \sigma^2 = \frac{2n_0n_1(2n_0n_1 - n)}{n^2(n-1)},$$

   where n_0 and n_1 are the counts of zeros and ones, respectively.

4. **Test for Longest-Run-of-Ones in a Block**: Analyzes the longest run of ones within blocks of length M. The test compares the observed distribution of longest runs to the expected distribution using a chi-square test.

5. **Binary Matrix Rank Test**: Tests for linear dependence among fixed-length substrings of the original sequence by constructing matrices. The rank of each $Q \times Q$ matrix is determined, and the distribution of ranks is compared to the expected distribution.

[5] https://www.nist.gov/publications/statistical-test-suite-random-and-pseudorandom-number-generators-cryptographic

6. **Discrete Fourier Transform (Spectral) Test**: Detects periodic features in the sequence by computing the discrete Fourier transform (DFT). The peak heights of the DFT are analyzed to detect deviations from randomness.

7. **Non-overlapping Template Matching Test**: Searches for occurrences of a specified m-bit pattern in the sequence without allowing overlaps. The number of matches in each block is counted, and the test uses a chi-square statistic to compare the observed and expected frequencies.

8. **Overlapping Template Matching Test**: Similar to the non-overlapping test but allows for overlapping occurrences of the specified m-bit pattern. The expected number of matches is given by the Poisson distribution.

9. **Maurer's Universal Statistical Test**: Measures the sequence's compressibility. A dictionary of observed patterns is built, and the entropy of the sequence is estimated. The test statistic is based on the average number of bits between repeated patterns.

10. **Linear Complexity Test**: Evaluates the linear complexity L of the sequence, defined as the length of the shortest linear feedback shift register (LFSR) that can generate the sequence. The test compares L to the expected linear complexity for a random sequence of the same length.

11. **Serial Test**: Compares the frequency of all possible overlapping m-bit patterns. The test uses chi-square statistics for both m-bit and $(m+1)$-bit patterns to evaluate the sequence.

12. **Approximate Entropy Test**: Compares the frequency of overlapping blocks of two consecutive lengths, m and $m+1$. The approximate entropy is given by

$$\phi_m = -\frac{1}{n-m+1} \sum_{i=1}^{n-m+1} \log_2 P(i),$$

    where $P(i)$ is the frequency of the i-th m-bit pattern.

13. **Cumulative Sums (Cusum) Test**: Analyzes the cumulative sum S_j of adjusted (centered) digits. The test statistic is

$$S_j = \sum_{i=1}^{j} (2X_i - 1),$$

    where X_i is the i-th bit in the sequence. The maximum and minimum values of S_j are compared to expected bounds.

14. **Random Excursions Test**: Identifies the number of cycles having exactly K visits to the origin in a random walk. The number of occurrences of each state within cycles is counted and compared to the expected distribution.

15. **Random Excursions Variant Test**: Similar to the Random Excursions Test, but focuses on the number of occurrences of each state (including the origin) within the random walks. The test uses chi-square statistics to compare observed frequencies to expected frequencies.

The NIST STS is also a different testing suite because in order to run it on randomly generated data, user input is required [1]. In Chapter 6 when we elaborate on our testing methodology, the reader will have an opportunity to understand how we navigated this inside the consolidated testing suite tool we developed.

4.2 Random Number Generator Selection

The three selected random number generators were handpicked based on common usage and research-based publication relevance. Each provides a unique perspective in the realm of cloud-based quantum cryptography as well as relevant insight to the state of randomness compared to other researched RNGs. To clarify, each of these RNGs are in fact not RNGs by themselves, but each provide necessary tools to create an RNG specifically for our research purposes. The reader may think of each of these as their own set of Lego pieces, and when an RNG is crafted with their respective pieces, they may operate similarly but with differing results. The constructions of the RNGs will be elaborated on further into this chapter.

In this aim we strove to answer the second question:

> *Which three contemporary RNGs will elicit the best comparative analysis on existing statistical testing research?*

The three selected random number generators are as follows:

- *Python Secrets RNG*
- *Wolfram Language RNG*
- *Microsoft Sparse Simulator RNG*

Provided below, we break each of these RNGs down into their constructions, their origin, and explain the purposes behind why we chose each of them. Each implementation provides both a bit and byte representation, which will be decided upon in Chapter 6 when we answer the question of how should the generated data be formatted.

4.2.1 Python Secrets RNG

The Python *secrets* module is crafted for creating cryptographically secure random numbers and handling sensitive information such as passwords, authentication tokens, and security keys [48]. It is recommended over the random module, which is intended for general-purpose use and not for security applications [48]. We picked the Python *secrets* module as the basis for our first RNG because it is one of the most effective and common modules used to create cryptographically secure applications [49]. Some of the main features of the *secrets* module include [48]:

- `SystemRandom` Class: Uses the highest-quality random number sources from the operating system.
- `choice` Function: Picks a random element from a non-empty sequence.
- `randbelow` Function: Returns a random integer within a specified exclusive upper bound.
- `randbits` Function: Generates an integer with a specified number of random bits.

For the purposes of this thesis, we utilized the `choice` Function. Provided below is the pseudo-code representation of the RNG we constructed using the `choice` Function in the Python *secrets* module:

```
function generate_random_bits(bit_or_byte_length):
    bits_or_bytes = empty list
    for i from 0 to bit_or_byte_length - 1:
        bit_or_byte = choose randomly between 0 and 1
        append bit_or_byte to bits_or_bytes
    return bits_or_bytes
```

Looking at the pseudo-code, the Python implementation is pretty straightforward. The function utilizes the *secrets* module to implement a choice between a ”1” or ”0” in either a bit or byte format.

4.2.2 Wolfram Language RNG

The Wolfram Language is a powerful symbolic programming language known for its comprehensive capabilities in computational mathematics and beyond [36]. Symbolic expressions construct everything in Wolfram Language, facilitating unparalleled programming flexibility [36]. The Wolfram Language excels in randomness generation, leveraging algorithms developed by Wolfram Research [36]. It ensures both efficiency and high-quality results across various types of random variables, whether discrete or continuous [36]. Users can specify a wide array of distributions symbolically, enabling precise control over randomness generation [36]. We picked the Wolfram Language as the basis for our second RNG because of its unique cloud-based architecture as well as its sourced algorithms from Wolfram Research. Key functions within the Wolfram Language include [36]:

- `RandomInteger` and `RandomReal`: Generates random integers or real numbers, individually or in arrays.
- `RandomComplex`: Generates random complex numbers.
- `RandomPoint`: Generates random points within user-specified regions.
- `RandomChoice` and `RandomSample`: Makes random selections or permutations from user-specified lists.
- `RandomPrime`: Generates random prime numbers.

With such an extensive list of randomness functions inside the Wolfram Language, it may be hard to select which function to implement in an RNG construction. However, to stay with consistent generation methodology, we opted to use the RandomChoice function just like in the Python *secrets* module. Provided below is the pseudo-code representation of the RNG we constructed using the `choice` Function in the Wolfram Language:

```
size = generationSize
file = constructFileName(PresentWorkingDirectory, "
   fileToWriteTo")
s = openFile(file, mode="append", format="chosen_format")
for i = 1 to size
    byteOrBit = chooseRandomByteOrBit()
    writeFile(s, byteOrBit)
closeFile(s)
```

The pseudo-code for the Wolfram Language is also pretty straightforward. The implementation utilized the `RandomChoice` function to choose between a ”0” and ”1” in either a bit or byte format. The key difference in this implementation compared to the *secrets* module implementation is that the Wolfram Language is accessed via the cloud. In order to store the generated data, we need to store it within a file accessible by Wolfram Cloud. Once the data is generated, we may download the data and use it as needed for statistical testing purposes.

4.2.3 Microsoft Sparse Simulator RNG

Quantum computing holds great promise for exponential speedups in computational tasks (taking advantage of various quantum algorithms), yet simulating quantum programs on classical hardware faces tough challenges due to the exponential growth in state vector size [34]. However, many quantum states resulting from algorithms exhibit significant sparsity, either inherent in their structure or caused by optimizations [34]. In response to this, Microsoft developed the Microsoft Sparse Simulator which is a novel approach that capitalizes on sparsity to drastically reduce memory usage and simulation runtime [34]. Because of this drastic reduction in memory usage and simulation runtime, we were able to confidently select this as our basis for our third RNG. The RNG was constructed using the Azure Quantum Development Kit [50] as well as the Q# programming language [51]. Since Q# allows you to create and manage individual qubits, we were able to construct an RNG completely from scratch using managed qubits. Provided below is the pseudo-code for the Q# implementation of the RNG:

```
function Random() -> Result:
    q = allocateQubit()
    ApplyHadamard(q)
    result = measure(q)
    reset(q)
    deallocateQubit(q)
    return result

function RandomNBits(N: Integer) -> Result[]:
    results = []
    for i = 0 to N - 1:
        r = Random()
        append results, r
    return results
```

In this provided pseudo-code, we first constructed a function `Random` which allocates a single qubit, applies the Hadamard Gate (placing it in superposition with a 50% chance of being measured as a ”0” or ”1”) and measures it. This result provides us with a simulated random binary ”0” or ”1” value. Next, we implement the `Random` function into the `RandomNBits` function. The `RandomNBits` function takes in a parameter of how many bits to generate and returns the list of the randomly generated bits. Now in order to call this function within Python, we use the Azure Quantum Development Kit [50] to help us out:

```
function ms_sparse_simulator(bit_length: Integer) -> Integer[]:
    bits = call Q# operation RandomNBits(bit_length)
    for i = 0 to length(bits) - 1:
        if bits[i] == "One":
            bits[i] = 1
```

```
        else:
            bits[i] = 0
    return bits
```

Since the binary output of the Q# call represents the binary data as (”One”)’s and (”Zero”)’s, we must account for that by converting them to (”1”)’s and (”0”)’s. This Python function allows us to either make use of the bits stored as bytes or as bits (via bit-packing into a byte), which will be specified in Chapter 6.

Chapter 5

Development of Consolidated Statistical Testing Tool

In this chapter we will elaborate on the second aim of this thesis - focusing on the development of our consolidated statistical testing tool. In completing this task, we finished development of our tool called *Cryptoguard*.

5.1 Cryptoguard Architecture

In this aim we strove to answer the first question:

> *How can we develop a tool that can be easily integrated into any statistical testing environments?*

We developed a highly-integrable tool that allows users to seamlessly apply a myriad of selected statistical testing suites on randomly generated binary data in the form of a Command Line Interface (CLI) tool called *Cryptoguard*. In order to develop this tool, several sub tasks had to be completed:

1. Selection of the development environment that we will develop our tool in.
2. Installation of the selected testing suites so that they can be used inside Cryptoguard while keeping integration independence.
3. Accessibility of Cryptoguard to allow users to be able to access and utilize this tool almost anywhere.

5.1.1 Selection of Development Environment

The selection of the development environment for our tool was vital information for us to plan the remainder of the sub tasks. For Cryptoguard, we chose to implement the tool in Python[1] because of it's flexibility and extensive libraries. Python's versatility allowed us to rapidly prototype and iterate on our ideas, and Python's integration capabilities with other languages such as Bash Scripting enhanced the tool's functionality. These factors collectively made Python an ideal choice for developing Cryptoguard, enabling us to deliver a powerful and effective solution. Now as far as

[1] https://www.python.org/

the kernel, we chose to operate within a Linux environment through an Ubuntu[2] distribution inside a Docker[3] container. This is due in part to Quantinuum's research that facilitates the installation of our selected testing suites in the form of Bash Scripting in an Ubuntu distribution [4]. Provided below is the outline of our completed selections:

1. Language Selection: Python and Bash Scripting
2. Kernel Selection: Linux Kernel through Ubuntu 20.04.6 LTS inside Docker container

5.1.2 Installation of Selected Statistical Testing Suites

The installation of our selected testing suites was made possible through existing research, building off of Quantinuum's research regarding their Statistical Testing Environment [4]. Through their existing Bash Scripts accessible through their published GitHub page[4], we were able to seamlessly install ENT, PractRand, and TestU01 statistical testing suites. Regarding NIST STS, we were able to install this testing suite into the same directory as the others via NIST's access page[5], and access it quite the same. Dieharder is the only statistical testing suite that the user has to manually install, but it is quite an easy task through executing an install command on an Ubuntu or Debian distribution. In order to manually install Dieharder, a user would run

```
sudo apt-get install -y dieharder
```

Since all the statistical testing suites are stored as executable files within Cryptoguard's project directory (excluding Dieharder), the installation of the testing suites is very compartmentalized and independent, which means that we have achieved our sub task's expectation.

5.1.3 Accessibility of Cryptoguard

To make Cryptoguard extremely accessible, we chose to use PyPi[6] to create a Python package that users can install via the `pip` package installer. In order to install Cryptoguard, a user just has to execute

```
pip install cryptoguard
```

Once the user has successfully installed Cryptoguard, they may be able to use the tool and all of its provided features. Assuming they have already installed Dieharder testing suite, they have full access to use this testing suite as desired.

[2] https://ubuntu.com/
[3] https://www.docker.com/
[4] https://github.com/CQCL/random_test?tab=readme-ov-file
[5] https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software
[6] https://pypi.org/

5.2 Cryptoguard Ease-of-Use

In this aim we strove to answer the second question:

In what ways can we make this tool very user-friendly to increase ease-of-use?

First and foremost, Cryptoguard was designed with ease-of-use at top of mind. When developing the tool, we looked to existing research in similar tools, and realized that the NIST STS testing suite was a perfect user-oriented tool to inspire the way Cryptoguard feels and functions. Cryptoguard's functionality essentially has three to four steps of user input depending on which route the user would like to take to test their samples of data, similar to that of the NIST STS testing suite. The reader has an opportunity to understand how similar Cryptoguard and NIST STS look and feel when we go over how we pre-selected the user input for NIST STS below and when we go over Cryptoguard examples further below.

We also wanted to ensure that running any of the statistical testing suites was hassle-free and convenient for the user. Fortunately, every single statistical testing suite we selected can be ran without any user input except for one, the NIST STS. As just discussed, the NIST STS does require user input, but we were able to navigate through this using NIST STS recommended settings along with our desire for what Cryptoguard offers. Provided below are the NIST STS user prompts as shown similarly in [1] along with an explanation for each on how we decided on the inputs:

1. **Executing NIST STS**

```
assess 1000000
```

   Explanation: In order to execute NIST STS, Cryptoguard executes the `assess` executable file which does require one user input. This user input specifies how long each sample should be, and according to NIST STS recommendations in [1], we predefined the bit stream sample length to be 1,000,000 bits.

2. **Generator Selection**

```
          G E N E R A T O R    S E L E C T I O N
          ---------------------------------------

  [0] Input File                  [1] Linear Congruential
  [2] Quadratic Congruential I    [3] Quadratic
     Congruential II
  [4] Cubic Congruential          [5] XOR
  [6] Modular Exponentiation      [7] Blum-Blum-Shub
  [8] Micali-Schnorr              [9] G Using SHA-1

 Enter Choice: 0

              User Prescribed Input File: /binary/file/to
                 /test.bin
```

Explanation: Cryptoguard allows users to specify the source of their random data, such as an input file. This aligns with our aim to support diverse data sources and integrate seamlessly into existing workflows which is why we pre-select ”0” and input the user specified binary file from Cryptoguard.

3. **Statistical Tests**

```
                S T A T I S T I C A L   T E S T S
                ---------------------------------

[01] Frequency                          [02] Block
  Frequency
[03] Cumulative Sums                    [04] Runs
[05] Longest Run of Ones                [06] Rank
[07] Discrete Fourier Transform         [08] Nonperiodic
  Template Matchings
[09] Overlapping Template Matchings     [10] Universal
  Statistical
[11] Approximate Entropy                [12] Random
  Excursions
[13] Random Excursions Variant          [14] Serial
[15] Linear Complexity

     INSTRUCTIONS
        Enter 0 if you DO NOT want to apply all of the
        statistical tests to each sequence and 1 if you
            DO.

Enter Choice: 1
```

Explanation: Users are given the option to apply all statistical tests to each sequence of data, promoting comprehensive analysis. This choice reflects Cryptoguard’s commitment to thorough testing and reliability assessment which is why we pre-select ”1” to run all of them.

4. **Parameter Adjustments**

```
      P a r a m e t e r   A d j u s t m e n t s
      -----------------------------------------
[1] Block Frequency Test - block length(M):          128
[2] NonOverlapping Template Test - block length(m): 9
[3] Overlapping Template Test - block length(m):     9
[4] Approximate Entropy Test - block length(m):      10
[5] Serial Test - block length(m):                   16
[6] Linear Complexity Test - block length(M):        500
```

```
Select Test (0 to continue): 0
```

Explanation: Cryptoguard honors NIST STS recommendations for the predefined parameters and does not opt to change them in any way. Thus, we pre-select ”0” to continue.

5. **Bitstreams and Input File Format**

```
How many bitstreams? 100

Input File Format:
 [0] ASCII - A sequence of ASCII 0's and 1's
 [1] Binary - Each byte in data file contains 8 bits of
    data

Select input mode:  1
```

Explanation: Users specify the number of bitstreams to test and the format of the input file. This step is following NIST's recommendation of 100 samples of 1,000,000 bits in correlation to the p-value they use in their post-test analysis to determine if a sample passes a test or not [1]. Please see [1] for more information on this matter.

To provide transparency for the user when selecting NIST STS to run on their data, a file called *nist_input.txt* is stored in the user specified results directory that stores all of these pre-selections. Since the user currently has no control on customizing the NIST STS inputs, every *nist_input.txt* file will contain the same pre-selected input content. The contents of the file looks like

```
0
/path/to/sample.bin
1
0
100
1
```

where each line corresponds to each pre-selected input for each user prompt discussed above.

Although the other statistical testing suites do not require any user input, we will go in depth on how they are executed to provide transparency and insight on the ease-of-use aspect for the other testing suites. As a matter of fact, every testing suite is run following the same structure. To be specific, there is exactly one bash script file for each testing suite, and each bash script executes the respective testing suite with efficiency. Each script is housed in a sub directory within Cryptoguard called *testing_suite_scripts*. Provided below is the pseudo-code outline for an arbitrary bash script that runs a testing suite:

```
# Start
# Read command-line arguments BINARY_FILE and RESULT_DIR
BINARY_FILE = $1
RESULT_DIR = $2
```

```
# Check if BINARY_FILE exists
if not exists(BINARY_FILE):
    print("Error: The specified binary file '" +
       BINARY_FILE + "' does not exist.")
    exit(1)

# Check if RESULT_DIR exists
if not exists(RESULT_DIR):
    print("Error: The specified directory '" + RESULT_DIR +
        "' does not exist.")
    exit(1)

# Calculate path to the testing suite directory
TEST_SUITE_DIR = dirname(realpath($0)) + "/../
   testing_suites"

# Execute the testing suite with timing
start_time = current_time()
execute("$TEST_SUITE_DIR/{testing_suite}", BINARY_FILE,
   output="> " + RESULT_DIR + "/test_{testing_suite}.log")
end_time = current_time()

# Record execution time
time_taken = end_time - start_time
write_to_file("{testing_suite}", RESULT_DIR + "/time")
append_to_file(read_file(RESULT_DIR + "/time_{testing_suite
   }"), RESULT_DIR + "/time")
remove_file(RESULT_DIR + "/time_{testing_suite}")

print("{testing_suite} test completed.")
# End
```

Going through this incrementally:

1. The command-line arguments are fed into the bash script. Every testing suite script receives the same arguments, ensuring consistency across executions. Error handling checks ensure that essential arguments are properly validated, though in the context of Cryptoguard managed by Python, such errors are anticipated to be managed before reaching this stage.

2. Next, the script executes the testing suite by substituting the user-specified binary file into the appropriate place in the testing suite executable. This ensures that each testing suite operates on the intended data set, maintaining accuracy and relevance to the user's input.

3. During execution, the script measures the time taken to complete the testing suite using a timing mechanism. This provides insight into the efficiency and performance of the testing suite, which is crucial for users evaluating the reliability of their data.

4. Upon completion, the script records the execution time in a designated temporary file *time_*{*testing_suite*} within the specified result directory *RESULT_DIR*. This temporary file then gets copied over to the official run time file *time*. The output from each statistical testing suite is also stored in their own respective log file as well. This systematic logging allows users to review and compare the performance of different testing suites over time as well as dive into the statistical analysis of each testing suite.

5. Lastly, the script ensures cleanup by removing any temporary files or logs *time_*{*testing_suite*} generated during the execution. This maintains the cleanliness and organization of the result directory, facilitating ease of interpretation and management of test results.

5.3 Detailed Steps Explanation

In this aim we strove to answer the third question:

What input should a user have to specify in order to test randomly generated data?

To test randomly generated data using Cryptoguard, a user should specify:

1. **Binary File Path:** The path to the binary file containing the data to be tested.

2. **Testing Setting:** The desired testing setting, such as Light, Recommended, All, or Custom. This is following the proposed testing settings outlined in Table 1 of [4] (excluding the Custom setting).

3. **Custom Setting Input (if applicable):** A binary representation indicating which specific tests to run, if the Custom setting is selected.

4. **Result Directory:** The directory where the test results should be stored.

Each of these steps will be elaborated on in the next few subsections. In Diagram 5.1, Cryptoguard's workflow can be understood from a visual perspective.

5.3.1 Handling Binary File Input

The tool prompts the user to specify the path to the binary file that needs to be tested. If the file does not exist or is not a valid file, it will prompt the user again until a valid file path is provided. This ensures that the testing process starts with a legitimate data file, preventing errors and ensuring the integrity of the test results. The data file can either be input as an absolute path:

```
/absolute/path/to/sample
```

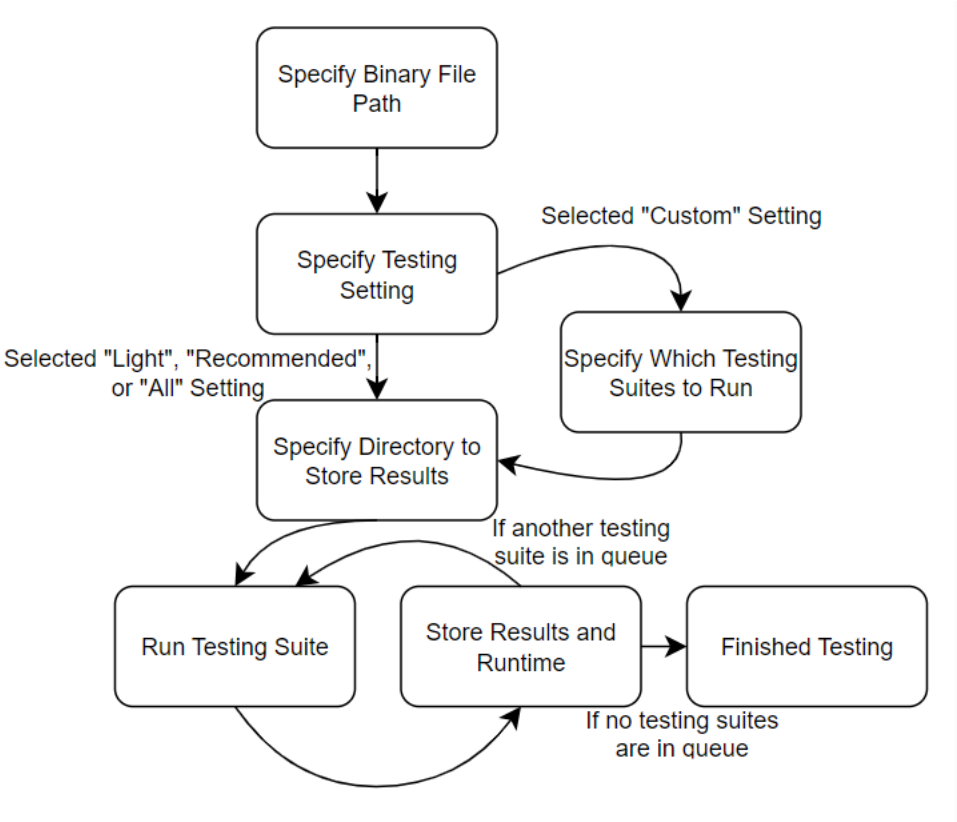

Figure 5.1: Block diagram representing Cryptoguard workflow

Or, the data file can be input as a relative path corresponding to the user's present working directory:

```
relative/path/to/sample
```

5.3.2 Handling Setting Input

The tool displays a list of available testing settings and prompts the user to select one. The available settings include:

- Light: ENT, PractRand, SmallCrush
- Recommended: ENT, PractRand, Rabbit, Dieharder, NIST STS
- All: All testing suites
- Custom: User-defined binary representation of the tests to run

The user selects a setting by entering the corresponding number. This selection determines which predefined set of tests will be applied to the binary file. To reiterate, the first three testing settings are based on the proposed settings outlined in Table 1 of [4].

5.3.3 Handling Custom Setting Input

For the custom setting, the tool prompts the user to enter a binary string where each bit represents whether a corresponding testing suite should be run. The length of the binary string must match the number of available testing suites. This allows users to tailor the testing process precisely to their needs by selecting specific testing suites. A detailed example will be shown in the next section for a more visible explanation.

5.3.4 Handling Directory Input

The tool prompts the user to specify a directory to store the results. If the directory does not exist, it will be created. This ensures that all test outputs are saved in a designated location, making it easy to access and review the results. The user can either specify a custom directory or use the current working directory by pressing ENTER. The same absolute and relative path input specifications apply as shown in Subsection 5.3.1.

5.3.5 Running Testing Suites

The tool iterates over the selected testing suites and runs each suite on the specified binary file. The results are stored in the specified directory, and the elapsed time for each test is displayed.

5.3.6 Handling Command-Line Arguments

The tool also supports command-line arguments for all inputs, allowing users to run the tool non-interactively by providing all necessary information upfront.

```
usage: cryptoguard.py [-h] [-v] [-l] [-g] [-b BINARY_FILE] [-s
   {1,2,3,4}] [-i BINARY_SETTING] [-d DIRECTORY]

cryptoguard is a Python package designed for conducting
   comprehensive testing of random number generators. It
   provides a collection of testing suites that evaluate the
   statistical properties and reliability of random number
   sequences.

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -l, --list-suites     List all available testing suites
  -g, --list-settings   List all available testing settings
  -b BINARY_FILE, --binary-file BINARY_FILE
                        Binary file to test
  -s {1,2,3,4}, --setting {1,2,3,4}
```

```
                        Testing setting number (1: Light, 2:
                          Recommended, 3: All, 4: Custom)
-i BINARY_SETTING, --binary-setting BINARY_SETTING
                        Binary representation of the tests to
                          run (only for Custom setting)
-d DIRECTORY, --directory DIRECTORY
                        Directory to store the results (will be
                            created if it doesn't exist)
```

5.4 Cryptoguard Tool Examples

The following examples demonstrate how to use the Cryptoguard tool in different scenarios.

5.4.1 Example 1: Custom Setting with Specific Tests

In this example, we start the Cryptoguard tool without any predefined user input, select the 'Custom' setting, and specify which tests to run.

1. Start the Cryptoguard tool:

```
cryptoguard
```

2. The tool prompts for a binary file to test:

```
B I N A R Y   F I L E
--------------------

Please specify the path to the binary file to test: /
  path/to/sample/sample.bin
```

3. The tool displays available testing settings and prompts for selection:

```
T E S T I N G   S E T T I N G S
------------------------------

[1] Light             (ENT, SmallCrush, PractRand)
[2] Recommended       (Dieharder, ENT, PractRand, NIST
  STS)
[3] All               (Rabbit, SmallCrush, PractRand,
  Alphabit, Dieharder, NIST STS, ENT, Crush)
[4] Custom

Please select testing setting (number): 4
```

4. Since 'Custom' is selected, the tool displays available testing suites and prompts for binary input:

```
        T E S T I N G   S U I T E S
        --------------------------

 [1] ENT                  [2] PractRand
 [3] SmallCrush           [4] Crush
 [5] Alphabit             [6] Rabbit
 [7] Dieharder            [8] NIST STS

Enter a 0 or 1 to indicate whether or not the numbered
   statistical
testing suite should be applied.

 12345678
 10100010
```

   Here, the user specifies the binary string '10100010' to run the ENT, SmallCrush, Dieharder, and NIST STS tests.

5. The tool prompts for the result directory:

```
        R E S U L T   D I R E C T O R Y
        -------------------------------

Please specify directory name to store results in (ENTER
    for pwd): results_directory
```

6. The tool runs the selected testing suites and stores the results in the specified directory.

5.4.2 Example 2: All User-Defined Input with Recommended Setting

In this example, we provide all necessary inputs at the beginning using command-line arguments, and the setting chosen is 'Recommended'.

1. Start the Cryptoguard tool with all inputs predefined:

```
cryptoguard -b path/to/binary/file.bin -s 2 -d
   results_directory
```

2. The tool automatically selects the 'Recommended' setting and runs the corresponding testing suites (ENT, PractRand, Rabbit, Dieharder, and NIST STS).

3. The results are stored in the specified directory.

5.4.3 Example 3: User-Defined Input for Custom Setting with Specific Tests

In this example, we choose the 'Custom' setting and specify which tests to run by providing a binary representation.

1. Start the Cryptoguard tool with a custom binary setting:

```
cryptoguard -b path/to/binary/file.bin -s 4 -i 10101000
    -d results_directory
```

2. The tool interprets the custom setting '10101000' as follows:

   - '1': Run the ENT test
   - '0': Skip the PractRand test
   - '1': Run the SmallCrush test
   - '0': Skip the Crush test
   - '1': Run the Alphabit test
   - '0': Skip the Rabbit test
   - '1': Run the Dieharder test
   - '0': Skip the NIST STS test

3. The results are stored in the specified directory.

5.4.4 Live Output of Cryptoguard

When running Cryptoguard, the tool provides live output to keep the user informed about the progress of the testing process. Below is an example of what the live output looks like when running the ENT testing suite:

```
Running the ENT testing suite...
Elapsed time: 0:00:01.056307
```

This output indicates that the tool is currently executing the ENT testing suite. The elapsed time is displayed continuously, showing how long the suite has been running. This feature allows the user to monitor the progress in real-time and estimate the remaining time for the tests to complete.

Additionally, for other testing suites, the output follows a similar format, with the suite's name and the elapsed time being updated periodically until the suite completes its execution. This live feedback is crucial for keeping users informed and ensuring transparency in the testing process.

5.4.5 Logging and Results Storage

Cryptoguard provides comprehensive logging and results storage capabilities to ensure that users can review the outcomes of their tests in detail. For each testing suite that is run, a separate log file is created in the user-specified results directory. The log files are named according to the format

```
test_{testing_suite}.log
```

where *testing_suite* is the name of the specific testing suite. This naming convention is used similarly within [4] as well.

Each log file contains detailed output exclusively from the corresponding testing suite executable, including results and any messages generated during the test runs. These files allow users to perform in-depth analysis and keep records of test outcomes.

In addition to individual log files, Cryptoguard also creates a file named `time` in the results directory. This file records the run times of each testing suite that was executed. The `time` file provides a summary of how long each test took to complete, facilitating assessment of test performance and efficiency. An example of a `time` file with every statistical testing suite ran looks like:

```
ENT

real    1m29.267s
user    1m27.224s
sys     0m2.000s

PractRand

real    2m38.069s
user    2m32.939s
sys     0m4.843s

SmallCrush

real    0m31.975s
user    0m30.560s
sys     0m1.389s

Crush

real    140m54.474s
user    135m20.261s
sys     5m13.720s
```

```
Alphabit

real    12m42.504s
user    12m36.140s
sys     0m6.040s

Rabbit

real    51m18.468s
user    50m27.754s
sys     0m43.864s

Dieharder

real    7m56.351s
user    6m53.250s
sys     1m22.658s

NIST

real    29m25.487s
user    29m6.115s
sys     0m15.650s
```

Overall, these logging and storage features enhance the usability and transparency of Cryptoguard, enabling users to easily access and analyze their test results.

Chapter 6

Generation of Data and Statistical Testing Results

After selecting the statistical testing suites, the random number generators, and developing the consolidated statistical testing CLI tool, Cryptoguard, we needed to generate our random data and use Cryptoguard for statistical analysis purposes.

6.1 Format of Random Data

In this aim we strove to answer the first question:

In what format should we generate the random data in?

Considering we developed Cryptoguard to handle files in binary format, the more obvious answer would be to generate the random data samples in binary format. This observation would be correct, and we opted to pursue generation of binary data for all three of our selected random number generators. Provided below are also some advantages to generating data in binary format rather than byte format:

- **Compactness:** Binary format (1 = 1) can be more compact than byte format (1 = 00000001), especially when dealing with large datasets (in which this research dealt with). This efficiency in storage is advantageous in resource-constrained environments or when handling massive amounts of data.
- **Direct Representation:** Binary format directly represents data as sequences of 0's and 1's, aligning closely with how data is processed at the hardware level in computing systems. This direct representation simplifies certain types of computations and analyses.
- **Compatibility:** Many statistical testing suites and cryptographic analysis tools are designed to process data in binary format. Using binary format ensures compatibility and seamless integration with these tools, avoiding unnecessary conversions or data transformations.

For the Python *secrets* RNG, the Wolfram Cloud RNG, and the Microsoft Sparse Simulator RNG we were able to successfully generate random binary 0's and 1's and bit pack them into bytes of data. Bit packing into a byte involves efficiently consolidating multiple individual bits of data into a single byte. This method optimizes storage and processing efficiency by reducing the number of storage units needed by a factor of 8. The pseudo-code for bit packing is provided below:

```
# Generate random bits
bits = generator(bit_length)

# Initialize an empty byte array and variables for the current
   byte and bit count
byte_data = []
current_byte = 0
bit_count = 0

# Pack the bits into bytes
for each bit in bits:
    if bit is 1:
        set the (7 - bit_count)th bit of current_byte to 1
    else if bit is 0:
        do nothing (bit is already 0 by default)

    increment bit_count by 1
    if bit_count equals 8:
        append current_byte to byte_data
        reset current_byte to 0
        reset bit_count to 0

# If there are remaining bits, add the last byte
if bit_count > 0:
    append current_byte to byte_data
```

Elaborating each step of this pseudo-code we can observe that:

1. First, we generate random bits using a generator function.
2. Next, we initialize an empty byte array and variables for the current byte and bit count.
3. Then, we iterate over each bit in the generated bits:
   (a) If the bit is 1, we set the corresponding bit in the current byte to 1.
   (b) If the bit is 0, we do nothing (bit is already 0 by default).
   (c) We increment the bit count by 1.
   (d) If the bit count reaches 8, we append the current byte to the byte array, reset the current byte to 0, and reset the bit count to 0.
4. Finally, if there are any remaining bits after the loop, we append the last byte to the byte array.

Every sample of binary data was processed through this bit packing code implemented in Python.

6.2 Sample Length of Random Data

In this aim we strove to answer the second question:

What should the generated size of each sample be?

Since our statistical testing data was intended for comparative analysis with the existing data published in [4], we adopted the same random data generation methodology. Additionally, we selected the same testing suites (with the addition of Crush) to maximize our comparative analysis opportunities. Their methodology is as follows [4]:

- Generate 10 samples of random data from each of the selected RNGs.
- Each sample must be 10Gbit in length.
- Statistical testing suites must execute directly on each sample, except for NIST, where only 100Mbit of each sample is tested (100 bitstreams of 1,000,000 bits outlined in Section 5.2).

6.3 Data Generation and Testing Methodology

In this aim we strove to answer the third question:

How do we consistently generate and test each sample of data?

Both of the data generation and testing methodologies follow in close alignment with Quantinuum's research methodologies [4] when they generated and tested a select number of RNGs, which we used in comparative analysis in Chapter 7. Provided below are the methodologies broken down for the reader:

6.3.1 Data Generation Methodology

Building on the previous sections that discussed the format and length of the data files, this section focuses on our data generation process. We generated 10Gbit samples of random data using the RNGs constructed from each of the three bases outlined in Chapter 4. To preface, all data generation was conducted on a Windows Laptop 2 with 16GB RAM and a 1.90GHz Intel i7-8650U processor, running the Windows 12 operating system. The procedure for each RNG is outlined below:

Python *secrets* RNG: To generate data with this RNG, we created 100 sub-samples of 100,000,000 bits of binary data each, and concatenated them into a single 10Gbit data file. This approach was chosen primarily to manage RAM usage effectively, as handling smaller sub-samples prevents excessive memory consumption and potential system slowdowns. Additionally, generating 100 sub-samples allowed us to monitor progress via an updated progress bar showing the number of sub-samples generated. We also implemented a timer within the generator to track how long it took for one sample to generate.

Wolfram RNG: Since the Wolfram RNG was constructed and executed within the Wolfram Cloud environment, we had to work within the confines of Wolfram Cloud restrictions. The primary restriction we navigated was the runtime limit of any executable code sequence. Operating under a standard subscription (not premium), we decided to generate 50,000,000 bits of random binary data per code execution, cycling through 200 code executions. We implemented a timer within each code execution and summed all timer outputs to determine the total time required to generate one sample of data.

Microsoft Sparse Simulator RNG For this RNG, we followed the same generation process as the Python *secrets* RNG. We generated 100 sub-samples of 100,000,000 bits of binary data each and concatenated them into a single 10Gbit data file. This method was chosen to manage RAM usage effectively, as handling smaller sub-samples prevents excessive memory consumption and potential system slowdowns. Additionally, by generating 100 sub-samples, we were able to monitor the generation progress via an updated progress bar indicating the number of sub-samples generated. We also included a timer within the generator to track the time taken to generate one sample.

All ten generated samples of data from each of the RNGs were stored in a simple directory structure as so:

```
data/
|
|-- secrets_rng/
|   |-- secrets_0.bin
|   |-- ...
|   '-- secrets_9.bin
|
|-- wolfram_rng/
|   |-- wolfram_0.bin
|   |-- ...
|   '-- wolfram_9.bin
|
'-- sparse_rng/
|   |-- sparse_0.bin
|   |-- ...
|   '-- sparse_9.bin
```

All of the generation timing data can be viewed in Appendix A.1.

6.3.2 Statistical Testing Methodology

The methodology elaborated on in this subsection focuses primarily on how we tested the generated samples of random binary data using Cryptoguard. Because Cryptoguard was developed for specialized use on a Linux kernel (i.e. Ubuntu or Debian distribution), we tested our data within the same environment. To preface, all statistical testing was conducted on a Windows Laptop 2 with 16GB RAM and a 1.90GHz Intel i7-8650U processor, running the Ubuntu 20.04.6 LTS operating system.

In order to obtain complete statistical testing coverage from all testing suites on our data samples, we decided to run Cryptoguard with the "All" testing setting selected. Since each sample was stored in an organized fashion, we were able to take advantage of `for-loop` logic within the command line as well. Provided below was our command that we executed once per RNG:

```
for i in ‘seq 0 9‘; do cryptoguard -b data/{rng_name}_rng/{
   rng_name}_$i.bin -s 3 -d results/{rng_name}_rng/{
   rng_name}_$i; done
```

All of the statistical testing suite results are stored in a simple directory structure as so:

```
results/
|
|-- secrets_rng/
|   |-- secrets_0
|   |   |-- test_ent.log
|   |   |-- test_practrand.log
|   |   |-- ...
|   |   '-- time
|   |-- ...
|   '-- secrets_9
|   |   |-- test_ent.log
|   |   |-- test_practrand.log
|   |   |-- ...
|   |   '-- time
|
|-- wolfram_rng/
|   |-- wolfram_0
|   |   |-- test_ent.log
|   |   |-- test_practrand.log
|   |   |-- ...
|   |   '-- time
|   |-- ...
|   '-- wolfram_9
|   |   |-- test_ent.log
|   |   |-- test_practrand.log
|   |   |-- ...
```

```
|   |   '-- time
|
'-- sparse_rng/
|   |-- sparse_0
|   |   |-- test_ent.log
|   |   |-- test_practrand.log
|   |   |-- ...
|   |   '-- time
|   |-- ...
|   '-- sparse_9
|   |   |-- test_ent.log
|   |   |-- test_practrand.log
|   |   |-- ...
|   |   '-- time
```

All of the statistical testing suite results can be viewed in Appendix A.2, and all statistical testing suite runtimes can be viewed in Appendix A.3.

Chapter 7

Analysis

After obtaining the statistical testing results from the ten samples respective to each of the three RNGs we selected, we were able to gain significant insight on how well these three RNGs perform when tested beyond what is required by standardization bodies like NIST. Additionally, using the statistical testing data provided by Quantinuum's research [4], we were able to also conduct discerning comparative analysis between our selected PRNGs and two contemporary QRNGs, Intel RDSEED and IDQ Quantis QRNG. It is important to note that all statistical testing data from the Crush testing suite will not be included in the analysis. The primary reason for this is to stay consistent with Quantinuum's analysis in [4], as this keeps our comparative analysis seamless.

Outlined below is an itemized list of what statistical data we analyzed:

- RNG Generation
- Number of Failed Statistical Tests
- Number of Failed and Suspicious Statistical Tests
- Statistical Testing Suite Runtimes
- Selected PRNGs vs. IDQ Quantis QRNG
- Selected PRNGs vs. Intel RDSEED

7.1 RNG Generation Times

After generating each of the ten samples for each of the three RNGs we selected, we recorded the generation times. RNG generation times give us immediate insight to how fast each RNG may

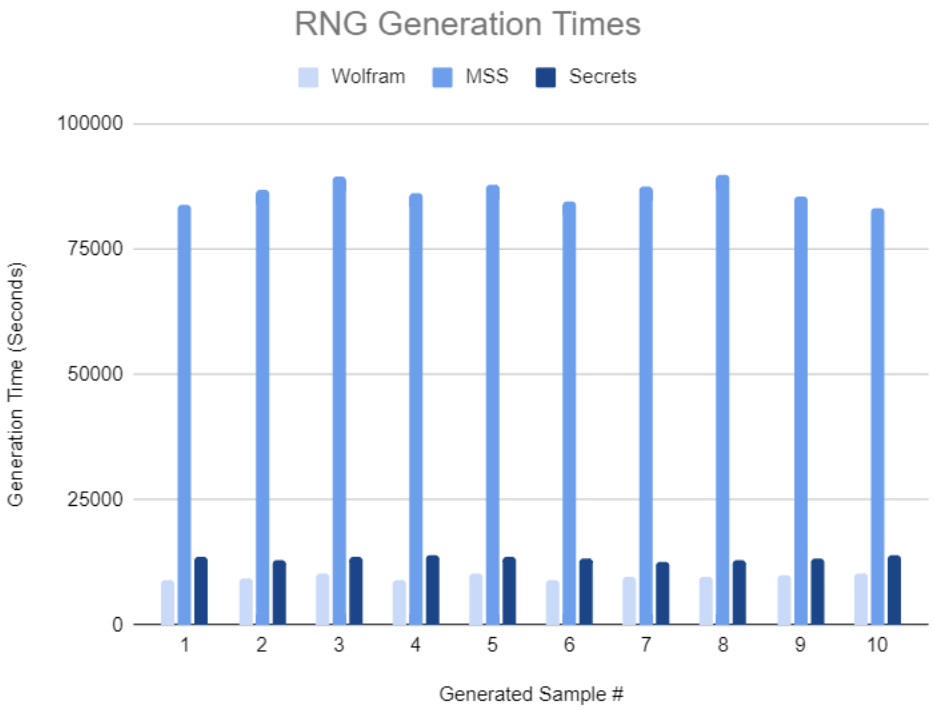

Figure 7.1: Column chart depicting the RNG generation times for each of the 10 samples of data respective to each of the 3 selected RNGs.

produce random binary output. In Figure 7.1, the reader may visually understand how each RNG performed when generating the ten random binary files in a side-by-side comparison. The vertical axis shows the generation times of each sample respective to its RNG in seconds. In terms of averages:

- Wolfram RNG averaged a total of 2 hours 39 minutes and 58 seconds to generate a sample (9,598 seconds).
- Microsoft Sparse Simulator RNG averaged a total of 24 hours 1 minute and 2 seconds to generate a sample (86,462 seconds).
- Python Secrets RNG averaged a total of 3 hours 42 minutes and 48 seconds to generate a sample (13,368 seconds).

The main observation that can be made according to this data is that the Microsoft Sparse Simulator RNG took by far the most amount of time to generate the binary data, followed by the

Python Secrets RNG, and the fastest generator, the Wolfram RNG. The raw data can be viewed in Appendix A.1.

7.2 Number of Failed Statistical Tests

Once we ran all of the statistical testing suites on every sample of data respective to their RNG, we were able to make concise observations on how well each RNG performed. In Figure 7.2,

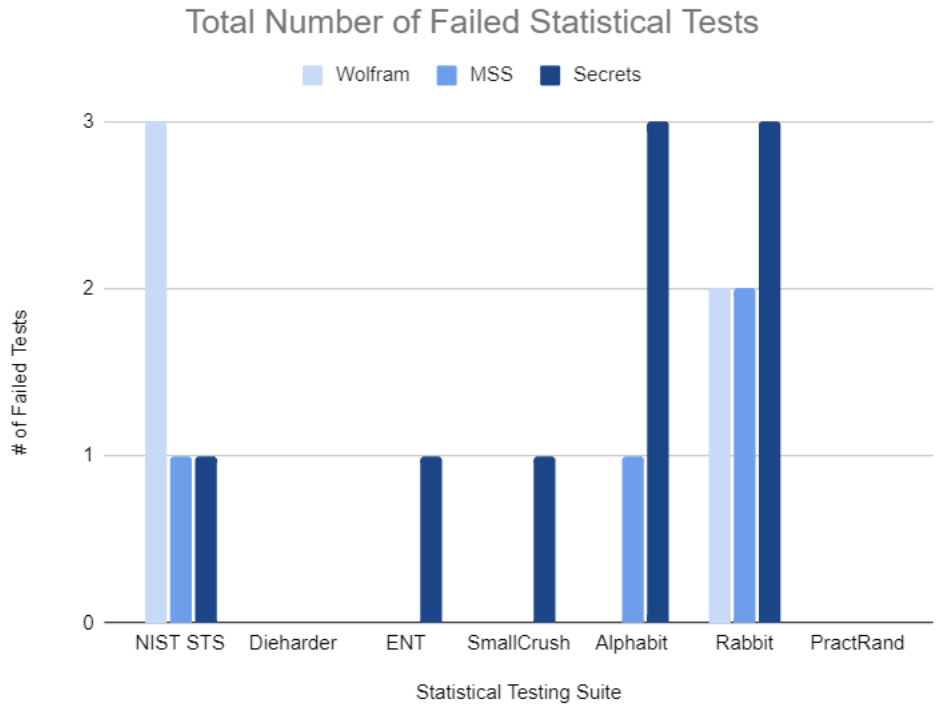

Figure 7.2: Column chart depicting the total number of failed tests over ten samples for each statistical testing suite for each of the RNGs.

the reader may visually understand how many statistical tests each RNG failed for each testing suite. The data that is displayed in this figure represented the total number of failed tests over the ten samples of randomly generated binary data. We were able to determine just how well they performed compared to each other as well as how well they performed against the standardized NIST STS compared to the other non-standardized testing suites.

When comparing the performance of each RNG against the others, an initial observation reveals that the Python Secrets RNG exhibited the highest number of failed tests among the three. Specifically, it failed one test each in NIST STS, ENT, and SmallCrush, and failed three tests in both Alphabit and Rabbit, resulting in a total of nine failed tests. The Wolfram RNG followed with the second highest number of failed tests, failing three tests in NIST STS and two in Rabbit, totaling five failed tests overall. In contrast, the Microsoft Sparse Simulator RNG performed the best, with only one failed test in both NIST STS and Alphabit, and two failed tests in Rabbit, totaling four failed tests. Based solely on ”pass” and ”fail” rates, the Microsoft Sparse Simulator RNG emerges as the most successful performer in this comparison. The raw data can be viewed in Appendix A.2.

7.3 Number of Failed and Suspicious Statistical Tests

Since both the Dieharder and PractRand statistical testing suites offer in-depth metrics on how well a random binary file performs on their statistical tests, we were also able to analyze how well each RNG performed by summing both the failed and ”suspicious” test results. Since only

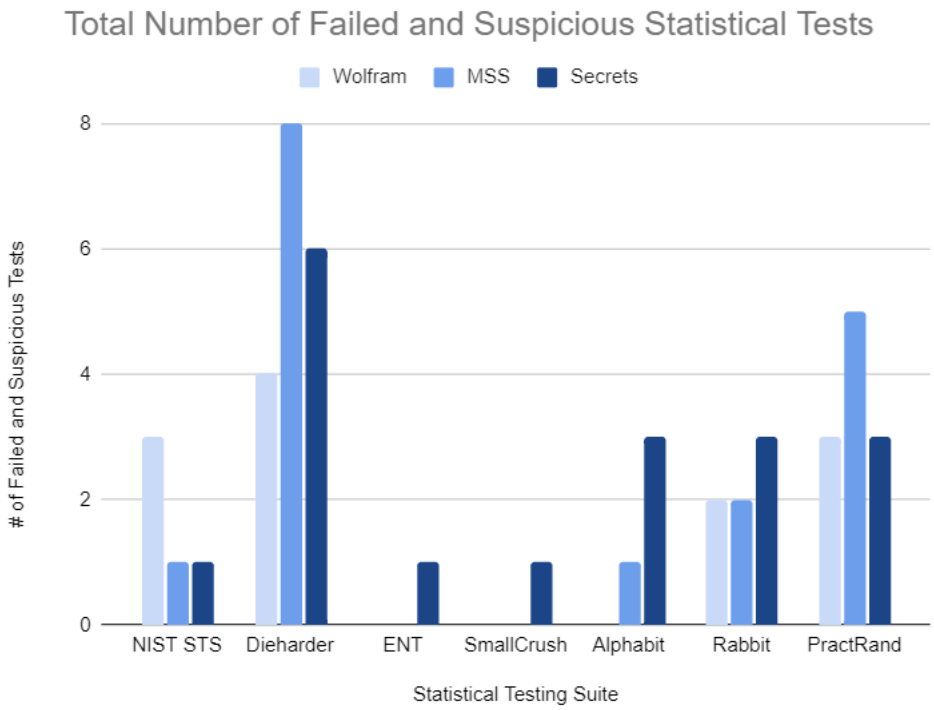

Figure 7.3: Column chart depicting the total number of failed and suspicious tests over ten samples for each statistical testing suite for each of the RNGs.

the Dieharder and PractRand statistical testing suites offered these additional metrics, Figure 7.3 displays the same numerical data as Figure 7.2 with the addition of the new ”suspicious” test results data. Upon analyzing the figure alongside the numerical data, similar insights emerge regarding the performance of each RNG. However, a closer examination focused strictly on the number of suspicious tests highlights that the Microsoft Sparse Simulator shows the highest count, with eight suspicious tests from Dieharder and five from PractRand, totaling thirteen suspicious tests. In contrast, the Python Secrets RNG follows with six suspicious tests from Dieharder and three from PractRand, totaling nine suspicious tests. The Wolfram RNG demonstrates the most favorable performance, with four suspicious tests from Dieharder and three from PractRand, totaling seven suspicious tests. The raw data can be viewed in Appendix A.2.

When considering the cumulative number of failed and suspicious tests across all testing suites, a nuanced perspective emerges. The Python Secrets RNG registers the highest combined count of failed and suspicious tests, totaling eighteen instances. The Microsoft Sparse Simulator follows

closely behind with fifteen combined failed or suspicious tests. In contrast, the Wolfram RNG stands out as the most successful performer, with a combined total of twelve failed or suspicious tests, indicating its strong performance in passing the majority of statistical tests with satisfactory results.

At this point in the statistical analysis, two conclusions can be made regarding which RNG performed the best out of the three. First, when evaluating based on the number of failed statistical tests, the Microsoft Sparse Simulator RNG exhibited the best performance, with only four failed tests in total—one each in NIST STS and Alphabit, and two in Rabbit. In contrast, the Wolfram RNG had five failed tests, while the Python Secrets RNG had the highest number of failed tests at nine. Therefore, based solely on the number of failed tests, the Microsoft Sparse Simulator RNG is the clear winner.

Second, when considering the cumulative number of both failed and suspicious tests, the conclusions shift slightly. The Python Secrets RNG had the highest combined count, with eighteen failed or suspicious tests in total. The Microsoft Sparse Simulator RNG followed with fifteen, and the Wolfram RNG had the fewest combined failed or suspicious tests at twelve. While the Microsoft Sparse Simulator RNG showed strong performance in terms of failed tests alone, the Wolfram RNG emerged as the most successful overall when taking into account both failed and suspicious tests, indicating its strong performance across a broader spectrum of statistical evaluations.

7.4 Statistical Testing Suite Runtimes

The next section presents a column chart illustrating the runtimes for all the statistical testing suites. The observations from Figure 7.4 indicate that the Wolfram RNG random data took the

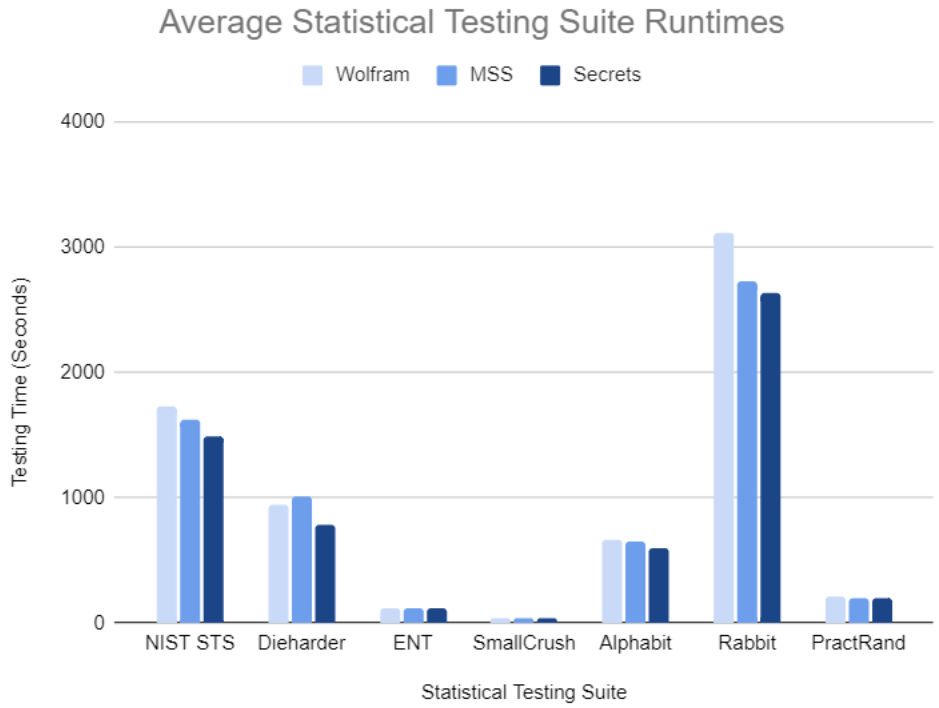

Figure 7.4: Column chart depicting the average runtimes each statistical testing suite took to run over the ten samples for each of the RNGs

longest time to run each testing suite, with average runtimes of 28 minutes and 42 seconds (1722 seconds) for NIST STS, 15 minutes and 44 seconds (944 seconds) for Dieharder, 2 minutes and 3 seconds (123 seconds) for ENT, 42 seconds for SmallCrush, 11 minutes and 4 seconds (664 seconds) for Alphabit, 51 minutes and 54 seconds (3114 seconds) for Rabbit, and 3 minutes and 32 seconds (212 seconds) for PractRand. The Microsoft Sparse Simulator RNG followed, with average runtimes of 26 minutes and 59 seconds (1619 seconds) for NIST STS, 16 minutes and 51 seconds (1011 seconds) for Dieharder, 2 minutes and 1 second (121 seconds) for ENT, 37 seconds for SmallCrush, 10 minutes and 48 seconds (648 seconds) for Alphabit, 45 minutes and 30 seconds (2730 seconds) for Rabbit, and 3 minutes and 15 seconds (195 seconds) for PractRand. The Python Secrets RNG exhibited the shortest runtimes, averaging 24 minutes and 42 seconds (1482 seconds) for NIST STS, 12 minutes and 58 seconds (778 seconds) for Dieharder, 1 minute and 52 seconds (112 seconds) for ENT, 38 seconds for SmallCrush, 9 minutes and 55 seconds (595 seconds) for Alphabit, 43 minutes and 45 seconds (2625 seconds) for Rabbit, and 3 minutes and 14 seconds (194 seconds) for PractRand.

This difference in runtimes may provide insights into the computational efficiency and complexity of the generated random sequences. Longer runtimes for the Wolfram RNG and Microsoft Sparse Simulator RNG could suggest that their sequences exhibit more complex patterns or structures, requiring more processing power and time to analyze. In contrast, the shorter runtime for the Python Secrets RNG might indicate simpler or more straightforward sequences. However, these runtimes do not necessarily correlate directly with the quality or robustness of the RNGs. While the Python Secrets RNG had the shortest runtimes, it also exhibited the highest number of failed tests, suggesting that shorter runtimes might reflect less thoroughness in randomness. Conversely, the Wolfram RNG, despite its longer runtimes, demonstrated fewer combined failed and suspicious tests, indicating better overall performance. Therefore, the runtime data adds another layer to our understanding but does not solely determine the best performer. The balance between runtime and the quality of randomness must be considered when determining this conclusion. The raw data can be viewed in Appendix A.3.

7.5 Selected PRNGs vs. IDQ Quantis

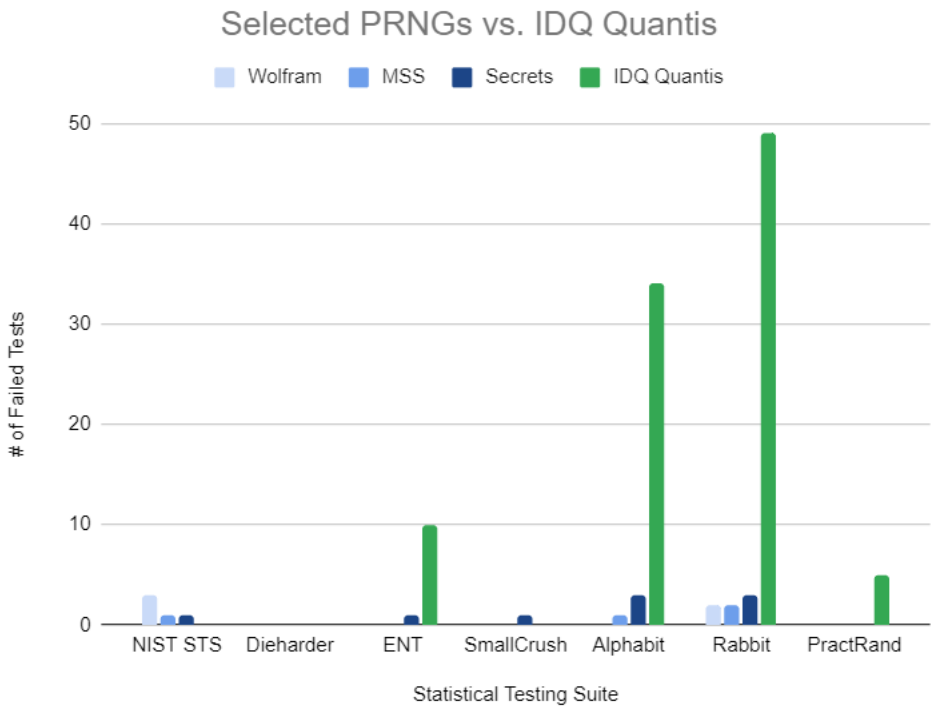

Figure 7.5: Column chart depicting the total number of failed tests over ten samples for our three selected PRNGs and IDQ Quantis RNG. This data was gathered from [4] and serves vital incite on how well our selected PRNGs perform when compared to this QRNG.

Figure 7.5 presents the total number of failed tests across the statistical testing suites for each PRNG, now including the data for IDQ Quantis RNG as reported in the research by Quantinuum [4]. Two significant observations can be made regarding their comparative performance. First, when focusing solely on the standardized NIST STS, the IDQ Quantis RNG demonstrates superior performance, with zero failed tests across ten samples. However, when considering the non-standardized testing suites, the IDQ Quantis RNG shows markedly poorer performance, with ten failed tests in ENT, thirty-four in Alphabit, forty-nine in Rabbit, and five in PractRand, culminating in a total of ninety-eight failed tests. Thus, although the IDQ Quantis RNG excels in the standardized NIST STS, it performs the worst overall when evaluated against the broader range of non-standardized testing suites.

7.6 Selected PRNGs vs. Intel RDSEED

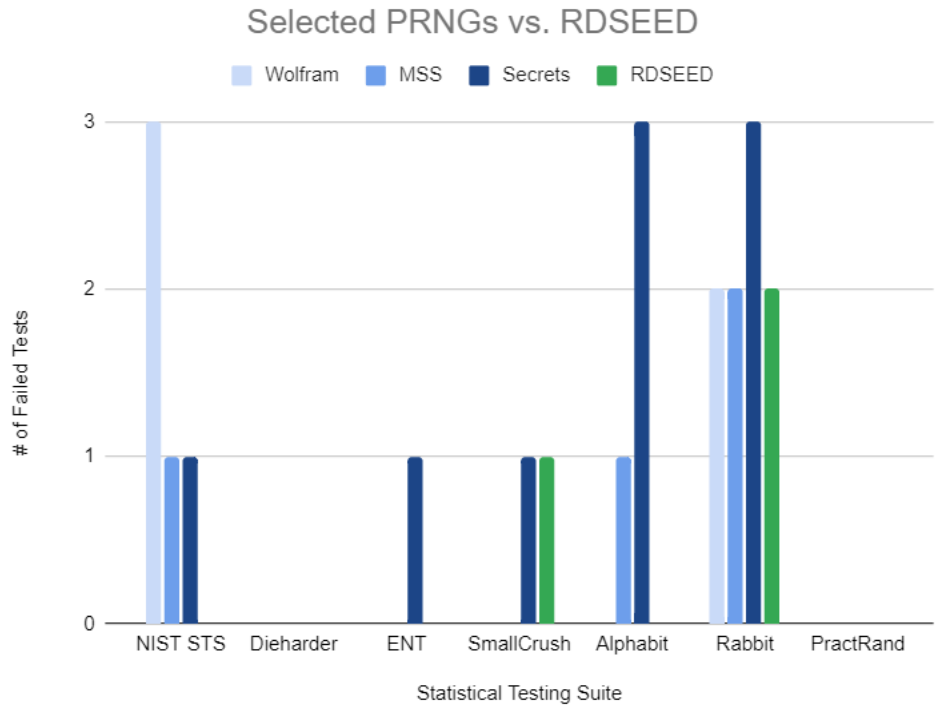

Figure 7.6: Column chart depicting the total number of failed tests over ten samples for our three selected PRNGs and Intel RDSEED. This data was gathered from [4] and serves vital incite on how well our selected PRNGs perform when compared to this QRNG.

Figure 7.6 presents the total number of failed tests across the statistical testing suites for each PRNG, now including the data for the QRNG RDSEED provided in the research by Quantinuum [4]. Two significant observations can be made regarding their comparative performance. First, when focusing solely on the standardized NIST STS, RDSEED demonstrates exceptional performance, with zero failed tests across ten samples. Additionally, in the non-standardized testing suites, RDSEED maintains its high performance, failing only one test in SmallCrush and two in Rabbit, for a total of three failed tests. Therefore, RDSEED not only excels in the standardized NIST STS but also performs robustly across the broader range of non-standardized testing suites, indicating its overall strong reliability and effectiveness as a QRNG.

When considering which PRNGs contend with this high-performing QRNG, the Microsoft Sparse Simulator RNG and the Wolfram RNG are notable. The Microsoft Sparse Simulator RNG, with a total of four failed tests, and the Wolfram RNG, with a total of five failed tests, both demonstrate strong performance, though not quite at the level of RDSEED. Thus, while these PRNGs show competitive results, RDSEED's performance stands out as the best among the RNGs evaluated.

Chapter 8

Conclusion and Future Work

This thesis has presented a few notable contributions to the advancements in statistical testing of randomly generated binary data. With the development of Cryptoguard, a consolidated statistical testing tool used to increase efficiency and testing coverage of randomly generated binary data, we have shown that broader statistical testing can be very beneficial in granting more insight into just how random arbitrary binary data is. Cryptoguard can be viewed at `https://github.com/jnaizer/cryptoguard`.

Our comprehensive analysis of our selected PRNGs and existing testing data on QRNGs from [4]'s publication, across a range of standardized and non-standardized statistical testing suites, revealed significant findings. The Microsoft Sparse Simulator RNG and the Wolfram RNG demonstrated robust performance with relatively low numbers of failed tests, indicating strong reliability and effectiveness in generating random sequences. However, when performing comparative analysis on the RDSEED QRNG with our PRNGs, RDSEED emerged as the standout performer, excelling in both standardized (NIST STS) and non-standardized testing suites with an exceptionally low total of three failed tests. This highlights RDSEED's superior capability in generating high-quality random numbers, making it a compelling choice for applications requiring stringent randomness criteria. All thesis statistical results can be viewed at `https://github.com/jnaizer/thesis_research_results`. Future work will focus on several key areas to build upon the development of Cryptoguard and findings of this thesis:

- Customizable NIST STS Input: Adding a new feature within Cryptoguard that allows the user to customize which options within the NIST STS are pre-selected. This would increase the flexibility and number of user-applications for the standardized testing suite within Cryptoguard.
- Implementation of Additional Testing Suites: Expand the scope of testing by incorporating more statistical suites into Cryptoguard, thereby enhancing the understanding of arbitrary RNGs' performance characteristics.
- Broader Statistical Testing: Apply a wider range of statistical testing suites to our selected RNGs to conduct a more comprehensive analysis of their performance across other testing suites.
- Extended Comparative Analysis: Conduct comparative analyses of the selected RNGs against other contemporary PRNGs and QRNGs to further benchmark their performances and identify potential areas for improvement.

Appendix A

Random Number Generator Analysis

This chapter displays all of the collected raw data from each of our selected random number generators when tested with all of our selected statistical testing suites.

A.1 Random Number Generator Generation Times

All raw generation times were collected within our data generation environment via implemented timers within each RNG data generation code.

RNG	Secrets	Wolfram	MSS
Sample 1	3h46m40s	2h30m53s	23h14m27s
Sample 2	3h36m22s	2h36m3s	24h05m18s
Sample 3	3h45m15s	2h50m40s	24h42m59s
Sample 4	3h52m04s	2h30m15s	23h53m12s
Sample 5	3h47m55s	2h51m13s	24h21m36s
Sample 6	3h41m19s	2h29m54s	23h29m04s
Sample 7	3h31m53s	2h41m01s	24h17m08s
Sample 8	3h34m27s	2h38m20s	24h56m45s
Sample 9	3h40m11s	2h43m34s	23h45m21s
Sample 10	3h48m58	2h49m12s	23h02m50s
Average	**3h42m48s**	**2h39m58s**	**24h01m02s**

Table A.1: Random Number Generator Generation Times for Secrets RNG, Wolfram RNG, and Microsoft Sparse Simulator RNG over their respective 10 samples each. The time generation data is structured in hours (h) minutes (m) seconds (s).

A.2 Statistical Testing Suite Results

Provided below is the raw data displaying the number of failed and suspicious tests for each of the ten samples for each of the three random number generators. Each statistical testing suite is labelled with its corresponding total number of total statistical tests it ran. For example, the ENT statistical testing suite executed a total of 6 statistical tests on each sample, denoted by the ”(6)”. The number of failed tests for each testing suite are then totalled at the bottom, signifying the total number of failed tests for that specific testing suite across the specific RNG samples.

RNG	NIST (15)	Dieharder (18)	ENT (6)	SmallCrush (15)	Crush (144)	Alphabit (17)	Rabbit (40)	PractRand (920)
Secrets 1	0	0 (1)	0	0	6	1	0	0 (0)
Secrets 2	1	0 (1)	0	0	8	0	0	0 (1)
Secrets 3	0	0 (0)	0	0	5	0	0	0 (0)
Secrets 4	0	0 (1)	1	1	9	0	1	0 (0)
Secrets 5	0	0 (0)	0	0	8	2	1	0 (0)
Secrets 6	0	0 (2)	0	0	7	0	0	0 (2)
Secrets 7	0	0 (0)	0	0	6	0	0	0 (0)
Secrets 8	0	0 (1)	0	0	7	0	1	0 (0)
Secrets 9	0	0 (0)	0	0	7	0	0	0 (0)
Secrets 10	0	0 (0)	0	0	7	0	0	0 (0)
Total	**1**	**0 (6)**	**1**	**1**	**70**	**3**	**3**	**0 (3)**

Table A.2: Total number of failed tests for each of the ten samples for the Python Secrets RNG. For the testing suites that offer additional metrics specifying any ”suspicious” tests (Dieharder and PractRand), the total ”suspicious” tests for each sample are denoted by parenthesis.

RNG	NIST (15)	Dieharder (18)	ENT (6)	SmallCrush (15)	Crush (144)	Alphabit (17)	Rabbit (40)	PractRand (920)
Wolfram 1	0	0 (0)	0	0	18	0	1	0 (0)
Wolfram 2	0	0 (0)	0	0	7	0	0	0 (1)
Wolfram 3	0	0 (0)	0	0	6	0	0	0 (0)
Wolfram 4	1	0 (1)	0	0	7	0	0	0 (0)
Wolfram 5	0	0 (1)	0	0	7	0	0	0 (1)
Wolfram 6	0	0 (0)	0	0	6	0	0	0 (1)
Wolfram 7	0	0 (0)	0	0	8	0	0	0 (0)
Wolfram 8	0	0 (0)	0	0	6	0	0	0 (0)
Wolfram 9	1	0 (2)	0	0	7	0	1	0 (0)
Wolfram 10	1	0 (0)	0	0	6	0	0	0 (0)
Total	**3**	**0 (4)**	**0**	**0**	**78**	**0**	**2**	**0 (3)**

Table A.3: Total number of failed tests for each of the ten samples for the Wolfram RNG. For the testing suites that offer additional metrics specifying any ”suspicious” tests (Dieharder and PractRand), the total ”suspicious” tests for each sample are denoted by parenthesis.

RNG	NIST (15)	Dieharder (18)	ENT (6)	SmallCrush (15)	Crush (144)	Alphabit (17)	Rabbit (40)	PractRand (920)
MSS 1	0	0 (2)	0	0	6	1	0	0 (0)
MSS 2	0	0 (0)	0	0	8	0	0	0 (0)
MSS 3	0	0 (0)	0	0	8	0	1	0 (1)
MSS 4	1	0 (1)	0	0	9	0	0	0 (0)
MSS 5	0	0 (0)	0	0	5	0	0	0 (0)
MSS 6	0	0 (0)	0	0	8	0	1	0 (1)
MSS 7	0	0 (1)	0	0	7	0	0	0 (1)
MSS 8	0	0 (2)	0	0	7	0	0	0 (1)
MSS 9	0	0 (0)	0	0	7	0	0	0 (1)
MSS 10	0	0 (2)	0	0	6	0	0	0 (0)
Total	**1**	**0 (8)**	**0**	**0**	**71**	**1**	**2**	**0 (5)**

Table A.4: Total number of failed tests for each of the ten samples for the Microsoft Sparse Simulator RNG. For the testing suites that offer additional metrics specifying any ”suspicious” tests (Dieharder and PractRand), the total ”suspicious” tests for each sample are denoted by parenthesis.

A.3 Statistical Testing Suite Runtimes

All raw statistical testing times were collected within our statistical testing environment via implemented timers within each statistical testing suite script.

RNG	NIST	Dieharder	ENT	Small Crush	Crush	Alphabit	Rabbit	PractRand
Secrets 1	22m20s	15m29s	1m47s	0m34s	118m44s	9m15s	38m28s	2m47s
Secrets 2	31m49s	8m37s	1m35s	0m34s	121m43s	9m35s	45m36s	3m0s
Secrets 3	24m7s	7m17s	2m34s	0m48s	153m14s	10m31s	45m56s	4m19s
Secrets 4	23m56s	16m29s	1m44s	0m37s	128m45s	9m37s	43m9s	3m13s
Secrets 5	24m38s	16m51s	1m41s	0m37s	130m38s	10m10s	44m35s	3m1s
Secrets 6	25m2s	17m24s	1m48s	0m37s	134m6s	10m7s	43m41s	3m18s
Secrets 7	24m15s	17m33s	1m57s	0m40s	140m19s	11m0s	48m52s	3m21s
Secrets 8	24m19s	17m25s	1m54s	0m38s	141m1s	9m44s	44m16s	3m19s
Secrets 9	24m15s	6m28s	1m48s	0m34s	133m0s	9m58s	44m4s	3m14s
Secrets 10	22m20s	6m16s	1m57s	0m38s	117m58s	9m14s	38m54s	2m47s
Average	**24m42s**	**12m58s**	**1m52s**	**0m38s**	**131m53s**	**9m55s**	**43m45s**	**3m14s**

Table A.5: Statistical Testing Suite Runtimes for the Secrets RNG. The testing suite runtimes are structured in minutes (m) seconds (s).

RNG	NIST	Dieharder	ENT	Small Crush	Crush	Alphabit	Rabbit	PractRand
Wolfram 1	35m5s	23m22s	1m33s	0m37s	147m22s	13m15s	53m14s	3m30s
Wolfram 2	33m19s	21m56s	2m38s	0m54s	168m10s	13m30s	54m9s	4m19s
Wolfram 3	33m50s	22m33s	2m28s	0m54s	168m57s	13m52s	56m13s	4m11s
Wolfram 4	27m3s	18m40s	2m31s	0m51s	168m55s	13m56s	53m20s	4m13s
Wolfram 5	22m1s	15m30s	1m52s	0m35s	120m38s	8m55s	38m11s	2m55s
Wolfram 6	22m6s	6m6s	1m43s	0m35s	119m53s	8m58s	38m28s	2m49s
Wolfram 7	22m8s	15m30s	1m36s	0m35s	120m8s	8m59s	38m56s	2m46s
Wolfram 8	30m39s	8m2s	2m9s	0m43s	161m32s	13m23s	52m48s	3m32s
Wolfram 9	36m6s	9m51s	2m32s	0m45s	159m10s	13m2s	53m17s	3m55s
Wolfram 10	23m6s	15m59s	1m37s	0m38s	125m17s	9m34s	40m26s	3m19s
Average	**28m42s**	**15m44s**	**2m3s**	**0m42s**	**145m54s**	**11m4s**	**51m54s**	**3m32s**

Table A.6: Statistical Testing Suite Runtimes for the Wolfram RNG. The time generation data is structured in hours (h) minutes (m) seconds (s).

RNG	NIST	Dieharder	ENT	Small Crush	Crush	Alphabit	Rabbit	PractRand
MSS 1	28m14s	23m48s	1m33s	0m35s	142m1s	12m30s	50m24s	2m53s
MSS 2	27m28s	15m51s	2m42s	0m34s	118m49s	9m4s	40m9s	2m47s
MSS 3	21m52s	15m43s	1m46s	0m35s	121m7s	9m9s	39m21s	2m56s
MSS 4	21m22s	15m38s	1m51s	0m35s	122m32s	9m16s	38m50s	2m49s
MSS 5	21m7s	18m44s	1m47s	0m35s	123m38s	9m6s	51m53s	2m49s
MSS 6	21m22s	16m0s	2m56s	0m43s	121m56s	9m8s	39m28s	4m11s
MSS 7	21m51s	15m28s	2m2s	0m33s	116m40s	8m45s	38m14s	2m40s
MSS 8	30m16s	18m57s	1m41s	0m43s	154m49s	12m45s	50m26s	4m0s
MSS 9	30m32s	20m31s	2m36s	0m45s	162m5s	14m0s	53m20s	4m9s
MSS 10	29m25s	7m56s	1m29s	0m32s	140m43s	12m43s	51m18s	2m38s
Average	**26m59s**	**16m51s**	**2m1s**	**0m37s**	**134m17s**	**10m48s**	**45m30s**	**3m15s**

Table A.7: Statistical Testing Suite Runtimes for the Microsoft Sparse Simulator RNG. The time generation data is structured in hours (h) minutes (m) seconds (s).

References

[1] L E Bassham, A L Rukhin, J Soto, J R Nechvatal, M E Smid, E B Barker, S D Leigh, M Levenson, M Vangel, D L Banks, and et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, 2010.

[2] Riccardo Cantoro, Nikolaos I. Deligiannis, Matteo Sonza Reorda, Marcello Traiola, and Emanuele Valea. Evaluating the code encryption effects on memory fault resilience. *2020 IEEE Latin-American Test Symposium (LATS)*, Mar 2020.

[3] Miguel Herrero-Collantes and Juan Carlos Garcia-Escartin. Quantum random number generators. *Reviews of Modern Physics*, 89(1), 2017.

[4] Cameron Foreman, Richie Yeung, and Florian J Curchod. Statistical testing of random number generators and their improvement using randomness extraction. *arXiv preprint arXiv:2403.18716*, 2024.

[5] Michael Tabb, Jeffery DelViscio, and Andrea Gawrylewski. What is 'the cloud' and how does it pervade our lives?, Dec 2021.

[6] Timothy Hollebeek. How to secure quantum computing in the cloud, Nov 2020.

[7] Aryya Paul. An introduction to cryptographic algorithms, Aug 2018.

[8] Itan Barmes. The quantum threat to cyber security, Sep 2020.

[9] Gui Alvarenga. What is the shared responsibility model? *CrowdStrike*, Nov 2022.

[10] Gartner. Gartner forecasts worldwide public cloud end-user spending to reach nearly $600 billion in 2023. *Gartner*, Oct 2022.

[11] Logan McCoy. Microsoft azure explained: What it is and why it matters, Jan 2023.

[12] Microsoft. Azure quantum - quantum cloud computing service: Microsoft azure, 2024.

[13] Microsoft. Virtual machines (vms) for linux and windows: Microsoft azure, 2023.

[14] Rupal Purohit. Microsoft azure storage account: A complete overview, Jun 2023.

[15] Mahesh Chand. What is azure functions: A beginner's tutorial, Jan 2023.

[16] Anthony Annunziata, Jerry Chow, Jay Gambetta, and Joe Raffa. Coming soon to your business: Quantum computing. *IBM*, 2018.

[17] Carolyn Mathas. The basics of quantum computing. *EDN*, Aug 2019.

[18] IBM Quantum, 2021.

[19] Brett Daniel. Symmetric vs. asymmetric encryption: What's the difference? *Trenton Systems*, May 2021.

[20] Cremarc. The importance of randomness in a quantum world, Apr 2022.

[21] Sudip Sengupta. Stream cipher vs. block cipher, Feb 2022.

[22] Shawn Wang. The difference in five modes in the aes encryption algorithm, Aug 2019.

[23] Subhasish Sarkar, 2020.

[24] Casey Crane. What is a message authentication code (mac)?, Feb 2023.

[25] John Carl Villanueva. What is hmac and how does it secure file transfers?, Dec 2022.

[26] Bill Buchanan. I know hmac, but what's cmac?, Oct 2021.

[27] Neeru Mago. Pmac: A fully parallelizable mac algorithm. *Apeejay Journal of Computer Science and Applications*, 3:36–44, Jan 2015.

[28] BrainKart. Authenticated encryption: Ccm and gcm.

[29] M J Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. *Recommendation for block cipher modes of operation:*, 2007.

[30] Suhas Hegde. Cloud cryptography: A reliable solution to secure your cloud. *Analytics Vidhya*, Oct 2022.

[31] Cloud Security Alliance. Cloud key management, 2023.

[32] Cloudflare. What is multitenancy? — multitenant architecture, 2023.

[33] Mehdi Ebady Manaa and Zuhair Gheni Hadi. Scalable and robust cryptography approach using cloud computing. *Journal of Discrete Mathematical Sciences and Cryptography*, 23(7):1439–1445, 2020.

[34] Samuel Jaques and Thomas Häner. Leveraging state sparsity for more efficient quantum simulations. *ACM Transactions on Quantum Computing*, 3(3):1–17, Jun 2022.

[35] Microsoft. Introduction to q# & quantum development kit - azure quantum, 2024.

[36] Wolfram. Wolfram cloud: Integrated computation, knowledge, deployment, 2024.

[37] California Institute of Technology. How will quantum technologies change cryptography? *Caltech Science Exchange*, 2023.

[38] Marek Sýs and Zdeněk Říha. Security, privacy, and applied cryptography engineering. *Faster Randomness Testing with the NIST Statistical Test Suite*, 2014.

[39] Darren Hurley-Smith and Julio Hernandez-Castro. Certifiably biased: An in-depth analysis of a common criteria eal4+ certified trng. *IEEE Transactions on Information Forensics and Security*, 13(4):1031–1041, 2018.

[40] Chris Doty-Humphrey. Practrand (practically random), 2016.

[41] Pierre L'Ecuyer and Richard Simard. Testu01. *ACM Transactions on Mathematical Software*, 33(4):1–40, Aug 2007.

[42] Robert G Brown. Dieharder: A random number test suite, 2024.

[43] Leilei Huang, Hongyi Zhou, Kai Feng, and Chongjin Xie. Quantum random number cloud platform. *npj Quantum Information*, 7(1), 2021.

[44] Marcin M. Jacak, Piotr Jóźwiak, Jakub Niemczuk, and Janusz E. Jacak. Quantum generators of random numbers. *Scientific Reports*, 11(1), Aug 2021.

[45] John Walker. Ent, Jan 2008.

[46] Lara Ortiz-Martin, Pablo Picazo-Sanchez, Pedro Peris-Lopez, and Juan Tapiador. Heartbeats do not make good pseudo-random number generators: An analysis of the randomness of inter-pulse intervals. *Entropy*, 20(2):94, Jan 2018.

[47] Meng Xiannong. Gap test, Oct 2002.

[48] Python Software Foundation. Secrets - generate secure random numbers for managing secrets, 2024.

[49] Swathi Arun. The secrets module of python, Sep 2021.

[50] Microsoft. Azure Quantum Development Kit.

[51] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018. ACM, February 2018.