Improving Classical Cryptography with Quantum Random Number Generation

Dillon J. Newell [1] – 1000407641

[1] *Department of Electrical and Computer Engineering, University of Toronto, Ontario, Canada*

1. Introduction

In the modern world, random number generation is an important subject for a variety of different applications. These can be considered with two categories. Non-critical applications are scenarios where security is not critical and can include computer games, simulations, and statistic sampling. Critical applications are scenarios where security is fundamental to the user and include gambling and cryptography. For this paper we will focus on random number generation as it applies to classical cryptography, and specifically, encryption key generation.

Encryption is a fundamental tool for online communication. Every day, millions of people rely on encryption to secure their online experience. Important consumer information such as passwords and banking information is secured through encryption to prevent information gain by attackers. While encryption algorithms themselves can be vulnerable to attacks, a more practical attack vector is obtaining the keys that these algorithms rely on. In general, obtaining the encryption key allows an attacker to completely bypass the encryption algorithm.

This paper is divided into the following sections:

2. The relevance of random number generation as it pertains to classical encryption schemes
3. Exploration of the current methods used for random number generation
4. Exploration of random number generation through quantum effects
5. Comparison of classical and quantum random number generation
6. Exploration of advantages that quantum random number generation can provide to classical encryption schemes over current methods
7. Conclusion

2. Random number generation in classical cryptography

2.1. Classical encryption schemes

In popular use, encryption schemes generally fall into one of two categories: symmetric and asymmetric key encryption. Both rely on high entropy random number generation for secure use.

2.1.1. Symmetric key encryption

Symmetric key encryption is any scheme where the same key is used to both encrypt and decrypt the data. This requires that both parties share a private encryption key beforehand; usually this is accomplished through a handshake phase.[a]

In symmetric key encryption it is paramount that the key remains secret. This of course implies that it must not be guessable. Therefore, predetermined secrets such as passwords are not overly secure keys. The least guessable key would be one that is selected completely at random. A truly random key (provided it is shared securely) is only guessable by an attacker through brute force.[b]

The key length is also an important factor to the security of the protocol. Any protocol can be brute forced (trying each possible key), so if there are only a few possible values for the key, it can easily be found. A key length that is as long as the data to be encrypted provides a much greater amount of security. *One-time pad* is an encryption protocol that requires a key length at least as long as the data and is the only known provably secure classical encryption protocol.[1],[c]

A widely used symmetric key encryption protocol is *Advanced Encryption Standard* (AES). AES was standardized by NIST and uses key lengths of 128, 192, or 256 bits.[2]

2.1.2. Asymmetric key encryption

In contrast to symmetric key encryption, asymmetric encryption uses two keys: one public and one private. The public key can encrypt data, but only the private key can decrypt. The namesake of the keys results from the public key being safe to distribute, as no one can read any encrypted data without the private key.

A widely used asymmetric encryption protocol is *Rivest-Shamir-Adleman* (RSA).[3] In RSA, keys are generated using the product of two very large prime numbers. The public key contains information about the product, but not the two primes themselves. If the primes are guessed, then the private key can be ascertained, and the protocol broken. The idea is that factorizing the product into its two primes is computationally infeasible.[d]

It can then be seen that asymmetric protocols like RSA also rely on numbers that are not easily guessable. In this case, guessing the two primes used in key generation would allow an

---

[a] While Quantum Key Distribution provides an excellent application to securely share encryption keys, we will only be considering classical protocols.

[b] In cryptography, it is generally considered that a protocol is broken only if there is a method to crack it faster than brute force. Therefore, a protocol that is secure up to brute forcing is considered "gold standard".

[c] The *one-time pad* is provably secure if and only if the key is a) truly random, b) at least as long as the data, c) never reused, and d) kept secret.

[d] In an example of quantum code breaking, Shor's algorithm has been predicted to factor prime products in polynomial time with a sufficient quantum computer, breaking RSA. Again, for this paper we will not be considering quantum effects outside of random number generation.

attacker to obtain the private key. Therefore, it is important that the primes used are selected at random, without any biases or patterns an attacker could detect.

2.2. Case study: Transport Layer Security (TLS)

TLS [4] is also known by its former name *Secure Sockets Layer* (SSL)[e] or through its most popular application *Hypertext Transfer Protocol Secure* (HTTPS). It is a widely used protocol for securing communications over a computer network. As of April 10th 2019, roughly 87.9% of websites are secured using either TLS or SSL.[5],[f]

TLS makes use of both symmetric and asymmetric encryption to provide secure browsing experiences for consumers. It is responsible for securing personal information such as passwords and banking information. The protocol contains two high-level phases: the handshake and the application.

2.2.1 TLS handshake

The handshake phase accomplishes several goals:

1. Deciding on which version of TLS to use based on client and server capabilities
2. Proving the identity of the server to the client [g]
3. Optionally proving the identity of the client to the server
4. Deciding on the encryption protocol to use for the application phase
5. Generating a secret key for the application phase

The handshake makes use of asymmetric encryption. Here, the server first sends a certificate to the client to prove its identity. This certificate contains pertinent information about the server, including domain name and the public key to use for encryption. The client must validate the certificate. It does this by verifying several parameters:

- The certificate is within its validity period (i.e., it has not yet expired)
- The certificate was issued to the domain that the client is trying to access
- The certificate was digitally signed by another trusted certificate

Here, the last point is of particular significance. Operating systems and web browsers come with a list of trusted *root certificates*. These certificates are from third-parties that the client implicitly trusts and include a public key that can be used to verify other certificates. When

---

[e] Technically, TLS is the successor to SSL, although it is also correct to view TLS simply as version increment of SSL. Still, TLS and SSL can be distinguished, with the context that using SSL is using an older version of TLS. (All versions of SSL are now deprecated).

[f] Roughly a quarter of these websites use invalid configurations of TLS and are either not secure or only partially secure.

[g] As with proving client identity to the server, this is also an optional component of the handshake. However, it is used for almost all websites secured with TLS.

a server sends its domain certificate, the client checks its trusted certificate list for one that has signed the domain certificate and validates the signature with the public key.

If an attacker is able to ascertain the private key of any implicitly trusted root certificate, it can then create certificates with arbitrary information that appear to be valid to the client. As an example, consider an attacker who has compromised a public WiFi network, such as one found at a coffee shop. They could then set up a website that imitates, e.g., a bank website, and create their own certificate for the domain. When the client accesses their bank's website, they will be served with a fraudulent certificate and website that appear valid. Despite showing as secure in their browser, any information sent by the client (e.g. login details) will be sent directly to the attacker.

Furthermore, if the attacker ascertains the private keys belonging to the legitimate website owner, they need not even make their own certificate. They can simply serve the original valid certificate and decrypt the client's responses.

In any case, after the client has validated the server's identity it will generate a *Pre-Master Secret*. This key is generated at random and used to generate the *Master Secret*, from which all following communication will be symmetrically encrypted with. The *Pre-Master Secret* is transmitted to the server after encrypting it with the server's public key, which can only be decrypted by someone holding the server's private keys.

2.2.2 Application phase

Once the handshake is complete, all further information is encrypted with the symmetric protocol decided during the handshake. In an uncompromised session, only the server and the client know the Master Secret that is used as the key for symmetric encryption.

The reason symmetric encryption is used for this phase is two-fold. First, symmetric encryption is typically much faster than asymmetric encryption, and website load times are reduced. Second, asymmetric encryption for this phase would mean the server has no way to send encrypted data to the client without a client public key. Therefore either the client would need its own valid certificate, or only data transmitted by the client would be encrypted.

Overall, TLS requires that both the server and the client have a way to efficiently generate random numbers: the server for its private key creation, and the client for Pre-Master Secret generation. These keys may be long-lived as well, e.g. domain certificates are typically valid for a year or more and root certificates may be valid for up to decades. If there is not sufficient entropy in the key generation, then an attacker could detect patterns and break TLS sessions. Furthermore, there are several points of weakness that can be used as attack vectors, since certificate chains typically include at least 3 certificates. Only one certificate needs to be compromised for an attacker to break TLS.

3. Classical random number generation

Now that we have perspective on how important random numbers are in classical cryptography, we can discuss existing methods used for random number generation.

Classical random number generators (RNGs) typically fall into one of two categories: pseudo-random number generators (PRNGs) and hardware random number generators (HRNGs)[h]. Each has their own advantages and drawbacks.

3.1. Pseudo-random number generators (software)

PRNGs rely on arithmetic to simulate random number generation. Since software is inherently deterministic, there is no way to provide true randomness from an arithmetical model. However, PRNGs can use algorithms to generate sequences of numbers that *appear* random.

In general, a PRNG can be completely described by its current state. On initialization a state is chosen, typically through a "seed" value. It is desirable for this seed initialization to be completely random.[i] The seed determines the initial state, and the number of possible states is known as the "seed space". This has two important implications:

- If an attacker can ascertain the PRNG algorithm and current state, they are able to determine all future values produced by the PRNG.
- A PRNG has only a finite number of states in which it can be – typically on the order of 2^{32}. Inevitably, the PRNG will reach a state that will transition to a previously occupied state. This means all PRNGs will eventually cycle their output.

PRNGs cannot circumvent these implications on their own. However, a common approach is to periodically "re-seed" the PRNG. This is typically seen in OS-level PRNGs, where data such as current cursor position and system uptime are used as entropy for re-seeding (see **Figure 1** for an example from the Windows XP PRNG).

At the core of any PRNG is the *transition function*. During the transition function, the state transitions to another and a pseudo-random number is produced. Transition functions may or may not be reversible. With a reversible transition function, an attacker ascertaining the current state may also be able to obtain all previous values produced by the PRNG.

Most PRNGs are not suitable for use with cryptography. This includes the default implementation in many programming languages, e.g. C#/.NET's *System.Random* [7] object. This is usually stressed in language documentation, but that does not prevent their use in cryptography by careless programmers.

---

[h] Also known as "true" random number generators or TRNGs.

[i] In some specialized applications, the ability to reproduce the random sequence can be desirable. For these applications (typically testing or simulation), being able to seed with a constant value is beneficial.

A subset of PRNGs known as cryptographically-secure PRNGs (CSPRNGs) have extra requirements to make them suitable for use in cryptography:

- Given the past outputs of the CSPRNG, an attacker must not be able to determine the next output.
- If an attacker ascertains the CSPRNG state, she must not be able to determine past outputs – known as *forward security*. This implies the transition function is irreversible.
- If an attacker ascertains the CSPRNG state, she must not be able to determine future outputs past a certain point – known as *backward security*. This implies the CSPRNG is periodically re-seeded with external entropy.

Most operating systems implement a CSPRNG, e.g. Windows's *CryptGenRandom* function [8] or Linux's */dev/random* [9] pseudo-device. Programming languages may also offer CSPRNG implementations.[j] CSPRNGs are currently the most popular method of generating random numbers for use in cryptography.

Source	Bytes requested
CircularHash	256
KSecDD	256
GetCurrentProcessID()	8
GetCurrentThreadID()	8
GetTickCount()	8
GetLocalTime()	16
QueryPerformanceCounter()	24
GlobalMemoryStatus()	16
GetDiskFreeSpace()	40
GetComputerName()	16
GetUserName()	257
GetCursorPos()	8
GetMessageTime()	16
NTQuerySystemInformation calls	
ProcessorTimes	48
Performance	312
Exception	16
Lookaside	32
ProcessorStatistics	up to the remaining length
ProcessesAndThreads	up to the remaining length

Figure 1: Entropy values used in seeding the Windows XP CSPRNG. Some values follow clear patterns, such as local time. Others may not change at all, such as user name. Obtained through reverse-engineering in [6].

Overall, PRNGs (and CSPRNGs) are very fast and easy to consume. We have seen that they have several downsides however:

- PRNGs can only emulate entropy, not create it

[j] Many language CSPRNGs simply forward to the OS-level CSPRNG, e.g. C#/.NET's *System.Security.Cryptography.RandomNumberGeneratorImplementation.* [10]

- A high entropy seed is needed for initialization, not provided by the PRNG
- All PRNGs will eventually cycle if they are not re-seeded
- PRNG implementations are subject to bugs

In 1951 John von Neumann said, “Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” While meant as a joke, his point still stands: software alone is not capable of truly producing randomness.

a) b)

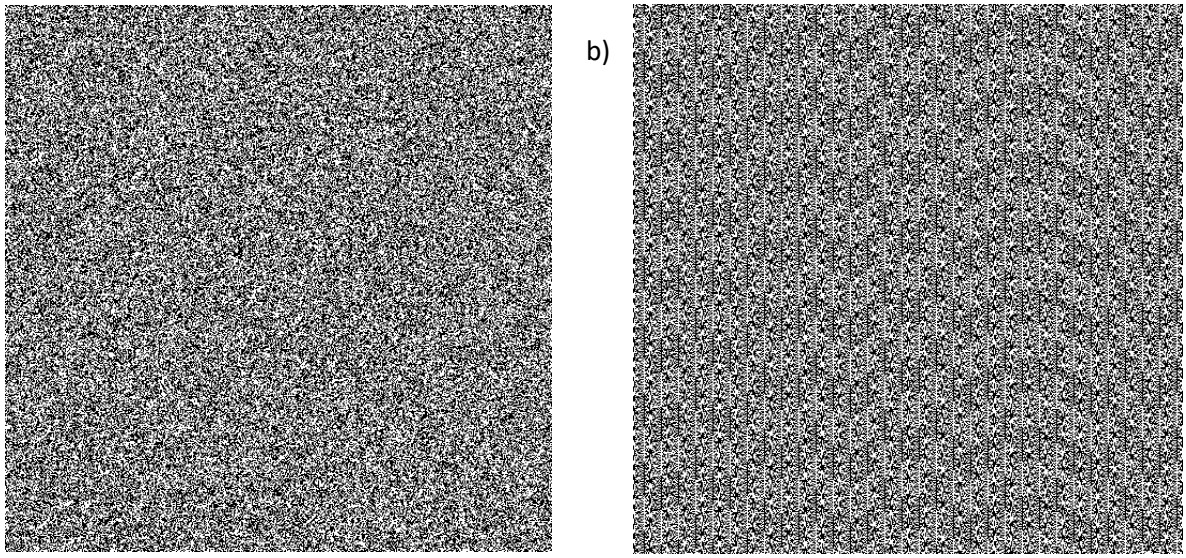

Figure 2: (a) a bitmap generated from atmospheric noise using Random.org, and (b) a bitmap generated from PHP’s rand() PRNG function. Clear patterns can be seen in the latter, while the former appears truly random. From [11].

3.2. Hardware random number generators

With the inherent faults of PRNGs, HRNGs aim to produce randomness without relying solely on arithmetical methods. They do this by leveraging physical phenomena considered to be truly random, such as:

- Thermal noise as measured by a resistor
- Atmospheric noise as measured by a radio receiver
- Clock drift in computational circuits

By using truly random sources, HRNGs can bypass many of the deficiencies in PRNGs such as forward and backward security and seed space. They do however come with their own deficiencies.

HRNGs, particularly those that measure noise, are subject to biases. These biases may come from biased noise produced by the device itself (e.g. from circuitry) or from an outside source (e.g. inside a computer, many predictable sources of noise exist). Furthermore, the resolution of entropy measurement is generally much less than the range of numbers that are desired to be produced. As such, HRNGs must collect entropy over time, making them very slow.

A particularly worrisome problem with HRNGs is that when they fail, they fail silently. Aging components in the source or measurement devices can produce significant bias. HRNGs typically get around this by employing a health check circuit verifying the randomness of produced numbers. However, it is very hard to quantify how random a sequence of numbers is.

Furthermore, HRNGs are commonly produced as "black boxes". This means the end-user must trust the manufacturer for quality and integrity. Manufacturers do this for two reasons. First, the methods used may be considered a trade secret. Second, manufacturers may be using *security through obscurity*. Security through obscurity is the paradigm that attackers are much less likely to be able to exploit the device if they do not know how it works. This idea is considered a fallacy by many security experts, and opponents believe open-source methods provide higher security since the implementation is subject to more scrutiny from the community. On the topic, NIST says "System security should not depend on the secrecy of the implementation or its components." [12]

Many implementations of HRNGs are in use today, some employed widely in computers that users may not be aware of. Some intentional and unintentional examples are:

3.2.1. Random.org

Random.org is a popular website that produces random numbers using atmospheric noise. It uses several stations deployed around the world to reduce local biases. [11] The website also provides APIs for developer use to leverage their generation in applications.

3.2.2. Arduino Uno

The Arduino Uno is a popular developer microcontroller board that contains several analog measurement pins. The default RNG is a simple PRNG, but the documentation states randomness can be improved by re-seeding the PRNG with the measurement of an unconnected analog pin.[13] The resulting measurement will depend on atmospheric noise, however it may be subject to extreme bias since it was not designed for this purpose.

3.2.3. Computer peripherals

OS-level CSPRNGs seed themselves with entropy gained from many metrics (see **Figure 1**). Some sources reflect entropy in computer components.[6] Percentage of active time from the central processor may be one example. Another may be the timing of disk reads/writes.

3.2.4. x86 RDRAND instruction

Starting with their Ivy Bridge range in 2012, Intel includes an HRNG circuit in every processor accessible through an x86 instruction extension.[14] It produces entropy through on-chip thermal noise. The entropy is then checked with a "health and swellness [*sic*]" circuit [15], before being used to seed a conventional PRNG. AMD introduced a matching circuit and instruction for their processors in 2015.

The RDRAND instruction is employed for seeding by many CSPRNGs, including the Linux */dev/random* pseudo-device. However, Intel keeps their exact specifications of the HRNG a trade secret, and it is impossible to gain access to the raw entropy production. This closed-source nature is contrary to Linux's open-source initiative and in 2013 the developer of */dev/random*, Theodore Ts'o, said "Relying solely on the hardware random number generator which is using an implementation sealed inside a chip which is impossible to audit is a BAD idea." As such, */dev/random* uses RDRAND only as part of its entropy collection.

Overall, HRNGs solve many problems with PRNGs but have drawbacks of their own:

- Entropy collection is typically very slow
- Black box implementations require trust by the end-user
- Entropy collection can be skewed by uncontrollable sources, including background noise
- Components are subject to aging and may fail silently overtime

3.3. Failures of classical RNGs

With a grasp on how classical RNGs work, we can discuss concrete examples of their failures. These range from poor development practices to simple mistakes to untrustworthy parties.

3.3.1. Netscape SSL (1996)

Netscape was a popular browser in the 90s, and its developers invented the SSL protocol we rely on today. However, their initial implementation was flawed.[16] The PRNG used for SSL was seeded from the entropy of only three values: time of day, process ID, and parent process ID. Time of day is a very predictable value, and the number of available process IDs is typically only on the order of 10^5 This led the seed to be easily guessable, and thus output of the PRNG could be predicted. Since Netscape was closed-source, it had to be reverse-engineered for the vulnerability to be discovered. It is unknown if attackers were able to find and exploit the vulnerability before it was made public and fixed.

3.3.2. Windows XP CSPRNG (2007)

More recently, researchers reverse-engineered the CSPRNG included in Windows XP and Windows Server 2000.[6] They found two problems. First, the transition function was easily reversible and thus the CSPRNG lacked forward security (technically making it an improper CSPRNG). Second, it re-seeded on long intervals, so ascertaining the state allowed many future values to be determined. These long intervals were not based on time, but on how many values have been produced. Furthermore, re-seeding was shared among all processes using the PRNG, with only a small amount of entropy being specific to the running process. Someone with access to the binaries that call the PRNG could figure out how the seeding would look on any other computer running the same binary. An example of a widely used binary calling this function is Internet Explorer, which used it for SSL communication.

Microsoft stated this problem was fixed with Windows Vista, however their implementation remains closed-source. The authors note that at the time of their research, Windows XP and its server counterpart were installed on 90% of computers worldwide.

3.3.3. Debian OpenSSL (2008)

OpenSSL is an open-source SSL/TLS implementation with significant popularity. It is the default TLS provider on most Linux distributions, and modified versions are integrated into web browsers such as Google Chrome. It provides not only SSL/TLS services, but also certificate signing.

In 2006, a developer made a one-line change to OpenSSL's CSPRNG that was flagged as redundant by development software. Unknowingly this change removed almost all the CSPRNG's entropy collection, resulting in a seed that only used the current process ID. The change was in an area specific to Debian releases, so only Debian-based operating systems such as Ubuntu were affected. The bug was not noticed until 2008.[17]

On Linux there are only around 35,000 possible process IDs. As such, brute forcing the CSPRNG would be trivial with hardware available at the time. Debian and Ubuntu are popular operating systems for web servers, and after the bug was discovered and fixed, all generated certificates from the past two years had to be revoked.

3.3.4. Concerns of government backdoor (2013)

Dual_EC_DRBG is a PRNG that relies on a number of constants for its core algorithm. In 2007, employees from Microsoft showed that with careful selection of these constants a backdoor could be inserted into the algorithm, effectively allowing the inserting party to access information about numbers generated.

In 2013 Edward Snowden released many documents containing highly classified information from the U.S. National Security Agency (NSA). These documents suggest that not only did the NSA create a backdoor for the *Dual_EC_DRBG* algorithm [18], but they had paid RSA Security[k] $10,000,000 to make it the default algorithm in their software [19].

The Snowden Documents also suggested many more backdoors into popular encryption products by the NSA. Among them were suggestions that the NSA worked with HRNG manufacturers to implement backdoors into physical chips [20], raising further concerns about black-box HRNGs.[l]

4. Quantum random number generation

Quantum random number generators (QRNGs) can be viewed as a specific type of HRNG. In fact, many classical HRNGs utilize noise that is described by quantum effects, except on a

[k] RSA Security was founded by the same inventors of the RSA encryption protocol, hence the same name. They produce a variety of encryption products.

[l] This was the context for Theodore Ts'o's earlier quote on not trusting sealed HRNGs.

macroscopic level. QRNGs take advantage of microscopic quantum effects that are considered fundamentally random, such as:

- Nuclear decay
- Optics (photon beam-splitting/phase fluctuation)
- Shot noise

4.1. On the randomness of quantum effects

Before we consider using quantum effects for random number generation, we must consider how truly random these effects are.

When quantum mechanics (QM) was young, many physicists were not convinced of the randomness of quantum effects. They believed that the universe is fundamentally deterministic, and any results that appeared random simply arose from variables encoded in quantum states that QM failed to account for. These so-called "hidden local variables" led opponents to believe current QM interpretations were incomplete. Perhaps the most prominent among these physicists was Albert Einstein, who famously said on the matter in 1926 "I am convinced God does not play dice."

Einstein's opposition to the probabilistic nature of QM interpretations culminated when he, along with Boris Podolsky and Nathan Rosen, released the EPR paper in 1935.[21] The EPR paper formulated a paradox that, in essence, said QM allowed two particles to interact in such a way that either:

- It is possible to measure their position and momentum simultaneously (violating Heisenberg's Uncertainty Principle), or
- Measurement of one particle affected the other at superluminal speeds (violating the theory of relativity)

They concluded that this paradox showed the Copenhagen interpretation of QM was incomplete. At the time, it was still held that QM could be described with existing suppositions of physics, i.e. that local realism held. What EPR did not know however, was that QM could not be described completely while assuming local realism, and they had unknowingly predicted what we now call quantum entanglement.

In 1964, John Bell released a paper entitled *On the Einstein Podolsky Rosen paradox*.[22] The theorem he presented, now known as Bell's Theorem, essentially stated the assumption of local realism was false. Bell introduced the *Bell inequality*, an inequality that cannot be violated under local realism, i.e. violation of the inequality implies there are no hidden local variables.

Today, multiple Bell inequalities exist and have been experimentally violated under various entanglement conditions. Thus, hidden local variable theories are no longer thought to be viable, and quantum effects are considered truly random (since, if the results are not truly random, there must be hidden local variables).

4.2 Quantum random number generators

Most QRNGs researched and implemented today rely on optics effects. We will consider three categories of this:

- Single photon detection (either spatial or temporal)
- Photon number detection
- Photon phase/frequency noise

4.2.1 Single photon detection QRNG

Among the earliest QRNG implementations, single photon detection (SPD) was usually employed. SPD directly follows from the probabilistic measurement in orthogonal bases. In SPD, a photon source is polarized in one basis and then passed through a polarizing beam splitter that measures in an orthogonal basis. The result is 50% of the photons being sent in either direction.

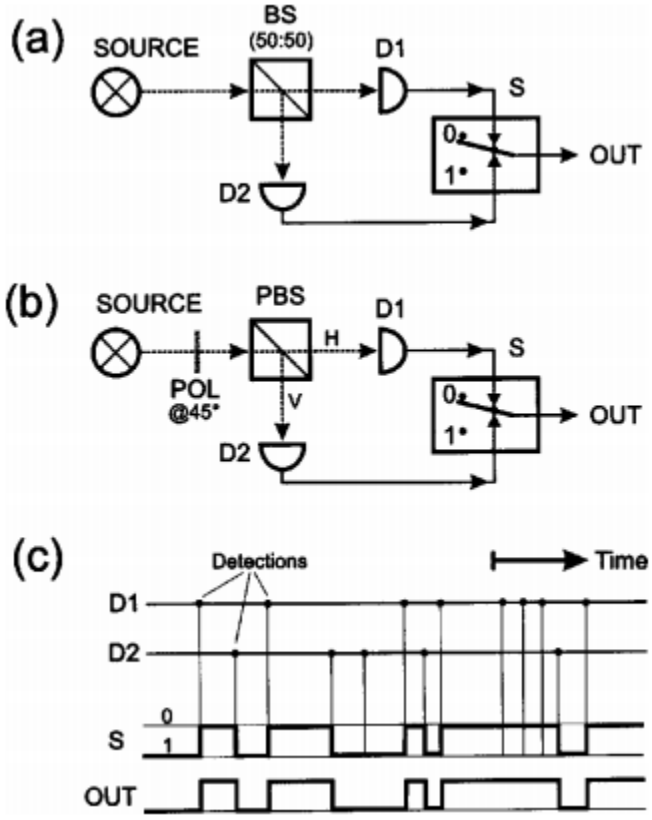

Figure 3: Experimental setup for SPD QRNG. The photons are processed through a 50/50 or polarized beam splitter. (c) shows the output values overtime, which are dependent on which detector was last fired. From [23].

In 2000, Jennewein *et al.* [23] set up an experiment utilizing SPD for random number generation. Their circuit was implemented as described above, with each of the detectors connected to a toggle switch. When a photon is incident on one of the detectors it flips the switch to high; when one is incident on the other it flips the switch to low. If multiple photons hit the same detector in a row, the switch is not changed.

The output of this circuit (which is just a single bit digital signal) is then measured periodically. On measurement, the output decides the next bit (1 for high, 0 for low) and the bit is appended to a 32-bit shift register. The consumer of the QRNG then takes a 32-bit number from the shift register after one is ready.

The final bitrate of random number generation was roughly 1Mbps, which is competitive with commercial classical HRNGs available today. Later refinements led to SPD QRNGs with bitrates as high as 152Mbps.[24]

4.2.2. Photon number detection

Later QRNG setups used photon number detection (PND). In this setup, a weak photon source is sent down a path, and the number of incident photons in a given period is measured. The number of incident photons follows a Poisson distribution:

$$P(n,\mu) = \frac{\mu^n}{n!}e^{-\mu}$$

$P(n,\mu)$ describes the probability of receiving n photons when there is a mean of μ photons being incident.

A QRNG of this type was presented in 2010 by Fürst *et al.*[25] They shone an LED through a photomultiplier tube set up to measure incident photons. In each sampling period, they recorded the number of incident photons, and decided the bit based on the parity of this number.

A problem the authors note is that a Poisson distribution is inherently skewed for one parity. In this case it was expected there would be more even numbers of photons than odd. They describe this bias with the following relation:

$$b = \frac{1}{2} - p_1 = \frac{1}{2} - \sum_{n=1,3,\ldots}^{\infty} P(n,\mu)$$

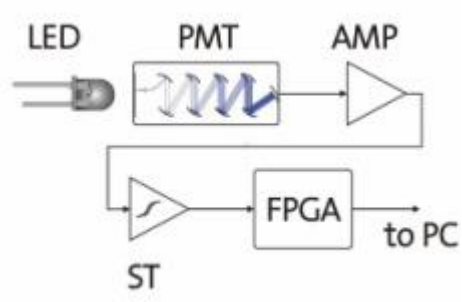

Figure 4: Experimental setup for PND QRNG. The photons pass through a photomultiplier tube, which has its output amplified and processed. From [25].

However, photomultiplier tubes set up for number measurement are subject to a phenomenon known as "dead-time". The photomultiplier tube must hold a charge in order to report a measurement. Thus, immediately after a measurement occurs, there is a recharge period where measurements cannot be performed. This means that multiple incident photons within a short period will only be seen as one photon.

The authors found that the bias caused by the dead time mitigated the bias resulting from the Poisson distribution, and concluded they had an unbiased source for even or odd numbers. Their final bitrate was roughly 50Mbps, and later similar implementations achieved up to 143Mbps [26].

4.2.3. Laser phase noise

Also in 2010, Qi, Chi, Lo, and Qian introduced the first QRNG based on photon phase fluctuations from a laser.[27] The photons from the laser were fed into a Mach-Zehnder interferometer. First, they go through a fiber coupler into two channels. One channel is longer than the other by ΔL, which delays the photons by $T_d = \frac{\eta \Delta L}{c}$, where η is the refraction index of

the fiber and c is the speed of light in a vacuum. If $T_d \gg \tau_c$ (the coherence time of the laser), the two channels can then be considered to be from independent sources.

With the phases of the two channels being essentially independent, measurement of the phase difference will be evenly distributed from the range $[-\pi, \pi)$. The authors then simply take the sign of the resulting measurement and use that as the random bit. This is an example of self-heterodyne detection.

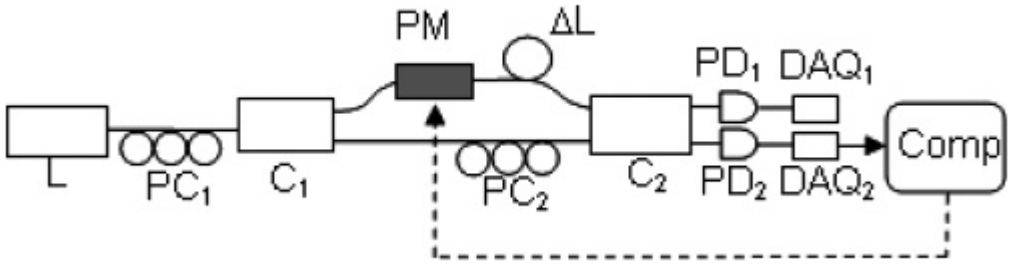

Figure 5: Experimental setup for self-heterodyne QRNG. The photons pass through a coupler, and one channel is longer. At the end, the output is processed by a computer so phase modulation can be used for feedback control, which is necessary because of phase drift due to temperature changes.

The final bitrate after refinement is 500Mbps. A later version from Nie *et al.* in 2015 [28] used active phase modulation. Not only did this remove the need for temperature-dependent feedback control, but it allowed them to decrease T_d to well below τ_c, and achieved a refined bitrate of 68Gbps.

4.3. Drawbacks of QRNGs

QRNGs can be seen as a special type of HRNGs. While classical HRNGs measure macroscopic effects, the sources still get their randomness from quantum effects. QRNGs use microscopic quantum effect sources but can be subject to similar drawbacks.

Classical noise – both from the device itself and external sources – can interfere with QRNGs. This may lead to biases in the output. Like traditional HRNGs, QRNGs must be designed around this problem and account for the bias in their output with post-processing.

Similarly, the black box problem applies to QRNGs. Being state of the art, commercial QRNG implementations may be kept as trade secret. This has the same implications it does for HRNGs. Not only does the manufacturer have to be trusted, but also the components can degrade. Again, these types of failures can be silent – randomness is very hard to quantify, especially when measurement at the raw entropy source is unavailable.

4.4. Trustless QRNGs

The ultimate protection against these problems is trustlessness – a system where trust is not required at all. This is reached in other fields of security. E.g., in 2008 the pseudonymous Satoshi Nakamoto published a paper [29] describing a trustless currency system called *Bitcoin*. The key point he stresses is that the protocol does not require trust of either party involved in the

transaction, nor of the nodes running the network. As another example, in Quantum Key Distribution we have protocols for secure communication that do not rely on trusting the channel or devices involved.

The next step for QRNGs is to make them trustless – devices whose output can be considered truly random without trusting the manufacturer, or even the components themselves. Trustless QRNGs exist and can be considered under two different categories: self-testing QRNGs and semi-self-testing QRNGs

4.4.1. Self-testing QRNGs

Self-testing QRNGs test their output for true randomness. This is done with Bell's Theorem. In the discussion in section 4.1, we quickly discussed how true randomness is related to local realism. In stricter terms, any bias or pattern in a QRNG's output that compromises its randomness actually manifests as hidden local variables. From Bell's Theorem, we know that Bell Inequalities can only be violated in the absence of hidden local variables. Therefore a QRNG producing outputs that violate Bell Inequalities can be considered truly random, without any consideration on the actual method used to produce the output.

The drawback to this method is that it is exceptionally slow. A self-testing QRNG utilizing SPD was introduced by Giustina *et al.* in 2013 [30] but achieved a meager bitrate of 0.4bps. For many applications this is clearly too slow. Generating a 256-bit AES key would take over 10 minutes, far longer than a user would tolerate when initiating a TLS session with a website. Nevertheless, this level of security could be desirable in applications where true entropy is required far more than speed.

4.4.2. Semi-self-testing QRNGs

It is not always necessary to remove trust of every component in a QRNG. As a compromise to the slow bitrate of self-testing QRNGs, semi-self-testing QRNGs are schemes where only part of the device is not trusted. They can fall into two general categories:

- Source-device-independent (SDI) QRNGs
- Measurement-device-independent (MDI) QRNGs

By trusting part of the QRNG hardware, Bell Tests are no longer required and other quantum effects can be exploited. In 2015 Marangon, Vallone, and Villoresi [31] introduced an SDI QRNG relying on electromagnetic effects. Their device first uses optical homodyne to measure the position quadrature of a Guassian state produced by the source. Randomness extraction is then performed using a conservative lower entropy bound. This bound is determined by randomly measuring in the momentum quadrature instead of position using the entropic uncertainty principle. Secure random bits are then generated and reinvested into the random quadrature selector to sustain the switching.

Using this method, the authors obtain a speed of 1Gbps. This is still well below the 68Gbps for trusted QRNGs, but far ahead of the previous self-testing example and sufficient for many purposes.

5. Classical *vs.* Quantum RNGs

We have discussed how QRNGs share similar drawbacks to HRNGs and looked at experimentally obtained bitrates. However, we have not discussed the commercial viability of QRNGs or the bitrates achieved by classical RNGs.

Today, QRNGs can be purchased as commercial devices. A company called id Quantique sells a variety of QRNGs that rely on SPD. The cheapest device is the *Quantis QRNG USB* [32], a $1500[m] purchase with a bitrate of 4Mbps. id Quantique also sells a PCIe based add-in-card for around $4500, which achieves 16Mbps.

To contrast, classical HRNGs are also available. The highest performing HRNG readily found is the TrueRNGpro [33], a USB-based device that sells for around $130 and has a 3.2Mbps bitrate. TrueRNG are not too open about their implementation, but it seems an avalanche diode effect is used for entropy sourcing.

Four software PRNGs were empirically tested for their bitrate:

- **C#/.NET System.Random: [7]** A general purpose, cryptographically-insecure PRNG that is the default RNG for C#/.NET development. Testing was done simply by requesting a variable number of bytes 100,000 times and averaging the time taken. Specifications of the software and hardware can be found in **Appendix 1.1**.
- **Windows 10 OS-level CSPRNG:** This CSPRNG is built-in to every Windows 10 computer and is what many consumers would be using for their TLS sessions. For testing, this was accessed through the C#/.NET *System.Security.Cryptography.RandomNumberGenerator* class, which forwards to the OS API call [10]. Testing methodology and hardware is the same as for the *System.Random* class. It was found that this CSPRNG scales with the number of bytes requested at a time. As such, results are included for requesting 32 bytes (typical scenario) and 6000 bytes (where it starts to level off) (see **Appendix 2**).
- **Linux /dev/random: [9]** Another OS-level CSPRNG, this function is built into the Linux kernel. Calls to this function block until enough entropy is gathered from system events, so it is very slow. Testing was done by copying from */dev/random* into */dev/null.* Specifications for the software and hardware can be found in **Appendix 1.2**.
- **Linux /dev/urandom:** Very similar to */dev/random*, but it does not block. If there is not enough entropy it will continue transitioning the PRNG, and thus it is much faster with the drawback that it may deliver poorer quality output. Tested with the same setup as */dev/random*.

[m] All dollar amounts in Canadian dollars.

A further result is included as Intel's RDRAND instruction. Values are not obtained empirically, but methodology can be found in **Appendix 3**.

The results are summarized in **Figure 6**. Coming in first for bitrate is the self-heterodyne QRNG at 68Gbps. In second place is the Windows CSPRNG at roughly 30Gbps when requesting 6000 bytes. Interestingly, *System.Random* is significantly slower at 1.4Gbps[n], despite not being cryptographically-secure. After looking through the source [7], this is because it is poorly optimized when requesting many bytes at a time. It is however well optimized for smaller byte counts and beats the Windows CSPRNG when requesting 32 bytes at a time (the latter only achieves around 600Mbps) (see **Appendix 2** for a graph of the scaling).

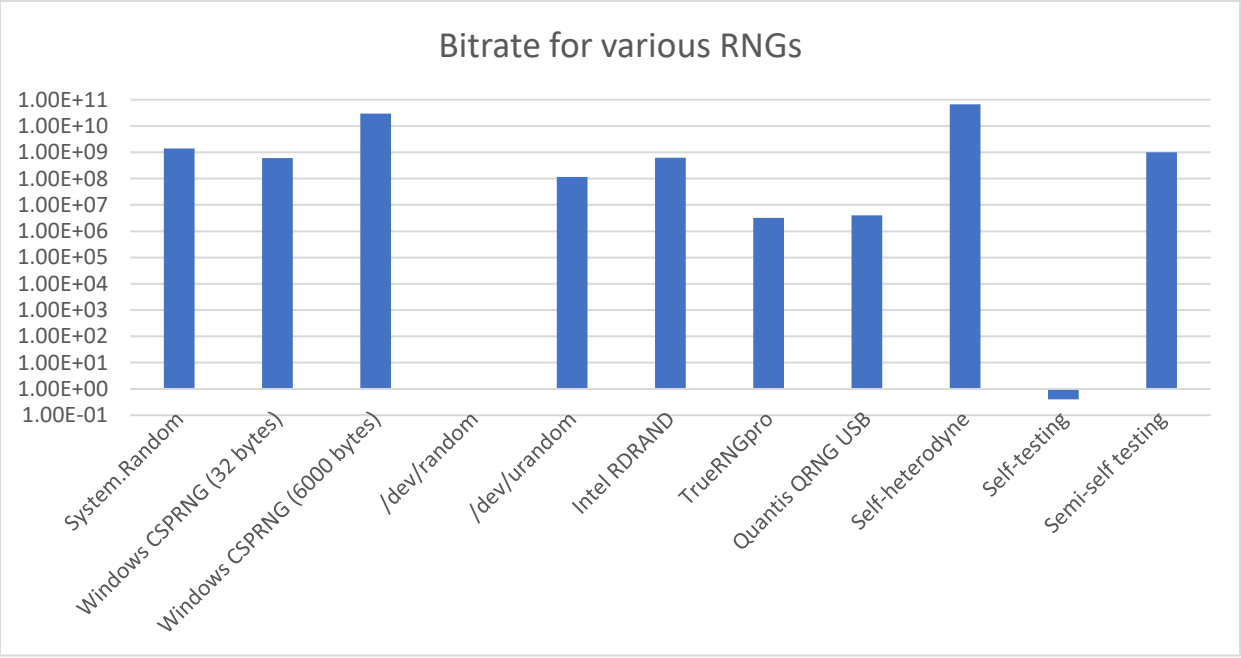

Figure 6: Bitrate results for various RNGs. Note the axis is on a logarithmic scale in bps. Also note /dev/random test did not complete

Not completing the test was */dev/random*. This is likely due to the quirks involved in running as a headless virtual machine: there is not enough hardware it can access to draw entropy. */dev/urandom* did complete however, but only with a measly 117Mbps.

Intel's RDRAND comes in at 622Mbps. However, it is important to note that this does not reflect the true entropy rate. Recall that Intel's HRNG is a black box and the entropy source cannot be measured, only the output of its accompanying PRNG can.

The commercially available *TrueRNGpro* and Quantis *QRNG USB* are significantly slower than the PRNGs – however they provide a level of entropy that the latter do not.

[n] *System.Random* testing was done with only one CPU thread. On the same CPU, it would be possible to instantiate 12 copies of the object and increase overall bitrate roughly 10x. The same cannot be said about OS-level PRNGs.

This leads to the next point: the quality of resulting entropy is not considered here. It is reasonable to assume the HRNG and QRNG products produce much better distributions, but they are still susceptible to side-channel attacks.

6. QRNGs for classical cryptography

Finally, we have enough information to get to the main point. How effective are QRNGs at improving classical cryptography?

To answer this question, we consider two very different use-case scenarios. First, scenarios that require large amounts of random bits but do not care as much about the quality of the output. Second, scenarios that absolutely care about the quality, regardless of how long generation takes.

6.1. Bandwidth-limited scenarios

In our discussion of TLS, there is not really a need for high-speed random number generation. Certificates are typically very long-lived on the order of years, and secrets used for sessions can be generated practically in real time with something like the *TrueRNGpro*'s 3.2Mbps.

A previously unexplored application of classical cryptography is *session tokens*. When a user logs-on to a website, they are usually given a "Remember Me" option so that they do not need to log-in when next opening the browser. This is accomplished by storing a session token in the browser as a *cookie*. Like encryption keys, it is important that this session token not be ascertained by an attacker since it could allow them to resume a logged-in session. Also like keys, it is best to choose these token values at random.

Similarly, on account registration a user typically decides on a password. In proper practice, this password is stored by the website as a *salted hash* to prevent rainbow attacks. The *salt* is a random value hashed alongside the user's password and is unique to that user. Therefore, a new salt is generated for every registration.

It is reasonable to assume that both session tokens and hash salts are generated at huge rates for popular websites. Here, the generation speed QRNGs can provide on the order of 10s of Gbps provides a clear advantage over classical HRNGs, particularly for servers that are generating both simultaneously.

6.2. Security-limited scenarios

Here we see advantages for TLS, and arguably the most important benefit provided by QRNGs.

As stated before, certificates for TLS are very long-lived. For a typical domain certificate that is valid for one year, taking minutes to generate the private key is very practical. Given the importance of a website's domain certificate, it is imperative that the keys are truly chosen at

random. An attacker that can ascertain the private keys to a domain certificate can theoretically defraud all users of that website.

This problem further compounds as we go up the certification path. All domain certificates can be traced back to a root certificate owned by a root certificate authority (RCA). There are not many RCAs. One RCA, IdenTrust, provides certification for 49.5% of TLS/SSL protected websites [5], for all of which the authority comes from a single root certificate. Root certificates are so important to RCAs that the private keys are split into fragments and stored in safes by several different executives of the company. Because of this, RCAs deploy "intermediate" certificates that are used to sign domain certificates, instead of risking the root certificate's private key fragments by bringing them out of storage. The root certificate private keys are then only needed when a new intermediate certificate needs to be generated.

Root certificates can be valid for as long as 30 years. An RCA may even hold a "key ceremony" when a new root certificate needs to be signed, where executives (along with lawyers and witnesses) lock themselves in a vault that has no communication to the outside world while the keys are generated. In the very unfortunate event that a root certificate is compromised, its revocation requires all browsers and OSs that include it to be updated to take effect. An attacker then could continue attacking users that do not update their software. Furthermore, that is assuming that the RCA is even aware their root certificate has been compromised. With root certificate keys an attacker could "prove" they are the owner of any website they choose. For a company like IdenTrust that secures half of all protected websites, such a compromise could hamstring the internet.

Given all this, an RCA can accept key generation taking hours or even days. This brings the 0.4bps self-testing QRNG into a new light, where its fundamental integrity far outshines its slow bitrate. Even during the generation of intermediate certificates such a low bitrate can be tolerated.

Aside from TLS, similarly important certificates may be needed in a variety of areas including military communication, ePassport signatures, or server access keys (e.g. SSH). Consider if the root certificate for a nation's ePassport certification is compromised; the attacker could pretend to be a citizen of that country.

7. Conclusion

Despite being an integral part to modern cryptography, conventional random number generators fall short in several areas. There are many examples of bad RNG implementations that have led to compromised products. Compounding this issue is the difficulty of statistically measuring randomness, so when failures do occur, they may go unnoticed for years.

While commercial QRNGs do not provide any obvious advantages over currently available HRNGs, they do make use of fundamentally random effects as predicted by quantum mechanics.

Theoretical QRNGs can provide bitrates far higher than current HRNGs and have uses in large platforms where a high amount of random number generation is needed.

Self-testing QRNGs provide a level of integrity that no classical RNG could reach. Using proven physical theories, they provide the only fundamentally secure RNG available. This is of great use for long-lived private keys that are highly critical, and where the time taken to generate is inconsequential.

While there are no self-testing or high-bitrate QRNGs commercially available today, the research is there. Integration of these devices in the near future could bolster classical cryptography, at least until more secure quantum cryptography is viable for widespread use.

References

[1] Shannon, C. Communication Theory of Secrecy Systems. (1949). *Bell System Technical Journal*; 28(4): 656-715. doi:10.1002/j.1538-7305.1949.tb00928.x.

[2] United States National Institute of Standards and Technology. Announcing the Advanced Encryption Standard (AES). (2001). *Federal Information Processing Standards Publication 197.* http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[3] Rivest, R., Shamir, A., Adlemane, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. (1978). *Communications of the ACM*; 21(2): 120-126. doi:10.1145/359340.359342.

[4] Dierks, T., Rescorla, E. The Transport Layer Security (TLS) Protocol, Version 1.2. (2008). https://tools.ietf.org/html/rfc5246.

[5] W3Techs Web Technology Surveys. Usage of SSL certificate authorities for websites. (2019). https://w3techs.com/technologies/overview/ssl_certificate/all.

[6] Dorrendort, L., Gutterman, Z., Pinkas, B. Cryptanalysis of the Random Number Generator of the Windows Operating System. (2007). http://www.cs.huji.ac.il/~dolev/pubs/thesis/msc-thesis-leo.pdf.

[7] Microsoft .NET Foundation. Random.cs. (2018). *.NET Core Source Code.* https://github.com/dotnet/corefx/blob/master/src/Common/src/CoreLib/System/Random.cs.

[8] Microsoft. CryptGenRandom function. (2018). *Windows API Documentation.* https://docs.microsoft.com/en-us/windows/desktop/api/wincrypt/nf-wincrypt-cryptgenrandom.

[9] Torvalds, L. Linux Kernel drivers/char/random.c comment documentation. (1999). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/char/random.c?id=refs/tags/v3.15.6.

[10] Microsoft .NET Foundation. RandomNumberGeneratorImplementation.Windows.cs. (2018). *.NET Core Source Code*. https://github.com/dotnet/corefx/blob/master/src/System.Security.Cryptography.Algorithms/src/Internal/Cryptography/RandomNumberGeneratorImplementation.Windows.cs.

[11] Random.org. Statistical Analysis. https://www.random.org/analysis/.

[12] United States National Institute of Standards and Technology. Guide to General Server Security. (2008). http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf.

[13] Arduino. Random. *Arduino Reference.* https://www.arduino.cc/reference/en/language/functions/random-numbers/random/.

[14] Intel Corporation. Intel Digital Random Number Generator (DRNG): Software Implementation Guide, Revision 1.1. (2012). http://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf.

[15] Hamburg, M., Kocher, P., Marson, M. Analysis of Intel's Ivy Bridge Digital Random Number Generator. (2012). *Cryptography Research, Inc.* https://web.archive.org/web/20141230024150/http://www.cryptography.com/public/pdf/Intel_TRNG_Report_20120312.pdf.

[16] Goldberg, I., Wagner, D. Randomness and Netscape Browser. (1996). *Dr. Dobb's Journal.* https://people.eecs.berkeley.edu/~daw/papers/ddj-netscape.html.

[17] Schneider, B. Random Number Bug in Debian Linux. (2008). https://www.schneier.com/blog/archives/2008/05/random_number_b.html.

[18] Perlroth, N. Government Announces Steps to Restore Confidence on Encryption Standards. (2013). *The New York Times.* https://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/.

[19] Menn, J. Exclusive: Secret contract tied NSA and security industry pioneer. (2013). *Reuters*. https://www.reuters.com/article/2013/12/20/us-usa-security-rsa-idUSBRE9BJ1C220131220.

[20] Perlroth, N., Larson, J., Shane, S. N.S.A Able to Foil Basic Safeguards of Privacy on Web. (2013). *The New York Times.* https://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html.

[21] Einstein, A., Podolsky, B., Rosen, N. Can Quantum-Mechanical Description of Physical Reality be Considered Complete? (1935). *Physical Review*; 47(10): 777-780. doi: 10.1103/PhysRev.47.777.

[22] Bell, J. On the Einstein Podolsky Rosen Paradox. (1964). *Physics*; 1(3): 195-200. doi: 10.1103/PhysicsPhysiqueFizika.1.195.

[23] Jennewein, T., *et al.* A fast and compact quantum random number generator. (2000). *Rev. Sci. Instrum*; 71: 1675-1680.

[24] Wahl, M. *et al.* An ultrafast quantum random number generator with provably bounded output bias based on photon arrival time measurements. (2011). *Appl. Phys. Lett.*; 98:171105.

[25] Fürst, H., *et al.* High speed optical quantum random number generation. (2010). *Opt. Express*; 18: 13029-13037.

[26] Applegate, M. *et al.* Efficient and robust quantum random number generation by photon number detection. (2015). *Appl. Phys. Lett.*; 107: 071106.

[27] Qi, B., Chi, Y.-M., Lo, H.-K., Qian, L. High-speed quantum random number generation by measuring phase noise of a single-mode laser. (2010). *Opt. Lett.*; 35: 312-314.

[28] Nie, Y.-Q., *et al.* 68 Gbps quantum random number generation by measuring laser phase fluctuations. (2015). arXiv:1506.00720.

[29] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008). https://bitcoin.org/bitcoin.pdf.

[30] Qiustina, M., *et al.* Bell violation using entangled photons without the fair-sampling assumption. (2013). *Nature*; 497: 227-230.

[31] Marangon, D. G., Vallone, G., Villoresi, P. Source-device-independent ultra-fast quantum random number generation. (2015). arXiv:1509.07390.

[32] id Quantique. Quantis Random Number Generator. https://www.idquantique.com/random-number-generation/products/quantis-random-number-generator/.

[33] ubldit. TrueRNGpro – USB Random Number Generator. http://ubld.it/products/truerngpro.

Appendix

1. Test specifications

1.1 *System.Random* and Windows CSPRNG test setup

- OS: Windows 10 x64, build 17134.648
- Runtime: .NET Core 2.1.9
- IDE: Visual Studio 2019 16.0.1
- CPU: Intel Core i7-8700k, 6core/12thread @ 5.0GHz
- Motherboard: Gigabyte Aorus Gaming 7 Z370
- GPU: 2x NVIDIA GTX 1080 Ti
- RAM: 32GB DDR4-2400
- Storage: Samsung 960 Evo 500GB
- PSU: Corsair HX1000i

1.2. Linux kernel */dev/random* and */dev/urandom* test setup

- OS: Ubuntu 16.04.6 LTS Server edition (VM)
- Hypervisor: Hyper-V under Windows Server 2019 Datacenter Edition
- CPU: Intel Core i7-5820k, 6core/12thread @ 3.7GHz
- Motherboard: ASUS X99 SABERTOOTH
- GPU: AMD R9 Fury X
- RAM: 32GB DDR4-3000
- Storage: Tiered Microsoft Storage Space, 2x (Samsung 850 Evo 250GB + Toshiba 5TB HDD) in RAID1
- PSU: Corsair RM1000i

2. Scaling of Windows CSPRNG

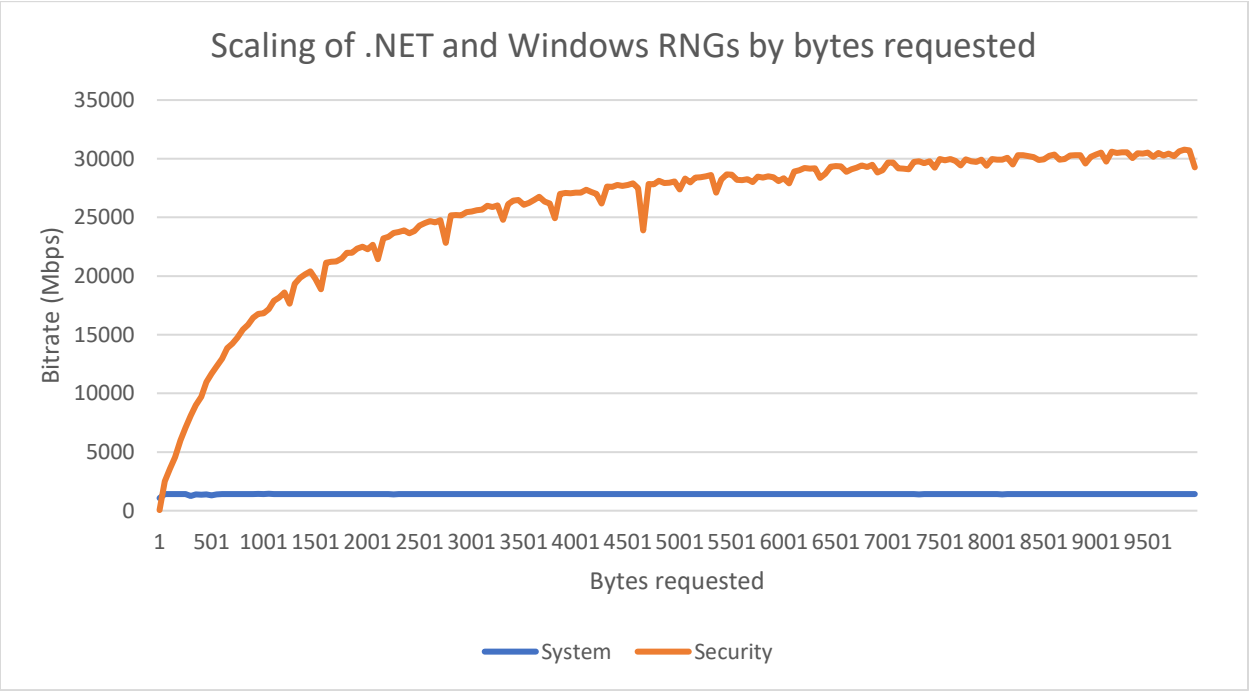

Figure 7: Scaling of .NET System.Random and System.Security.Cryptography.RandomNumberGenerator (forward to Windows CSPRNG). System.Random stays very flat, but the Windows CSPRNG scales up to around 6000 bytes and levels off around 30Gbps. On a smaller scale, the System.Random performs higher for considerable time before being overtaken; the scale is large here to show the leveling off.

3. Intel RDRAND performance calculation

Intel states that the output of their HRNG takes 463 clock cycles to produce on Kaby Lake processors, regardless of operand size (maximum 64 bits). So, assuming a clock rate of 4.5GHz, we can get bitrate:

$$bitrate = \frac{(frequency) \times (number\ of\ bits\ each\ operation)}{(number\ of\ cycles\ needed)}$$

$$= \frac{4500000000\ \text{cycles/s} \times 64\ \text{bits}}{463\ \text{cycles}}$$

$$= 622\text{Mbps}$$