Pure Random Number Generator

Project Report submitted in partial fulfillment of the requirement for the degree of Bachelor of Technology.

in

Computer Science and Engineering/Information Technology

By

Abhijit Srivastava *(*131307*)*

under the Supervision of

Dr. Yashwant Singh

to

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology Waknaghat,

Solan-173234, Himachal Pradesh

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"**Pure Random Number Generator**"** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from August 2016 to May 2017 under the supervision **of Dr. Yashwant Singh, Associate Professor** Computer Science Department. The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Abhijit Srivastava
131307

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Yashwant Singh Associate Professor
Department of Computer Science

Dated:

Acknowledgement

I take this opportunity to express my profound gratitude and deep regards to my guide Mr. Yashwant Singh for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by his time to time shall carry me a long way in the journey of life on which I am about to embark.

The in-time facilities provided by the Computer Science department throughout the project development are also equally acknowledgeable.

At the end I would like to express my sincere thanks to all my friends and others who helped me directly or indirectly during this project work.

Date: April 30th, 2017 **Abhijit Srivastava**

131307

Table of Contents

Serial Number	Topics	Page Numbers
1	**Chapter-1. Introduction**	1
2	1.1 About Random Number Generators	2
2	1.1.1 Random Number Generators (RNGs)	2
3	1.1.2 Pseudo-Random Number Generators (PRNGs)	3
4	1.1.3 True Random Number Generators (TRNGs)	5
5	1.1.4 Cascade Construction RNGs	6
6	1.2 Problem Statement	7
7	1.3 Objective	7
8	1.4 Methodology	7
9	**2. Literature Survey**	8
10	2.1. Intel® Digital Random Number Generator (DRNG)	8
11	2.2. Quantis Random Number Generator	13
12	2.3. True Random Number Generator With a Metastability-Based Quality Control	15
13	2.4. True Random Number Generator based on compact chaotic oscillator	15
14	2.5. A truly random number generator based on thermal noise	15
15	2.6. Simple true random number generator for any semiconductor technology	16
16	2.7. True Random Number Generator Based on ROPUF Circuit	16
17	2.8. A True Random Number Generator algorithm from digital camera image noise for varying lighting conditions	16
18	2.9 The Mersenne Twister is a pseudorandom number generator	17
19	2.10 Yarrow algorithm	18
23	**Chapter-3 System Development**	22
24	3.1 Broad-Scale Distribution of Working Process	22
25	3.2 Modular Distribution	23
26	3.2.1 MODULE 1: Android Application for Sensor Data Collection	23

27	3.2.2 MODULE 2: Recording Data Using SQLite	24
28	3.2.3 MODULE 3: Server-Side Scripting for extraction of meaningful information.	25
29	3.2.4 MODULE 4: Entropy pool generator	27
30	3.2.5 MODULE 5: Range optimization	27
31	**Chapter-4 Performance Annalysis**	29
32	4.1 Algorithmic Complexity	29
33	4.2 Resource Utilization at runtime	29
39	4.3 Applications Contributions	30
40	4.3.1 Caesar Cipher	30
41	4.3.2 Games	32
42	4.3.3 Science	32
34	**Chapter-5 Conclusion**	34
35	5.1 Conclusions	34
43	5.2 References	35

List of Figures, Graphs and tables

S. No.	Tables, Figures and Graphs	Page Number
1	Table 1.1: Comparison between PRNG and TRNG	2
2	Figure 1 : Digital Random Generator Overview	9
3	Figure 2 : Digital Random Generator Component Architecture	10
4	Figure 3 : Robustness and Self-Validation	12
5	Figure 4 : Quantium based Random Number Generator	13
6	Figure 5 : Android code snippit	23
7	Figure 6 : Android App Live Graph	24
8	Figure 7 : Android App UI	24
9	Figure 8 : Sampling Database Screenshot	25
10	Figure 9 : Screenshot of range optimizer UI	28
11	Figure 10 : Resource Utilization at Runtime	29
12	Graph 1 : Raw sensor data of Game Rotation Vector [While Climbing Stairs]	25
13	Graph 2 : Raw sensor data of GeoMagnetic Rotation Vector [While Walking]	26
14	Graph 3 : Processed data of Game Rotation Vector [While Climbing Stairs]	26
15	Graph 4 : Processed sensor data of GeoMagnetic Rotation Vector [While Walking]	27

Chapter-1. Introduction

As we all know computer based "von Neumann Architecture" are designed to achieve 100% efficiency, i.e. Generation of any random data states that the computation went wrong. That is why for the generation of random numbers, programmers and mathematicians have been designing complex algorithms with a large period for the generation of pseudo-random numbers.

The word ‘pseudo’ means the pseudo-random numbers are not purely random in a way you might expect, at least not if it is compared to dice rolls or lottery tickets. Essentially, pseudo-random numbers generators are algorithms that use some kind of mathematical formula or just a pre-calculated tables to generate sequences of numbers that appear to be randomly generated. An example of a pseudo-random numbers generator can be the linear congruential methodology.

In comparison with pseudo-random numbers generators, true random numbers generators take into account randomness from some physical phenomena and introduce its reading into the computer. You can imagine an example of a trivial die connected to a computer, but generally, people use a phenomenon that is simple to connect to the computer as compared to a die. The physical phenomenon can be very general, like the slight variations in the movement of mouse or time lag between keystrokes while typing on the keyboard. But, you should be careful about which source you are choosing. For instance, it can be difficult to use keystrokes in this fashion, the reason being keystrokes are often buffered into the computer's operating system memory, that means several keystrokes are collected into the buffer before they can be sent to the program waiting for data to process. From the side of program waiting for the keystroke data, it might seem as all the keys are pressed almost at the same time, and there will now not be a lot of difference there after all.

The table below provides a characteristic comparison between the Pseudo-Random Number Generators and Pure-Random Number Generators.

Table 1.1: Comparison between PRNG and TRNG

Characteristic Features	**PRNG**	**TRNG**
Efficiency of Generation	Excellent efficiency	Poor efficiency
Determinism of Numbers	Deterministic in nature	Nondeterministic in nature
Periodicity in data	Periodic in nature	Aperiodic in nature

These features thus make True-Random Number Generators suitable for roughly the set of applications that Pseudo-Random Number Generators are not suitable for, for instance data encryption, games, and gambling. Although, the poor generation efficiency and nondeterministic nature of True-Random Number Generators make them less suitable for simulation and modeling type of applications, which often needs more data than it is feasible to computes with any True-Random Number Generator.

1.1 About Random Number Generators

This section explains the basic concepts behind the random number generation.

1.1.1 Random Number Generators (RNGs)

A random Number Generators is a software or hardware of any type that can produce a sequence of numbers between any interval [min, max] such that values appear are pure not deterministic in nature .

Each and every new value has to be mathematically independent of any previous value or data. I.e. given a computed sequence of numbers, a particular data is not more likely

to follow after it as the next value of the Random Number Generator's random sequence. The overall series of numbers chosen between the interval shall be uniformly distributed. In other words, all the numbers(or values) should be equally likely and none should be more "popular" or occur more occasionally within the Random Number Generator's output than the others.

The sequence should also be unpredictable in nature. An attacker should not be able guess some or all of the numbers in a computed sequence. Predictability can take some form of forward prediction and backtracking.

As the computing systems are deterministic by nature, producing quality random numbers that have these features is much more tougher than it might look. Taking the second's value from the computer system clock, a general method, may seem random, but the method of process scheduling and other system effects may lead in some values occurring far more occasionally than the others. External data sources as the interval between keystrokes or movement of the mouse may likewise, upon extensive analysis, show that numbers do not distribute evenly across the interval of all possible. Beyond these characteristics, some other desirable random number generator features include:

- The random number generator should be quick in computing a value and can cater a large number of requests in a short interval of time.
- The random number generator should secure against attackers.

1.1.2 Pseudo-Random Number Generators (PRNGs)

One broadly used approach for getting good Random Number Generators statistical behavior is to leverage statistical modeling in the creation of a Pseudo-Random Number Generator. A Pseudo-Random Number Generators is a predictable algorithm, typically implemented in software that generates a series of numbers that looks random. A Pseudo-Random Number Generators needs a seed value that has to be used to set the state of the

given model. Once seeding is done, it can then compute a series of value that exhibit a better statistical behavior.
Pseudo-Random Number Generators exhibit periodicity that is size dependent on its internal state model. I.e., after computing a long series of values, all variations in the internal state will be exhausted and the series of values to follow shall reoccur an earlier series. The best Pseudo-Random Number Generators algorithms available today, too large and complex that this drawback could practically be unseen. For example, the Mersenne Twister MT19937 Pseudo-Random Number Generator has 32-bit word length has a high periodicity of 219937-1.A key feature of all Pseudo-Random Number Generators is that they are predictable. I.e, given a certain seed value, the same Pseudo-Random Number Generators will always produce the exact same series of "random" values. The reason behind this is that, a Pseudo-Random Number Generators is computing the successive value based upon a certain internal state and a pre-defined algorithm. So, while a generated series of numbers exhibits the mathematical properties of randomness, the cumulative behavior of the Pseudo-Random Number Generators is entirely deterministic.

In situations, the deterministic nature of Pseudo-Random Number Generators is an advantage. For instance, in some simulation and experimental, researchers would may want to examine the outcome of different approaches using the same series of input values. Pseudo-Random Number Generators provide a way to compute a large series of random data inputs that are repeatable by using the same Pseudo-Random Number Generator, seeded with the repeated value.

In other situations, however, this determinism is highly unwanted. for instance a server application that computes random numbers that are to be used as cryptographic keys in information exchanges among client applications over secure communication network. An attacker who knows the Pseudo-Random Number Generator in use and also knew the seed value would quickly be able to guess each and every key that is being generated by the Pseudo-Random Number Generator. Even with a highly sophisticated seeding algorithms, an attacker who has the information about the Pseudo-Random Number Generator in use can

deduce the state of the Pseudo-Random Number Generator by observing the series of generated values.

Pseudo-Random Number Generators researchers have researched to solve this problem by creating what are known as Cryptographically Secure Pseudo-Random Number Generator or the CSPRNGs. Many complex techniques have been designed in this field, for instance, applying a cryptographic hash to a series of consecutive integer numbers, using a block cipher to encrypt a series of consecutive integer numbers, and XORing a stream of Pseudo-Random Number Generator generated numbers with plaintext. Such methods improve the problem of inferring a Pseudo-Random Number Generator and its state by exponentially increasing its computational complexity, but the final values may or may not have the correct statistical features needed for a good random number generator. Further, an attacker can find any deterministic algorithm by a number of methods (e.g., memory attacks, sophisticated, a disgruntled employee, disassemblers etc). Even more simpler, attackers can find or infer Pseudo-Random Number Generator seeding by narrowing down its range of possible numbers or by surfing the memory in any manner. Once the algorithm in use and its seed values are known, an attacker is be able to guess each and every random number computed, both in past as well as in future.

1.1.3 True Random Number Generators (TRNGs)

For situations where the predictable nature of Pseudo-Random Number Generators is a problem to be avoided (for example, computer security and gaming), a better way is that of True Random Number Generators.

Instead of implementing a any mathematical model to deterministically compute numbers that look random and have the correct statistical features, a True Random Number Generators extracts entropy from any physical phenomenon and then uses the values to generate random values. The physical phenomenon is also called an entropy source and can be selected among a wide range of physical phenomenon that are naturally available, or is made usable, to the computing device using the True Random Number Generators. For example, one can try to

use the time interval between users consecutive keystrokes or movement of mouse as an entropy source. As stated earlier, this method is crude in practice and resulting value series usually fail to meet desired features. Selection of an entropy source in a True Random Number Generators is a key problem facing True Random Number Generators designers.

Beyond desired features, True Random Number Generators should be scalable and fast. This poses a serious challenge for many True Random Number Generators, the reason for that is sampling an entropy source that is external to the computing device usually needs device I/O and large delay relative to the computing efficiency of today's computers. Thus, sampling any entropy source in True Random Number Generators is slow with respect to the computation needed by a Pseudo Random Number Generators to simply compute its successive random value. Unlike Pseudo Random Number Generators, however, True Random Number Generators are not predictable. That is, a True Random Number Generators need not be seeded, and its selection of random numbers in any given series is almost unpredictable. An attacker cannot observe of a particular random value series to guess successive values in an efficient way. This feature also implies that True Random Number Generators have no periodicity. Although repeats in the random values are possible, they cannot be guessed in any manner.

1.1.4 Cascade Construction RNGs

A general method implemented by modern operating systems and cryptographic algorithms is to take input values from an entropy source to create pool of entropy. This entropy pool is used to supply non-deterministic random values that regularly seeds a Cryptographically Secure Pseudo Random Number Generators. This Cryptographically Secure Pseudo Random Number Generators generates cryptographically secure random values that appear truly random.

The main advantage of this method is performance efficiency. It was previously stated that sampling of any entropy source is usually slow and generally additional waiting for a real-time sampling work to transpire. In comparison, Cryptographically Secure Pseudo Random

Number Generators computations are efficient since they are computation-based and sidelines entropy source delays. This method usually leads to improved performance: a slow entropy source periodically sending a fast Cryptographically Secure Pseudo Random Number Generators capable of computing a large number of random numbers from a single seed value.

1.2 Problem Statement

As discussed in introduction the pseudo-random number generation is not a full proof method of generation of random number because of its property of reverse engineering ability. Random numbers generators should not use a seed value and generated a number which cannot be predicted at all costs.

1.3 Objective

1. Study of the existing Random number generators.
2. To design a "Pure Random Number Generator ".
3. To study a performance analysis of designed "Pure Random Number Generator ".

1.4 Methodology

The data is first collected from a physical phenomenon probably non periodic and storing it as entropy poll. This entropy poll is then accessed by an algorithm to generate random numbers as per requirements of a system of a user.

1.5 Organization

Chapter 2 deals with the first objective of study of existing Random number generator and finding a comparative statistics of advantages and drawbacks. Chapter 3 shows the system development of pure random number generated designed in the project. Chapter 4 deals with the performance analysis and applications of Pure Random Number Generator. Chapter 5 defines the final conclusion followed by Appendix.

Chapter-2. Literature Survey

2.1. Intel® Digital Random Number Generator (DRNG)

The "Digital Random Number Generator" or DRNG is an efficient and innovative hardware approach for generation of high-quality, high-performance entropy and random number. It comprises of the new Intel 64 Architecture instructions the 'RDRAND' and the 'RDSEED' and an underlying Digital Random Number Generator hardware implementation.

In context to the Random Number Generator taxonomy stated above, the Random Number Generator follows the cascade construction Value Generator model, using a processor resident entropy source to periodically seed a hardware-implemented Cryptographically Secure Pseudo Random Number Generators. Unlike the software methods, it includes a high-quality entropy source usage that can be sampled quickly enough to repeatedly seed the Cryptographically Secure Pseudo Random Number Generators with high-quality entropy values. It also represents a self-stationed hardware module that is isolated from any king of software attacks on its internal state. This results in a solution that achieves Random Number Generator objectives with considerable features.

This method of digital random number computation is not same in its process with respect to true random number computation in that it is implemented into the processor and can be accessed using Intel 64 instruction set. The response times are comparable to those of competing Pseudo-Random Number Generators approaches implemented in any software. This method is scalable enough for the demanding applications to use it as an extensive source of random values and not merely a high quality seed for a software-based Pseudo-Random Number Generators. Software running at all privilege levels can access random values through the instruction set, ignoring any intermediate software or libraries.

Applications for the Digital Random Number Generator

Information security is a key application that utilizes the DRNG. Cryptographic protocols rely on RNGs for generating keys and fresh session values (e.g., a nonce) to prevent replay attacks. In fact, a cryptographic protocol may have considerable robustness but suffer from widespread attack due to weak key generation methods underlying it (e.g., the Debian*/OpenSSL* fiasco (3)). The DRNG can be used to fix this weakness, thus significantly increasing cryptographic robustness.

Closely related are government and industry applications. Due to information sensitivity, many such applications must demonstrate their compliance with security standards like FISMA, HIPPA, PCIAA, etc. RDRAND has been engineered to meet existing security standards like FIPS 140-2 and NIST SP800-90 and thus provides an underlying RNG solution that can be leveraged in demonstrating compliance with information security standards.

Other uses of the D-Random Number Generator include:

- Communication protocols
- Bulk entropy applications like secure disk wiping or document shredding
- Monte Carlo simulations and scientific computing
- Gaming applications
- Protecting online services against Random Number Generator attacks
- Seeding software-based Pseudo-Random Number Generators of arbitrary width

Digital Random Number Generator Overview

This section, describes in some detail the components of the DRNG using the "RDRAND" and "RDSEED" instructions and their interaction.

Processor View: Figure provides a high-level schematic of the RDRAND and RDSEED Random Number Generators. As shown, the DRNG appears as a hardware module on the processor. An interconnect bus connects it with each core.

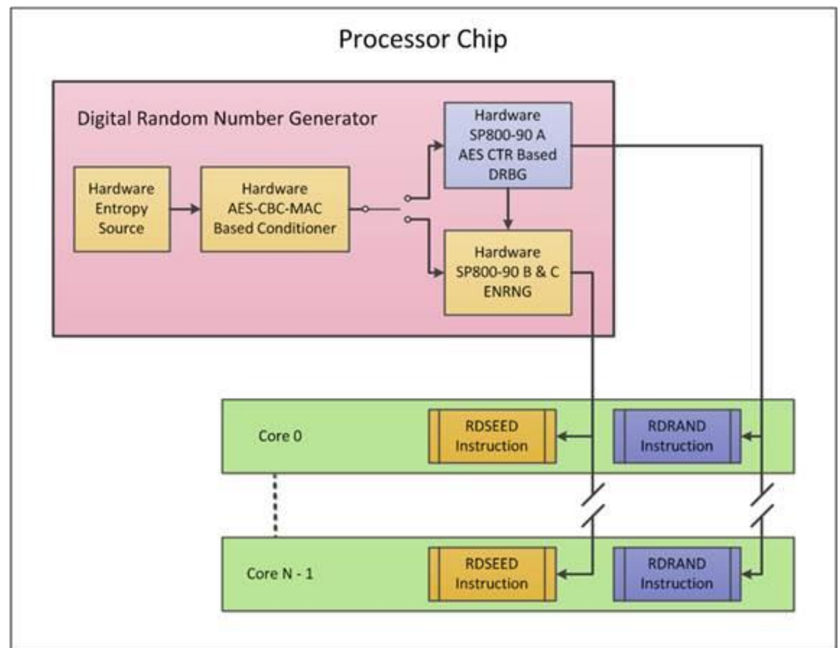


Figure 1 : Digital Random Generator Overview

The RDRAND and RDSEED instructions (detailed in section 4) are handled by microcode on each core. This includes an RNG microcode module that handles interactions with the DRNG hardware module on the processor.

<u>Component Architecture</u>: As shown in figure the DRNG can be thought of as three logical components forming an asynchronous production pipeline: an entropy source (ES) that produces random bits from a nondeterministic hardware process at around 3 Giga bits per sec, a conditioner that uses AES in CBC-MAC mode to distill the entropy into high-quality nondeterministic random numbers, and two parallel outputs:

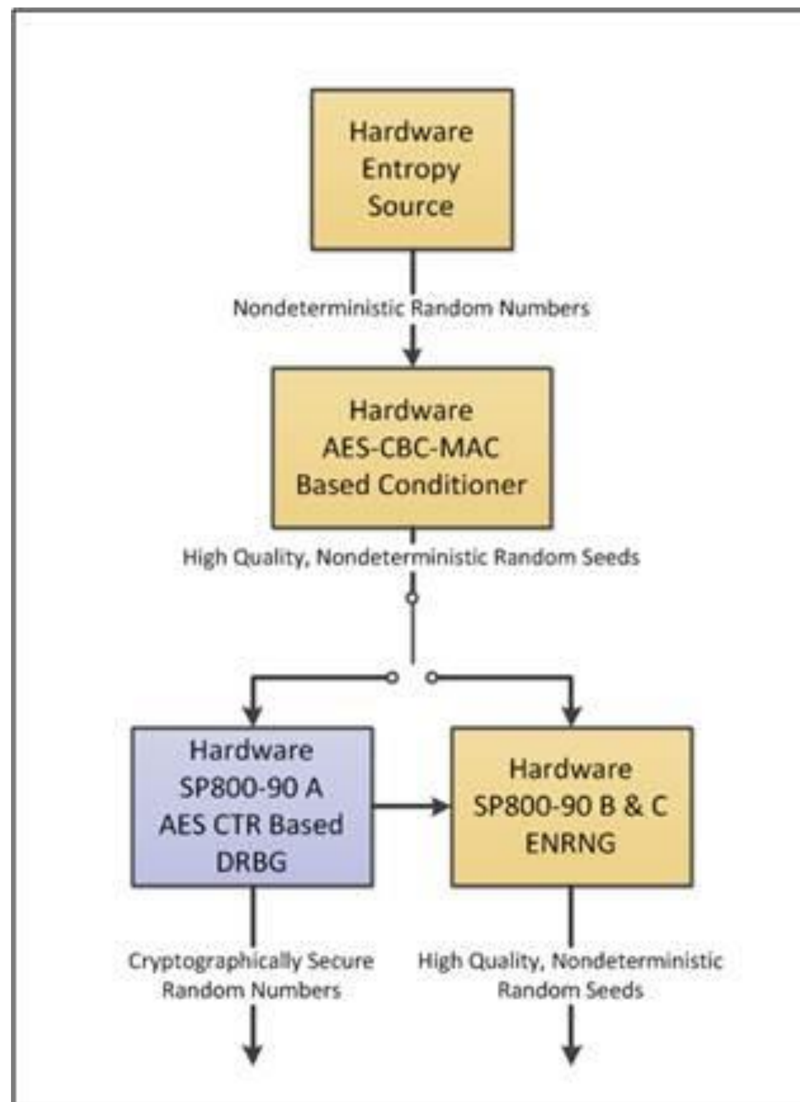


Figure 2 : Digital Random Generator Component Architecture

1. The random bit generator which is seeded from the conditioner.
2. An enhanced, nondeterministic random number generator that provides seeds from the entropy conditioner.

Note that the conditioner *does not* send the same seed values to both the DRBG and the ENRNG. This pathway can be thought of as an alternating switch, with one seed going to the DRGB and the next seed going to the ENRNG. This construction ensures that a software application can *never* obtain the value used to seed the DRBG, nor can it initiate a Denial of Service attack against the DRBG through repeated executions of the RDSEED instruction.

The conditioner can be equated to the entropy pool in the cascade construction RNG described previously. However, since it is fed by a high-quality, high-speed, continuous stream of entropy that is fed faster than downstream processes can consume, it does not need to maintain an entropy pool. Instead, it is always conditioning fresh entropy independent of past and future entropy.

<u>The final two stages are:</u>

1. A hardware CSPRNG that is based on AES in CTR mode and is compliant with SP800-90A. In SP800-90A terminology, this is referred to as a DRBG, a term used throughout the remainder of this document.
2. An ENRNG that is compliant with SP800-90B and C.

Entropy Source (ES)

The all-digital Entropy Source also known as a non-deterministic random bit generator (NRBG), provides a serial stream of entropic data in the form of zeroes and ones.

The ES asynchronously runs on a circuit which is self-timed and uses thermal noise to output a random stream of bits at the rate of 3 Giga Hertz. The Entropy Source does not need any dedicated external power supply. The Entropy Source is designed to function properly over a wide range of operating conditions, exceeding the normal operating range of the processor.

Bits from the ES are passed to the conditioner for further processing.

The Deterministic Random Bit Generator

The primary role of this generator is to spread a conditioned entropy sample into a large set of random values, thus increasing the amount of random numbers available by the hardware module. This is done by employing a standards-compliant DRBG and continuously reseeding it with the conditioned entropy samples.

The DRBG chosen for this function is the CTR_DRBG defined in section 10.2.1 of NIST SP 800-90A (6), using the AES block cipher. Values that are produced fill a FIFO output buffer that is then used in responding to RDRAND requests for random numbers.

The DRBG autonomously decides when it needs to be reseeded to refresh the random number pool in the buffer and is both unpredictable and transparent to the RDRAND caller. An upper bound of 511 128-bit samples will be generated per seed. That is, no more than 511*2=1022 sequential DRNG random numbers will be generated from the same seed value.

Enhanced Non-deterministic Random Number Generator

The role of the enhanced non-deterministic random number generator is to make conditioned entropy samples directly available to software for use as seeds to other software-based DRBGs.

Values coming out of the ENRNG have multiplicative brute-forceprediction resistance, which means that samples can be concatenated and the brute-force prediction resistance will scale with them. When two 64-bit samples are concatenated together, the resulting 128-bit value will have 128 bits of brute-force prediction resistance ($2^{64} * 2^{64} = 2^{128}$). This operation can be repeated indefinitely and can be used to easily produce random seeds of arbitrary size. Because of this property, these values can be used to seed a DRBG of any size.

Robustness and Self-Validation

To ensure the DRNG functions with a high degree of reliability and robustness, validation features have been included that operate in an ongoing manner at system startup. These include the DRNG Online Health Tests (OHTs) and Built-In Self Tests (BISTs), respectively. Both are shown.

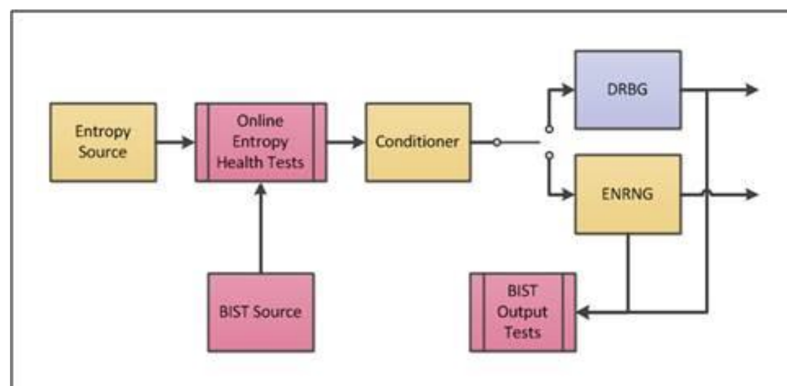


Figure 3 : Robustness and Self-Validation

Online Health Tests (OHTs)

Online Health Tests (OHTs) are designed to measure the quality of entropy generated by the ES using both per sample and sliding window statistical tests in hardware.

Per sample tests compare bit patterns against expected pattern arrival distributions as specified by a mathematical model of the ES. An ES sample that fails this test is marked "unhealthy." Using this distinction, the conditioner can ensure that at least two healthy samples are mixed into each seed. This defends against hardware attacks that might seek to reduce the entropic content of the ES output.

Sliding window tests look at sample health across many samples to verify they remain above a required threshold. The sliding window size is large (65536 bits) and mechanisms ensure that the ES is operating correctly overall before it issues random numbers. In the rare event that the DRNG fails during runtime, it would cease to issue random numbers rather than issue poor quality random numbers.

2.2. Quantium based Random Number Generator

There are two primary sources of practical quantum mechanical randomness: thermal noise and quantum mechanics at the sub-atomic or atomic level. Quantum mechanics guesses that various physical phenomena, such as the nuclear decay of an atoms, are fundamentally random in nature and cannot exactly be predicted. And, as we live at a temperature above 0 Kelvin or the absolute zero, every single system has slight random variation; for instance, molecules of gasses constituting air constantly bounce off each other in a random fashion. This randomness is one of the quantum phenomenon and thus unpredictable.

Figure 4 : Quantium based Random Number Generator

Because the final state value of quantum events cannot in principle be computation, they are the perfect standard for random number generation. Some quantum phenomena used are as follows:

- **Shot noise**: A quantum noise source in electronic circuits. A simple instance can be a photodiode biased lamp shine. Arriving photons generate noise in the implemented circuit, according to the principle of uncertainty in quantum mechanics.
- **A nuclear decay radiation source:**
- **Photons traveling through a semi-transparent mirror**. It is a mutually exclusive event (reflection/transmission) are detected and clubbed together as '0' or '1' bit to represent values respectively.
- Signal amplification on the base of a reverse-biased transistor. The emitter of the transistor is saturated with electrons and once in a while they will pass through the band-gap and exit via the base of the transistor. This signal is then further amplified using a few more transistors and the result fed into a computer to represent zeros and ones.
- **Schmitt trigger.** In a degenerate optical parametric oscillator, the binary phase state selection due to spontaneous parametric down-conversion leading to the binary phase state selection.

First pointed out in 2001, and certified to the highest levels of entropy testing, Quantis delivers reliable randomness at rates up to 16 Mega bits per second. It is a family of random number generating hardware which use the random nature of quantum physics as a source of true randomness.

The product version in existence compatible with most platforms are:

1. USB device
2. PCI Express (PCIe) board

2.3. True Random Number Generator Using a Metastability-Based Quality Control

It is a true random number generator based on metastability that achieves high entropy and passes randomness tests. By measuring the metastable resolution time the generator meas ures the degree of randomness regardless of the output bits. The system computes the original r andom noise level at the time of metastability and tunes itself to a chieve a high probability of randomness. Dynamic control enables the system to respond to deterministic noise and a qual ifier module grades the individual metastable events to produce a high-entropy random bit-stream.

The grading module allows the user to trade off output bitrate with the quality of the bitstream. A fully integrated true random number generator was fabricated in a 0.13 mom bulk CMOS technology with an area of 0.145 mm^2.

2.4. True Random Number Generator based on compact chaotic oscillator

True Random Number Generator (TRNG) based on CMOS designed compact discrete-time chaotic oscillator is presented. The chaotic oscillator was designed using 3 transistors ap circuit in order to construct an approximate V shape characteristic (inverse tent map). Simulation of the chaotic oscillator was described and examined in terms of bifurcation diagr am and transient waveform to show that it has a desirable output and suitability for TRNG. The TRNG has been used a chaotic oscillator to generate a random signal and increase the random-ness of the output signal through a dual oscillator sampling method and XOR. The circuit was designed and simulated in 0.18µm CMOS technology with 1.8 voltage supply. Furthermore, it was tested to be functional for output bit rate 23 Mbps and passed all test methods in NIST suit standard. The proposed TRNG exposes a potential alternative in both compact and robust rand om bit sequence that suitable to various other applications in security.

2.5. A truly random number generator based on thermal noise

A simple circuit to generate truly random numbers, which is based on the thermal noise of the resistor, is presented, as well as some simulation results. The circuit can be fabricated using standard CMOS process.

2.6. Simple true random number generator for any semiconductor technology

True random number generators (TRNGs) are needed in cryptography for key generati on, in challenge response authentication procedures and for countermeasures against power an alysis attacks. Such true randomness requires utilizing random physical hardware effects. It is the goal to make the TRNG usable for different semiconductor technologies. This approach is based on ring oscillators with multiple taps in combination with a simple post processing by exclusive OR ambivalence (XOR) compression. Verifications with a test chip and several FPGA implementations showed that standard digital library elements and the digital design flow can be used without any constraints for compilation and special layout rules. A proper choice of sampling frequency and compression coefficient ensures a random output with an extremely low bias for different technologies which can be checked online easily. It was show n that for passing the online test with a given bias limit the generated random data passes the statistical tests.

2.7. TRNG Based on ROPUF Circuit

The method of generating true random numbers utilizing the circuit primarily designed as PUF based on ring oscillators. The goal is to prove that it is possible to design the universal cryptosystem, that can be used for various applications the PUF can be utilized for asymmetric cryptography and generating asymmetric keys, TRNG for symmetric cryptography, nonce's and salts.

2.8. A TRNG algorithm from digital camera image noise for varying lighting conditions

This True Random Number Generator (TRNG) using the images taken by the web or mobile phone cameras. The three RGB color channels to obtain the random numbers whereas previous studies used only one. The algorithm excludes each pixel's saturated values to get its unbiased bits. An additional transposing operation shuffles the raw sequence to achieve better randomness.

The final sequence passes all of the NIST randomness tests. The algorithm involves very few calculations and is especially suitable for smartphones. With modern mobile cameras, it can work on the go and achieve a fast bit rate. With readily available commodity hardware with no hardware changes, we observe a random number generate a rate of 60 Mbps.

2.9 The Mersenne Twister

The Mersenne Twister is a pseudo random number generator (PRNG). It is by far the most widely used general purpose PRNG. Its name derives from the fact that its period length is chosen to be a Mersenne prime.

The Mersenne Twister was developed in 1997 by Makoto Matsumoto and Takuji Nishimura. It was designed specifically to rectify most of the flaws found in older PRNGs. It was the first PRNG to provide fast generation of high-quality pseudorandom integers.

The most commonly used version of the Mersenne Twister algorithm is based on the Mersenne prime 219937−1. The standard implementation of that, MT19937, uses a 32-bitword length. There is another implementation that uses a 64-bit word length, MT19937-64; it generates a different sequence.

Advantages

The commonly used version of Mersenne Twister, MT19937, which produces a sequence of 32-bit integers, has the following desirable properties:

1. 1.It has a very long period of 219937 − 1. While a long period is not a guarantee of quality in a random number generator, short periods can be problematic.
2. 2.It is k-distributed to 32-bit accuracy for every $1 \leq k \leq 623$ (see definition below).
3. 3.It passes numerous tests for statistical randomness, including the Diehard tests.

Disadvantages

The large state space comes with a performance cost: the 2.5 KiB state buffer will place a load on the memory caches. In 2011, Saito & Matsumoto proposed a version of the Mersenne Twister to address this issue. The tiny version, TinyMT, uses just 127 bits of state space.

By today's standards, the Mersenne Twister is somewhat slow unless the SFMT implementation is used . It passes most, but not all, of the stringent TestU01 randomness tests. Multiple Mersenne Twister instances that differ only in seed value (but not other parameters) are not generally appropriate for Monte Carlo simulations that require independent random number generators, though there exists a method for choosing multiple sets of parameters.

It can take a long time to start generating output that passes randomness tests, if the initial state is highly non random particularly if the initial state has many zeros. A consequence of this is that two instances of the generator, started with initial states that are almost the same, will usually output nearly the same sequence for many iterations, before eventually diverging. The 2002 update to the MT algorithm has improved initialization, so that beginning with such a state is very unlikely.

2.10 Yarrow algorithm

The Yarrow algorithm is a family of cryptographic pseudorandom number generators devised by John Kelsey, Bruce Schneier and Niels Ferguson. The Yarrow algorithm is explicitly unpat ented,royalty free and open source; no license is required to use it. Yarrow is incorporated in iOS and Mac OS X for their /dev/random devices, as did FreeBSD in the past.

An improved design from Ferguson and Schneier, Fortuna, is described in their book, Practical Cryptography, and FreeBSD has now moved to using this.

Principles

One of the most important principles of Yarrow is to make a PRNG that is better at resisting real world attack. The former widely used designs such as ANSI X9.17, RASREF 2.0 PRNG, have loopholes that provide attackers opportunities under some circumstances. Some of the m are not intentionally designed to face real world attacks. Another principle of Yarrow is that system designers with little knowledge about how the PRNG works can incorporate it into their own real world product fairly easily.

Components

The design of Yarrow consists of four major components including an entropy a ccumulator , reseed mechanism, generation mechanism and reseed control.

Yarrow accumulates entropy into two pools: the fast pool, which provides frequent reseeds of the key to keep the duration of key compromises as short as possible;the slow pool, which provides rare but conservative reseeds of the key. This makes sure that the reseed is secured even when the entropy estimates are very very optimistic in nature.

The reseed mechanism connects the entropy accumulator to the generating mechanism .Reseeding from the fast pool uses the current key and the hash of all inputs to the fast pool since startup to generate a new key; reseeding from the slow pool behaves similarly, except it also uses the hash of all inputs to the slow pool to generate a new key. Both of the reseedings reset the entropy estimation of the fast pool to zero, but the last one also sets the estimation of the slow pool to zero. The reseeding mechanism updates the key constantly, so that even if the key of pool information is known to the attacker before the reseed, they will be unknown to the attacker after the reseed.

The reseed control component is leveraging between frequent reseeding, which is desirable but might allowiterative guessing attacks, and infrequent reseeding, which compromises more information for an attacker who has the key.Yarrow uses the fast pool to reseed whenever the source passes some threshold values, and uses the slow pool to reseed whenever at least two of its sources pass some other threshold value. The specific threshold values are mentioned in the Yarrow-160 section.

Generation

Yarrow160 uses threekey tripleDES in counter mode to generate outputs. C is an nbit counter value; K is the key. In order to generate the next output block, Yarrow follows the functions shown here.

Yarrow keeps count of the output block, because once the key is compromised, the leak of the old output before the compromised one can be stopped immediately. Once some system security parameter Pg is reached, the algorithm will generate k bits of PRNG output and use them as the new key. In Yarrow160, the system security parameter is set to be 10, which means

Pg = 10. The parameter is intentionally set to be low to minimize the number of outputs that can be backtracked.

Reseed

The reseed mechanism of Yarrow160 uses SHA1 and tripleDES as the hash function and block cipher. The details steps are in the original paper.

Implementation of Yarrow-160

Yarrow160 can be implemented in Java, and FreeBSD. The examples can be found in "An imp lementation of the Yarrow PRNG for FreeBSD" by Mark R. V. Murray.

Pros and cons of Yarrow

•Yarrow reuses existing building blocks.
•Compared to previous PRNGs, Yarrow is reasonably efficient.
•Yarrow can be used by programmers with no cryptography background in a reasonably secure way. Yarrow is portable and precisely defined. The interface is simple and clear. These features somewhat decrease the chances of implementation errors.
•Yarrow was created using an attack-oriented design process.
•The entropy estimation of Yarrow is very conservative, thus preventing exhaustive search attacks. It is very common that PRNGs fail in real world applications due to entropy overestimation and guessable starting points.
•The reseeding process of Yarrow is relatively computationally expensive, thus the cost of attempting to guess the PRNG's key is higher.
•Yarrow uses functions to simplify the management of seed files, thus the files are constantly updated.
•To handle cryptanalytic attacks, Yarrow is designed to be based on a block cipher that is secured. The level of security of the generation mechanism depends on the block cipher.
•It tries to avoid data dependent execution paths. This is done to prevent side channel attacks such as timing attacks and power analysis. This is an improvement compared to earlier PRNGs, for example RSAREF 2.0 PRNG, that will completely fall apart once additional information about the internal operations are no longer secured in nature.

•Yarrow uses cryptographic hash functions to process input samples, and then uses a secure up date function to combine the samples with the existing key. This makes sure that the attacker cannot easily manipulate the input samples. PRNGs such as RSAREF 2.0 PRNG do not have the ability to resist this kind of chosen-input attack.
•Unlike ANSI X9.17 PRNG, Yarrow has the ability to recover from a key compromise. This m eans that even when the key is compromised, the attacker will not be able to predict future outputs forever. This is due to the reseeding mechanism of Yarrow.
•Yarrow has the entropy samples pool separated from the key, and only reseeds the key when t he entropy pool content is completely unpredictable. This design prevents iterative guessing at tacks, where an attacker with the key guess the next sample and checks the result by observing the next output.

Cons

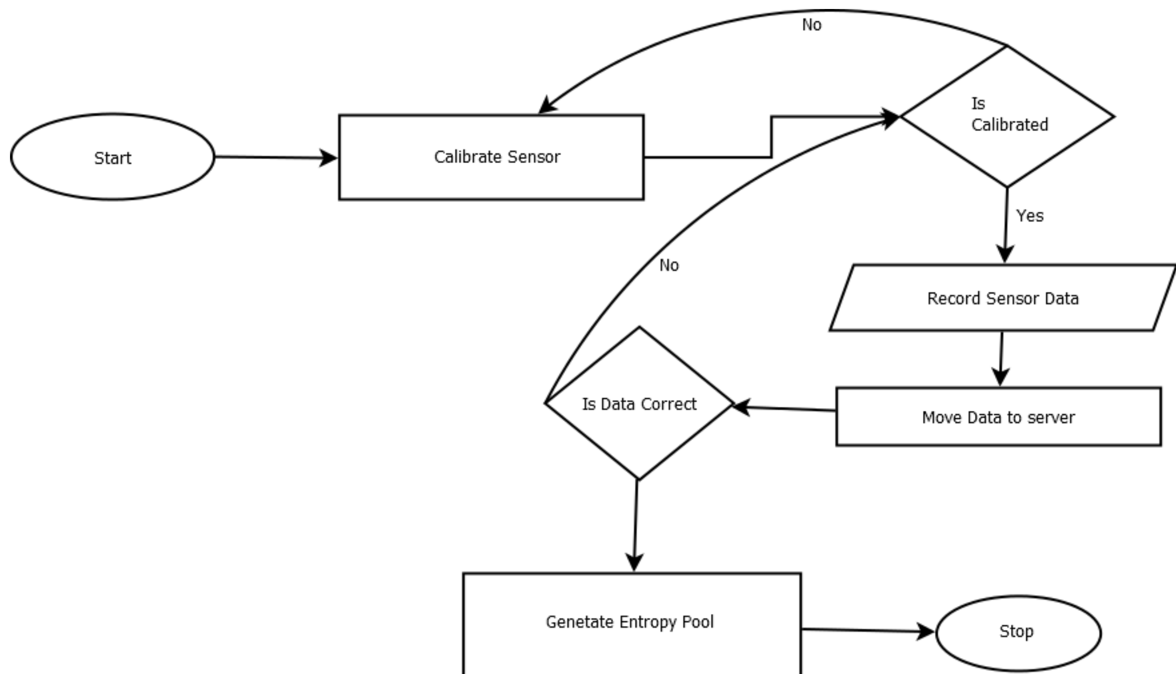
•Since the outputs of Yarrow are cryptographically derived, the systems that use those outputs can only be as secure as the generation mechanism itself. That means the attacker who can break the generation mechanism will easily break a system that depends on Yarrow's outputs. This problem cannot be solved by increasing entropy accumulation.
•Yarrow requires entropy estimation, which is a very big challenge for implementation. It is hard to be sure how much entropy to collect before using it to reseed the PRNG. This problem is solved by Fortuna (PRNG), an improvement of Yarrow. Fortuna has 32 pools to collect entropy and removed the entropy estimator completely.
•Yarrow's strength is limited by the size of the key. For instance, Yarrow160 has an effective key size of 160 bits. If the security requires 256 bits, Yarrow160 is not capable of doing the job

Chapter-3 SYSTEM DEVELOPMENT

The System development of "Pure Random Number Generator are as follows;

3.1 Broad-Scale Distribution of Working Process

The process starts with data collection from the mobile sensors. Then the data is recorded on the mobile device in csv format and then moves over to the server for analysis and processing. Once the processing is done the data is moved to the entropy pool, which is used as the source for the random number generation. the flowchart below on this page depicts the flow of data from mobile sensors to the entropy pool.



3.1.1 Step1: Data Collection

An android application is used to read the values returned from the sensor of a mobile handset.

The program then segregates and transforms the data into a comma separated values (.csv) format. And stores it over the mobile storage on each cellular unit.

3.1.2 Step2: Moving Database to Server

This data set is then passed on to the A MySQL Server for processing. The data will be processed and passed as requested by the user.

3.1.3 Step3: Extraction of meaningful information

PHP is used and a server side language. The algorithm on the server side splits the decimal sensor value and takes 3 to 7 the digit making it the random number for that sensor at that point of time.

3.1.4 Step4: Range Optimization

This random number is processed by a ranged algorithm that forces it to lie in a given interval keeping the unpredictability intact. Now this generated random number is made to fall on the graph and results are shown as below The code for each shall be included in the last section of the report.

3.2 Modular Distribution

3.2.1 MODULE 1: Android Application for Sensor Data Collection

Each mobile nowadays is equipped with some sensors even if it ranges as low as 3000 bucks. The android application uses specific classes to extract data from the handset's Sensor.

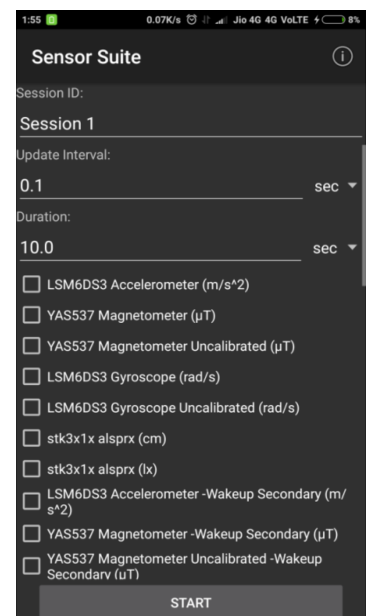


Figure 5 : Android App UI

MainActivity.java

```java
package simplicial.software.sensor_suite.application;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.app.AlertDialog.Builder;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Build.VERSION;
import android.os.Bundle;
import android.os.Environment;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import simplicial.software.sensor_suite.models.b;
import simplicial.software.sensor_suite.models.l;
import simplicial.software.sensor_suite.models.o;
import simplicial.software.sensor_suite.models.s;
```

Figure 6 : Android code snippit

The Android Application asks for the sampling rated from the user in seconds and the data recording interval. It also gives a list of sensors in the form of a checklist of sensors to be chosen for data recording. This session is recorded corresponding to a specific session ID.

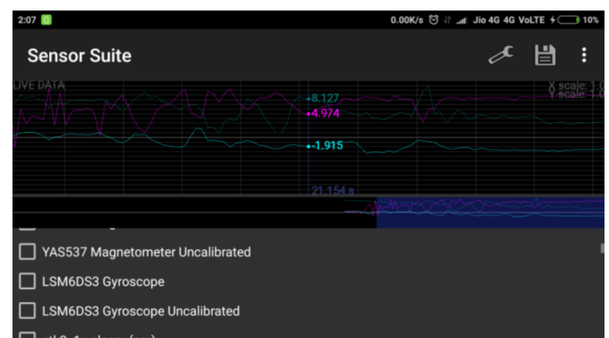


Figure 7 : Android App Live Graph

3.2.2 MODULE 2: Recording Data Using SQLite

What is SQLite?

SQLite is open source Structured Query Language database that stores values to a text file on any device. Android devices already comes in with built in SQLite. It supports all the

RDBMS features. In order to access this database, you do not need any kind of connections for it like JDBC,ODBC etc

Database - Package

The main package is "***android.database.sqlite*"**. The package contains the classes to manage your databases in form of tables.

Elapsed Time (seconds): 20.0		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
		NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Sensor	Timestamp (seconds)	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Primary Unit	
stk3x1x alsprx	-3.986483887	80.0	n/a	n/a	n/a	n/a	n/a	lx	
AMD	-0.1292146	2.0	0.0	0.0	0.0	0.0	0.0		
YAS537 Magnetometer	0.022	7.83538818359375	-47.92633056640625	17.93670654296875	n/a	n/a	n/a	µT	
YAS537 Magnetometer Uncalibrated	0.022	168.87054443359375	-56.1004638671875	-75.30059814453125	161.03515625	-8.17413330078125	-93.237304687	µT	
LSM6DS3 Accelerometer	0.023770019	1.093994140625	6.32257080078125	6.5353546142578125	n/a	n/a	n/a	m/s^2	
YAS537 Magnetometer	0.032040283	6.4453125	-47.92633056640625	17.93670654296875	n/a	n/a	n/a	µT	
YAS537 Magnetometer Uncalibrated	0.032040283	167.48046875	-56.1004638671875	-75.30059814453125	161.03515625	-8.17413330078125	-93.237304687	µT	
LSM6DS3 Accelerometer	0.033413574	1.093994140625	6.32257080078125	6.5353546142578125	n/a	n/a	n/a	m/s^2	
LSM6DS3 Accelerometer -Wakeup Secondary	0.033413574	1.093994140625	6.32257080078125	6.5353546142578125	n/a	n/a	n/a	m/s^2	
YAS537 Magnetometer	0.042111084	6.2744140625	-49.12567138671875	18.83697509765625	n/a	n/a	n/a	µT	
YAS537 Magnetometer Uncalibrated	0.042111084	167.3095703125	-57.2998046875	-74.40032958984375	161.03515625	-8.17413330078125	-93.237304687	µT	
YAS537 Magnetometer -Wakeup Secondary	0.042111084	6.2744140625	-49.12567138671875	18.83697509765625	n/a	n/a	n/a	µT	
YAS537 Magnetometer Uncalibrated	0.042111084	167.3095703125	-57.2998046875	-74.40032958984375	161.03515625	-8.17413330078125	-93.237304687	µT	

Figure 8 : Sampling Database Screenshot

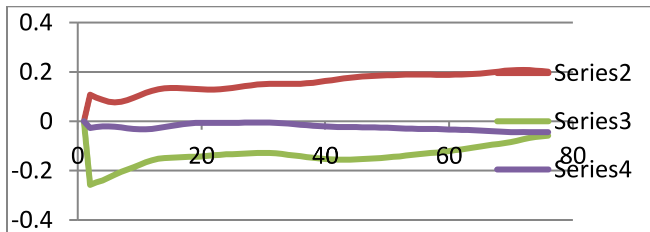
After completion of working of this module the data is stored locally to the device and is ready to be transferred to server for processing.

3.2.3 MODULE 3: Server-Side Scripting for extraction of meaningful information.

Once the server receives the data it processes it is ready to be process.

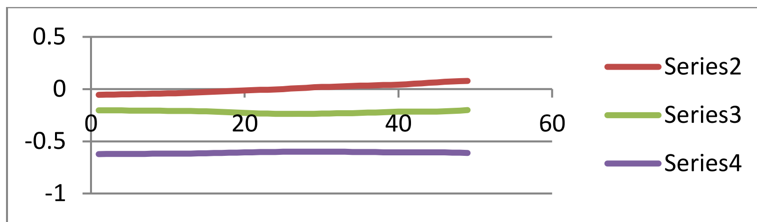
Instances of Raw data is as follows.

1. Game Rotation Vector [While Climbing Stairs]



Graph 1 : Raw sensor data of Game Rotation Vector [While Climbing Stairs]

2. GeoMagnetic Rotation Vector [While Walking]



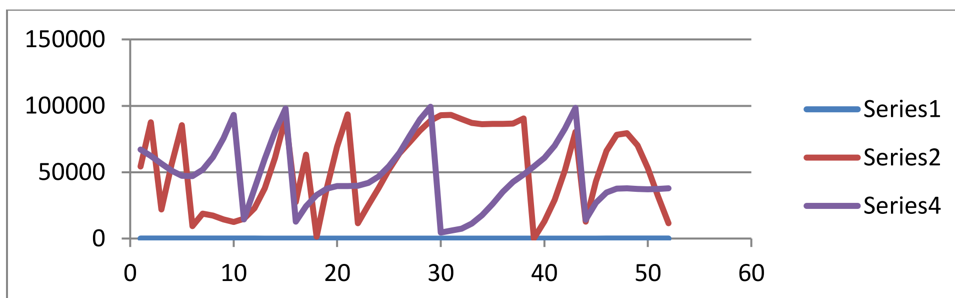
Graph 1 : Raw sensor data of GeoMagnetic Rotation Vector [While Walking]

As we can observe that though this data is not predictable in any sense but it is not scattered in a range and has some of the other pattern . This tells us that the raw data needs to be processed more than this.

```
if($datatype=="Processed")
{
    list($p1, $p2) = explode(".", $row[$loop_1]);
    $digits = substr($p2, 2, 5);
    echo $digits;
}
else
```

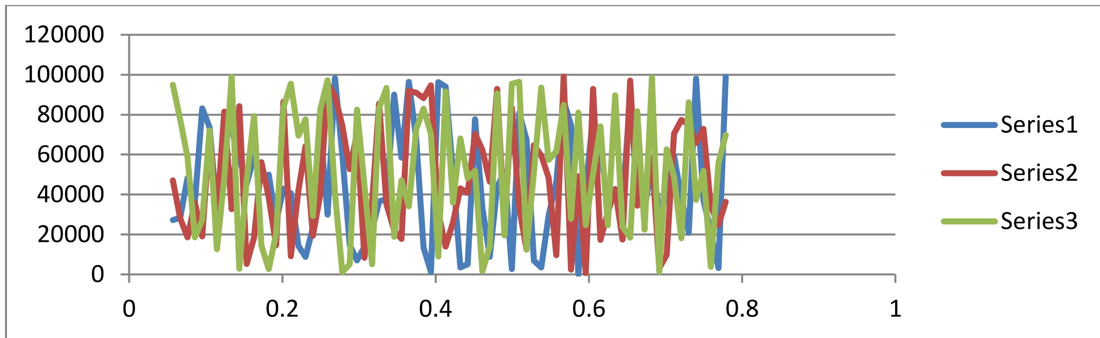
The above codes stripes the numeric data and take some of the digits. Lets observe the behavior and pattern of the data after processing.

1. Game Rotation Vector [While Climbing Stairs]



Graph 3 : Processed data of Game Rotation Vector [While Climbing Stairs]

2. GeoMagnetic Rotation Vector [While Walking]



Graph 4 : Processed sensor data of GeoMagnetic Rotation Vector [While Walking]

Similar results were obtained while walking running and stationary device. and thus can be used for to fill the entropy pool.

3.2.4 MODULE 4: Entropy pool generator

Entropy pool helps us to work in offline mode i.e. when sensors are offline.Entropy generator picks up the values from sensor database lists it in a table.

```
$query_2="INSERT INTO pool (sno,value) VALUES ($pool_count,'$digits')";
$digits!=""?mysql_query($query_2):$pool_count--;
```

Showing rows 0 - 24 (3858 total, Query took 0.0000 seconds.)

	sno	value
ete	1	75025
ete	2	76718
ete	3	75068
ete	4	75025
ete	5	75073
ete	6	35828
ete	7	64196
ete	8	98507
ete	9	01916
ete	10	93538

This entropy pool is used to get random number as per requirement. The count flag keeps a count of used and used data.

	flag	value
Edit Copy Delete	count	132

3.2.5 MODULE 5: Range optimization

The purpose of this module is to generate the value under a give interval.

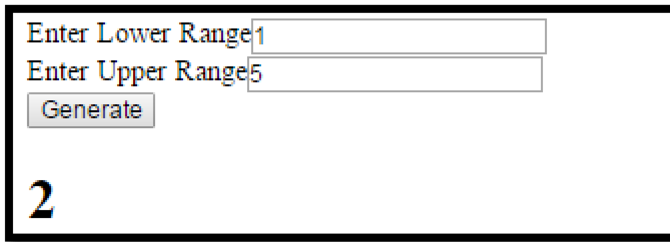


Figure 9 : Screenshot of range optimizer UI

The above interface takes the upper range and lower range, optimizes the seed within the range and displays the output.

The code snippet shows a how the number has been manipulated.

```
$rand_seed=intval($row[1]);
$normalizer=$rand_seed%($max-$min+1);
$number=$min+$normalizer;

echo $number;
```

Chapter-4 PERFORMANCE ANALYSIS

4.1 Algorithmic Complexity

All the algorithms used in any module of the project is O(n) for n random numbers. This shows the random number generating algorithms are not CPU intensive. And as the server is multithread it can cater to large number of requests at the same time. The feature out rules the drawback of low data rate of pre-existing Pure Random number generators

4.2 Resource Utilization at Runtime

This stress testing tells us the processor requirement of entropy collection algorithm when implemented in a mobile device under load. Here we can see that 20% of Ram is required to collect data for 10 seconds at very high sampling rate of 0.002 sec/sample. Data collection for 10 seconds gives a set of 4000 random numbers.

Figure 10 : Resource Utilization at Runtime

4.3 Applications and Contributions

4.3.1 Caesar Cipher

In cryptography, a Caesar cipher, also known as shift cipher, Caesar's cipher, Caesar's code or Caesar shift, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is shifted' a certain number of places down the alphabet. For instance, with a shift of 1, A would be replaced by B, B would become C, and so on. The method is named after Julius Caesar, who apparently used it to communicate with his generals.

More complex encryption schemes such as the vigenere cipher employ the Caesar cipher as one element of the encryption process. The widely known ROT13 encryption' is simply a Caesar cipher with an offset of 13. As with all single-alphabet substitution ciphers, the Caesar cipheris easily broken and in modern practice offers essentially no communication security.

The encryption of Caesar cipher can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,..., Z = 25. Encryption of a letter x by a shift n can be described mathematically as,

$$E_n(x) = (x + n) \mod 26.$$

Decryption is performed similarly,

$$D_n(x) = (x - n) \mod 26.$$

Instance:

To pass an encrypted message from one person to another, it is first necessary that both parties have the key' for the cipher, so that the sender may encrypt it and the receiver may decrypt it. For the Caesar cipher, the key is the number of characters to shift the cipher alphabet.

Here is a quick instance for the encryption and decryption of Caesar cipher. The text we will encrypt is 'cryptography', with a shift (key) of 3.

Plain Text	c	r	y	p	t	o	g	r	a	p	h	y
Alphabet Number + Key	2 + 3	17 + 3	24 + 3	15 + 3	19 + 3	14 + 3	6 + 3	17 + 3	0 + 3	15 + 3	7 + 3	24 + 3
Cipher Text	f	u	b	s	w	r	j	u	d	s	k	b

Decryption is just as easy, by using an offset of -3.

Cipher Text	f	u	b	s	w	r	j	u	d	s	k	b
Alphabet Number - Key	5 - 3	20 - 3	1 - 3	18 - 3	22 - 3	17 - 3	9 - 3	20 - 3	3 - 3	18 - 3	10 - 3	1 - 3
Plain Text	c	r	y	p	t	o	g	r	a	p	h	y

Security:

Caesar cipher is not a secure cryptosystem because there are only 26 possible keys to try out, we can simply try each possibility and see which one results in a piece of readable text. If you happen to know what a piece of the ciphertext is, or you can guess a piece, then this will allow you to immediately find the key.

If this is not possible, a more systematic approach is to match up the frequency distribution of the letters. By graphing the frequencies of letters in the ciphertext, and by knowing the expected distribution of those letters in the original language of the plaintext, a human can easily spot the value of the shift by looking at the displacement of particular features of the graph. his is known as frequency analysis. For instance, in the English language the plaintext frequencies of the letters E, T, (usually most frequent), and Q, Z (typically least frequ ent) are particularly distinctive.

Implementation:

We can use the randomly generated value as encryption key for Caesar cipher or and other similar encryption algorithm.

4.3.2 Games

Unpredictable were first investigated in the context of gambling developing, sometimes, pathological forms like apophenia. Many randomizing devices such as dice, shuffling playing cards wheels seem to have been developed for use in games of chance.Electronic gambling equipmentcannot use these and so theoretical problems are less easy to avoid; methods of creating them are sometimes regulated by governmental gaming commissions.

Modern electronic casino games contain often one or more random number generators which decide the outcome of a trial in the game. Even in modern slot machines, where mechanical reels seem to spin on the screen, the reels are actually spinning for entertainment value only. They eventually stop exactly where the machine's software decided they would stop when the handle was first pulled. It has been alleged that some gaming machines' software is deliberately biased to prevent true randomness, in the interests of maximizing their owners' revenue; the history of biased machines in the gambling industry is the reason government insp ectors
attempt to supervise the machines electronic equipment has extended the range of supervision. Some thefts from casinos have used clever modifications of internal software to bias the outco mes of the machines at least inthose which have been discovered. Gambling establishments keep close track of machine payouts in an attempt to detect such alterations. Random draws are often used to make a decision where no rational or fair basis exists for making a deterministic decision.

4.3.3 Science

Many methods of statistical analysis, such as the bootstrap method, require random numbers. Monte Carlo methods in physics and computer science require random numbers.

Random numbers are often used in parapsychology as a test of precognition.

Statisticalsampling

Statistical practice is based on statistical theory which is, itself, founded on the concept of randomness. Many elements of statistical practice depend on randomness via random numbers. Where those random numbers fail to be actually random, any subsequent statistical analysis may suffer from systematic bias. Elements of statistical practice that depend on randomness include: choosing a representative sample of the population being examined, disguising the protocol of a study from a participant (see randomized controlled trial) and Monte Carlo simulation.

These applications are useful in auditing (for determining samples such as invoices) and experimental design (for instance in the creation of double-blind trials).

Analysis

Many experiments in physics rely on a statistical analysis of their output. For instance, an experiment might collect Xrays from an astronomical source and then analyze the result for periodic signals. Since random noise can be expected to appear to have faint periodic signals embedded in it, statistical analysis is required to determine the likelihood that a detected signal actually represents a genuine signal. Such analysis methods requires the generation of random numbers. If the statistical method is extremely sensitive to patterns in the data (such as those used to search for binary pulsars), very large amounts of data with no recognizable pattern are needed.

Simulation

In many scientific and engineering fields, computer simulations of real phenomena are commonly used. When the real phenomena are affected by unpredictable processes, such as radio noise or day-to-day weather, these processes can be simulated using random or pseudo-random numbers.

Pseudo random numbers are frequently used in simulation of statistical events, a very simple instance being the outcome of tossing a coin. More complicated situations are simulation of population genetics, or the behavior of subatomic particles. Such simulation methods, often called stochastic methods, have many applications in computer simulation of real-world processes.

Chapter-5 Conclusion

5.1 Conclusions

The above developed model successfully generates pure random numbers for any give finite range. It is not fully dependent on online connectivity with the physical phenomenon i.e. random numbers can be generated at any point of time. The model also shows the ability to use these random number as per requirements. The model also over comes the major drawback of low bit rate with no extra load on the processor as the algorithm used is O(1).

5.3 References

1. Intel® Digital Random Number Generator (DRNG)
Reference URL: https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide.
Revision 2.0 : May 15, 2014

Accessed on: September 7th, 2016

2. Quantis Random Number Generator
Reference URL: http://www.idquantique.com/random-number-generation/quantis-random-number-generator/
Data of Issue: 2015-01-29
Accessed on: September 7th, 2016

3. Carlos Tokunaga; David Blaauw; Trevor Mudge, "True Random Number Generator With a Metastability-Based Quality Control", IEEE Journal of Solid-State Circuits, Volume: 43, Issue: 1, Pages: 78 - 85,

4. Huang Zhun; Chen Hongyi, "A truly random number generator based on thermal noise" ASICON 2001. 2001 4th International Conference on ASIC Proceedings, Volume: 35, Issue: 1, Pages: 862 - 864, Year: 2001

5. Simona Buchovecká; Róbert Lórencz; Filip Kodýtek; Jirí Bucek, " True Random Number Generator Based on ROPUF Circuit", 2016 Euromicro Conference on Digital System Design (DSD), Volume: 31, Issue: 1, Pages: 519 - 523, Year: 2016

6. C.S. Petrie, J.A. Connelly, "noise-based random bit generator IC for applications in cryptography", Proceedings of the 1998 IEEE International Symposium on Circuits and Systems, Cat. No.98CH36187, Year: 1998

Chapter -6. Appendix

6.1 Android Application [MainActivity.java]

```
 package simplicial.software.sensor_suite.application;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.app.AlertDialog.Builder;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Build.VERSION;
import android.os.Bundle;
import android.os.Environment;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import simplicial.software.sensor_suite.models.b;
import simplicial.software.sensor_suite.models.l;
import simplicial.software.sensor_suite.models.o;
import simplicial.software.sensor_suite.models.s;

public class MainActivity
  extends Activity
  implements ak
{
  public List a = new ArrayList();
  public double b = 0.1D;
  public s c = null;
  public l d = null;

  public void a()
  {
    if (this.d != null) {
      this.d.a(System.currentTimeMillis());
    }
  }

  protected void onCreate(Bundle paramBundle)
  {
    simplicial.software.sensor_suite.models.a.b().a(this);
    o.a = new o(this);
    super.onCreate(paramBundle);
    setContentView(2130903040);
    if (paramBundle == null) {
      getFragmentManager().beginTransaction().add(2131296256, new p()).commit();
    }
  }

  public boolean onCreateOptionsMenu(Menu paramMenu)
```

```
{
  getMenuInflater().inflate(2131230720, paramMenu);
  return true;
}

protected void onDestroy()
{
  simplicial.software.sensor_suite.models.a.b().b(this);
  super.onDestroy();
}

@SuppressLint({"InlinedApi", "NewApi"})
public boolean onOptionsItemSelected(MenuItem paramMenuItem)
{
  Object localObject;
  switch (paramMenuItem.getItemId())
  {
  default:
  case 2131296296:
  case 2131296299:
  case 2131296298:
    for (;;)
    {"
      return super.onOptionsItemSelected(paramMenuItem);
      if (findViewById(2131296258).getVisibility() != 8) {
        findViewById(2131296258).setVisibility(8);
      }
      for (;;)
      {
        return true;
        findViewById(2131296258).setVisibility(0);
      }
      getFragmentManager().beginTransaction().replace(2131296256, new u(al.a(this))).addToBackStack(null).commit();
      return true;
      if (this.c == null) {
        break;
      }
      b.a(this, this.c);
    }
    if (!simplicial.software.a.a.a.a())
    {"
      paramMenuItem = new AlertDialog.Builder(this);
      paramMenuItem.setTitle("Error");
      paramMenuItem.setMessage("External storage is not writeable.");
      paramMenuItem.show();
      return false;
    }
    if (Build.VERSION.SDK_INT >= 19) {}
    for (localObject = new File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOCUMENTS) + "/Exported Sensor
Data/");; localObject = new File(Environment.getExternalStorageDirectory() + "/Exported Sensor Data/"))
    {
      ((File)localObject).mkdirs();
      localObject = new File((File)localObject, "sensor_data.db");
      try
      {
        simplicial.software.a.a.a.a(getDatabasePath("sensor_data.db"), (File)localObject);
```

```
        AlertDialog.Builder localBuilder = new AlertDialog.Builder(this);
        localBuilder.setTitle("Exported");
        localBuilder.setMessage("Database copied to:\n" + localObject);
        localBuilder.show();
      }
      catch (IOException paramMenuItem)
      {
        localObject = new AlertDialog.Builder(this);
        ((AlertDialog.Builder)localObject).setTitle("Error");
        ((AlertDialog.Builder)localObject).setMessage(paramMenuItem.getMessage());
        ((AlertDialog.Builder)localObject).show();
        return false;
      }
    }
  case 2131296297:
    new ac(this).show(getFragmentManager(), null);
    return true;
  case 2131296300:
    getFragmentManager().beginTransaction().replace(2131296256, new ag()).addToBackStack(null).commit();
    return true;
  }
  int i = 0;
  for (;;)
  {
    if (i >= getFragmentManager().getBackStackEntryCount()) {
      return true;
    }
    getFragmentManager().popBackStack();
    i += 1;
  }
 }
}
```

6.2 Entopy Pool Developer

```
<?php

		set_time_limit(1000);

		$host="localhost";
		$username="root";
		$password="";
		$database="project";


		$sensor=array(		"Game Rotation Vector",								"Game
Rotation Vector -Wakeup Secondary",
		"GeoMagnetic Rotation Vector",
		"GeoMagnetic Rotation Vector -Wakeup Secondary",
		"Gravity","Gravity -Wakeup Secondary",
		"Linear Acceleration",
		"Linear Acceleration -Wakeup Secondary",
		"LSM6DS3 Accelerometer",
		"LSM6DS3 Accelerometer -Wakeup Secondary",
		"LSM6DS3 Gyroscope",
		"LSM6DS3 Gyroscope -Wakeup Secondary",
```

```
"LSM6DS3 Gyroscope Uncalibrated",
"LSM6DS3 Gyroscope Uncalibrated -Wakeup Secondary",
"Motion Acce",
"Orientation",
"Orientation -Wakeup Secondary",
"Rotation Vector",
"Rotation Vector -Wakeup Secondary",
"SensorTimestamp (seconds)",
 "Step Counter",
"Step Counter -Wakeup Secondary",
"stk3x1x alsprx",
"stk3x1x alsprx -Non Wakeup Secondary",
"stk3x1x alsprx -Wakeup Secondary",
"YAS537 Magnetometer",
"YAS537 Magnetometer -Wakeup Secondary",
"YAS537 Magnetometer Uncalibrated",
"YAS537 Magnetometer Uncalibrated -Wakeup Secondary");


mysql_connect($host,$username,$password);
@mysql_select_db($database) or die( "Unable to select database");

$datatype="Processed";
$pool_count=0;

for($loop_2=0;$loop_2<=19;$loop_2++)
{
		$query="select * from test_4 where `COL 1`= '$sensor[$loop_2]'";
		$run=mysql_query($query);

		echo "<center><table border=\"01\" width=\"100%\">";
		while ($row=mysql_fetch_array($run))
		{
				echo "<tr>";
				for($loop_1=2;$loop_1<=4;$loop_1++)
				{
						echo "<td>";
						if($loop_1>1)
						{
								if($datatype=="Processed")
								{
										$pool_count++;
										list($p1, $p2) = explode(".", $row[$loop_1]);
										$digits = substr($p2, 2, 5);
										echo $digits;

										$query_2="INSERT INTO pool (sno,value) VALUES
						($pool_count,'$digits')";

										$digits!=""?mysql_query($query_2):$pool_count--;

								}
								else
								{
										echo $row[$loop_1];
																}
```

```
						}
					else
					{
						echo $row[$loop_1];
					}
					echo "</td>";
				}

				echo "</tr>";
			}
			echo "</table></center>";

			echo "</br></br>";

		}
?>
```

2.3 Range Optimizer

```
	$min= intval($_GET['min']);
	$max= intval($_GET['max']);

	if($min >= $max )
		echo " redefine range ";
	else
	{
		$host="localhost";
		$username="root";
		$password="";
		$database="project";
		mysql_connect($host,$username,$password);
		@mysql_select_db($database) or die( "Unable to select database");

		$query="select * from flag where 1";
		$run=mysql_query($query);

		$row=mysql_fetch_array($run);
		//echo $row[1];

		$flag=intval($row[1])+1;
		$flagg=(string)$flag;
		$sql = "UPDATE `flag` SET value = $flagg WHERE flag = 'count'";
		mysql_query($sql);

		$query="select * from pool where sno = $flagg";

		$run=mysql_query($query);

		$row=mysql_fetch_array($run);

		$rand_seed=intval($row[1]);
		$normalizer=$rand_seed%($max-$min+1);
		$number=$min+$normalizer;
```

```
            echo $number;

        }
?>
```

2.4 Cipher

```
        require_once('rand.php');

                $obj = new true_random();
                $rand_key = $obj->true_rand(1,26);

        function Cipher($ch, $key)
        {
                if (!ctype_alpha($ch))
                        return $ch;

                $offset = ord(ctype_upper($ch) ? 'A' : 'a');
                return chr(fmod(((ord($ch) + $key) - $offset), 26) + $offset);
        }

        function Encipher($input, $key)
        {
                $output = "";

                $inputArr = str_split($input);
                foreach ($inputArr as $ch)
                        $output .= Cipher($ch, $key);

                return $output;
        }

        function Decipher($input, $key)
                return Encipher($input, 26 - $key);

        $str="A b7jh*o";
        echo "Random Key : ".$rand_key."</br>";
        echo "String : ".$str."</br>";

        $enc = Encipher($str, $rand_key);
        echo "Encipher : ".$enc."</br>";

        $dec = Decipher($enc, $rand_key);
        echo "Decipher : ".$dec."</br>";
        ?>
```