# A Quasigroup Based Random Number Generator for Resource Constrained Environments

Matthew Battey
Computer Science
University of Nebraska at Omaha
Omaha, NE 68182

Abhishek Parakh
Nebraska University Center for Information Assurance
University of Nebraska at Omaha
Omaha, NE 68182

***Abstract* – This paper proposes a pseudo random number generator (PRNG) based on quasigroups. The proposed PRNG has low memory requirements, is autonomous and the quality of the output stream of random numbers is better than other available standard PRNG implementations (commercial and open source) in majority of the tests. Comparisons are done using the benchmark NIST Statistical Test Suite and compression tools. Results are presented for quality of raw stream of random numbers and for encryption results using these random numbers.**

***Index Terms* – cryptography, random number generation, quasigroups, efficient encryption**

## I. Introduction

Pseudo random number generators (PRNG) play an important role in computing and communication technology, whether it be for online gambling, reducing collisions on Ethernet networks, or securing data through cryptographic techniques.

A common application of PRNGs is to secure communication of large amounts of data, over insecure public channels. A common approach is to exchange PRNG seeds using a public key cryptosystems and then XOR the stream of data with the stream of pseudo random numbers produced from the PRNG. The receiver can invert the XOR operation as he can also generate the same stream of pseudo random numbers.

Such light weight encryption systems are of great advantage in low-powered environments such as sensor networks and mobile devices including smart phones and tablet computers. These can be used to encrypt large amounts of data being produced by these devices in the form of pictures, videos and teleconferencing. The proposed pseudo random number generator has low memory requirements, low computation requirements (if any at all – most operations are table look up) and has high throughput making it an ideal choice for such environments.

In the following we first discuss the PRNG choices available to programmers, followed by a primer on quasigroups. We then present the proposed algorithms and results.

Among programming languages, C, Java, C++, Objective-C, and C#.Net are the top five [1]. In all, these languages make use of only two different algorithms to generate pseudo random numbers, SHA-1 [2] [3] [4] [5] [6] and AES [3] [2], with a third (ARC4) available only on BSD derived operating systems [7]. While the C derived languages (C, C++, Obj-C) do not provide secure PRNG interfaces directly, common practice makes use of either OpenSSL's `rand` or the BSD's `arc4random`. Here, OpenSSL's `rand` makes use of SHA-1 [5], and `arc4random` uses an implementation of the ARC4 stream cipher.

The National Institute of Standards and Technology (NIST), a division of U.S. Department of Commerce, produces the Federal Information Processing Standards Publication (FIPS PUBS) [8]. While these focus on standards set for the United States Federal Government, they are adopted by every major software vendor. FIPS-180 specifies SHA-1 as the preferred algorithm for producing random numbers [9](versions 1-4). For this reason, the operating system vendors have chosen SHA-1 as part of the standard offering.

Microsoft Windows supplied the SHA-1 algorithm via the `CryptGenRandom` API in versions up to Windows Vista. From that point on, Microsoft began offering an AES "Counter Mode" (AES-CM) [10] algorithm through the same API [3]. The Microsoft.NET CLR virtual machine makes use of `CryptGenRandom` directly [2], hence both Win32 and Microsoft.NET applications will use either SHA-1 or AES-CM. Java followed suit, implementing the FIPS-180 standards SHA-1.

BSD is an outlier in a few ways. First, most UNIX like operating systems chose not to directly provide secure PRNGs other than the linear congruential generator found in libc. Instead they rely on third-party packages, such as OpenSSL's `rand`. Second, BSD provided `arc4random` in the C standard library which also made its way into Mac OS X, but again, OpenSSL is shipped as part of the Mac OS X distribution as well.

Although the United States government has adopted SHA-1 as the preferred mechanism for random number generation, both SHA-1 and RC4 (arc4random) have their limitations. It has been found that in the `CryptGenRandom` function, using the SHA-1 algorithm [11], up to 128 kilo-bytes of past

and future random data may be revealed with a non-trivial approach taking $2^{23}$ steps. Issues with the RC4 key stream generator have been noted for some time. Statistical biases have been found that may distinguish RC4 output from other random strings with only 225 bytes [12]. For this and other reasons, NIST chose to deprecate SHA-1, supposedly phasing out the hash algorithm's adoption by 2010 [13]. Although SHA-1 and RC4 are no longer considered secure, by July of 2012 they have not been faced out with the exception of the Microsoft Windows implementation, being operating system dependent [3].

In this paper we present a new pseudo random number generator, based on quasigroups, that is both stroage and computationally efficient. The quality of stream of random numbers produced is tested using the NIST Statistical Test Suite and compression methods. The results show that the proposed random number generator performs better in most cases and at par in other cases with the commercially available test suite. Further, the random number generator does not depend on any hashing algorithm or third party sub-algorithm and is completely self-contained and operating system independent.

## II. Background on Quasigroups and Quasigroup Stream Ciphers

Quasigroups are square matrix structures that support a forward and an inverse operation. The elements in the matrix are so arranged that no element repeats in any row or column and every element appears in every row and column. Quasigroup (QG) stream ciphers are cryptosystems based on these structures [14] [15]. In particular, we define a forward operation such that,

$$a \cdot b = c$$

and an inverse operation (via an inverse quasigroup [14]),

$$a / c = b$$

Here the elements on the left hand side of the above equations represent the column and row indices of the quasigroup matrix and the right hand side element is an element from the quasigroup matrix. The operations '·' and '/' produce the right hand side by lookup operation in the quasigroup matrix. The order of a quasigroup is the number of rows or columns it has. For example, a QG of size 256x256 has order 256 containing 64 K elements.

A quasigroup based stream cipher works as follows:

1. Choose a seed, $s$, in the same range as the QG's order.
2. The input plaintext, $M$, is broken into a stream of 8-bit words as $m_1, m_2, \ldots, m_n$. The corresponding cipher text, $C$, is a stream of 8-bit words $c_1, c_2, \ldots, c_n$, where,

$$c_1 = s \cdot m_1$$
$$c_2 = c_1 \cdot m_2$$
$$\ldots$$
$$c_n = c_{n-1} \cdot m_n$$

Quasigroups have also been used to construct block ciphers and when compared against competing systems such as AES, they perfomed better in almost all tests [14].

Gligoroski and Markovski explore the use of quasigroups for pseudo random number generation [16]. Markovski and Dimitrova looked at applying multiple QG stream cipher passes on an initial vector of a single value. Here, a string such as "111111111" is encrypted multiple times, until sufficiently random data is acquired. A minimum of two passes is necessary as a single pass reduces the algorithm to a single repeating permutation of size $n$ where $n$ is the order of the quasigroup. Markovski and Dimitrova refer to this as the period of the QG-PRNG [17], and suggest the period may be extended through each successive pass of the QG stream cipher. In a second study, Markovski, Gilgorski and Kocarev (MGK) suggest the application of a QG stream cipher to a non-uniform, but random source, making it unbiased [18]. Another implementation of QG PRNG linearizes the QG matrix in a raster like operation, Godavarty [19]. The original QG is kept, and the linearized values are fed through the QG stream cipher. The output values are rotated by a pre-determined factor and fed back through the QG stream cipher to produce another set of $n^2$ values. Godavarty's method, unlike MGK, does not need to have a randomized input vector equivalent in size to the output itself and has lower memory requirements than Markovski and Dimitrova method.

## III. Construction of Low Overhead Quasigroup

Consider a quasigroup of order 6, such as the one in table 1(A). In this quasigroup, each cell may be represented by the expression $(r + c) \ mod \ n$, where $r$ is the row index, $c$ is the column index and $n$ is the order of the group, 6 in this case. Additionally shuffling the rows and columns will produce a randomized quasigroup (See table 1 (B)). We create a randomized quasigroup by shuffling the row and column indices, using $(r + c) mod \ n$ , resulting in only $2n \cdot ceiling(log_2(n))$ bits to store the state of the quasigroup. Further, if an uncalculated quasigroup were stored, the storage requirement would be $n^2 ceiling(log_2(n))$ bits. The memory improvement is thus $O(n^2)$.

If we compare the storage performance to other quasigroup PRNG methods, we see that Godavarty's method [19] has a storage requirement of $2n^2 ceiling(log_2(n))$ and the Markovski et. al. methods [17] [18] require $n^2 ceiling(log_2(n)) + R$, where $n$ is the quasigroup's order and $R$ is the number of desired random bits.

## IV. THE PROPOSED LOW OVERHEAD QUASIGROUP (LOQG) PRNG

Denote the quasigroup matrix as *qgls* and the value denoted by $qgls(s_1,s_2)$ is the value in the matrix at $(s_1,s_2)$.

TABLE 1. QUASIGROUPS OF ORDER 6 WHERE THE QUASIGROUP IN (B) IS THE SHUFFLED FORM OF QUASIGROUP IN (A)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 1 | 2 | 3 | 4 | 5 | 0 |
| 2 | 2 | 3 | 4 | 5 | 0 | 1 |
| 3 | 3 | 4 | 5 | 0 | 1 | 2 |
| 4 | 4 | 5 | 0 | 1 | 2 | 3 |
| 5 | 5 | 0 | 1 | 2 | 3 | 4 |

(A)

| | 1 | 4 | 5 | 0 | 2 | 3 |
|---|---|---|---|---|---|---|
| 0 | 1 | 4 | 5 | 0 | 2 | 3 |
| 3 | 4 | 1 | 2 | 3 | 5 | 0 |
| 5 | 0 | 3 | 4 | 5 | 1 | 2 |
| 2 | 3 | 0 | 1 | 2 | 4 | 5 |
| 4 | 5 | 2 | 3 | 4 | 0 | 1 |
| 1 | 2 | 5 | 0 | 1 | 3 | 4 |

(B)

**Initialize**

1) Begin with a randomized Latin square to form a QG using the low overhead quasigroup to reduce storage
2) Randomly choose two integer seed values ($s_1$, $s_2$) from $[0,n)$.
3) Maintain an index *i*, initialized to zero, used for reshuffling the QG.

**Generate**

1) Let $rand := qgls(s_1,s_2)$
2) Let $s_1 := s_2$ and $s_2 := rand$
3) Shuffle column *i* with column $s_1$, and row *i* with row $s_2$
4) Increment *i mod n*
5) Output *rand* as the random value
6) Repeat

**Initialization with external seed**

1) Initialize as specified in Initialize step 1.
2) Choose a random seed vector of length *l*.[1]

For each 8-bit word in the seed vector *S* of length *l* do the following:

3) Shuffle column *i* with column $S_{j-1}$, and row *i* with row $S_j$
4) Increment *i mod n* and repeat to (3), until $i = l$
5) Finally, let $s_1 := S_{l-1}$ and $s_2 := S_l$

The algorithm begins with the *Initialize* subroutine, a randomized quasigroup matrix and the seed values of $s_1$ and $s_2$. To generate random numbers call the *Generate* subroutine. One can initialize the random number generator with an external seed vector to improve the quality of random numbers (see foot note[1]).

## V. EVALUATION PROCESS

Typically, the abilities of a PRNG are gauged against other well-known PRNG outputs. Test suites such as NIST Statistical Test Suite (NIST-STS) [20] and compression algorithms, like GNU-zip, allows one to establish a greater confidence in the quality of a proposed algorithm [21].

The NIST-STS has pre-established thresholds for pass/fail rating of a random sequence. Often, well known PRNGs fail these tests. Hence, it is important to execute multiple tests and compare success rates of LOQG PRNG with established PRNG algorithms.

The evaluation process of the LOQG PRNG begins by establishing parameters guarding the selection of PRNG output data. We produce 1000 sample outputs, each $2^{22}$ bits (512 KB) for all of the evaluated PRNG algorithms. Test results compare the LOQG PRNG with both commercial PRNGs and previously proposed QG PRNGs. The tested PRNG algorithms are:

- LOQG PRNG order 256
- MGK with glibc-random() driver [18]
- Dimitrova and Markovski [17]
- OpenSSL rand [5]
- arc4random [7]
- Java - SecureRandom [6]
- Microsoft.NET - RandomNumberGenerator [2]

NIST-STS performs a number of tests that compare bit strings of a certain length. If a bit string contains *m* bits, it is referred to as *m*-bit block. Because each of the tests look at varying characteristics of the sample data, the size of *m* differs between tests, and is not configurable by the user. The NIST-STS suite includes the following tests:

- Approximate Entropy (AE) – A test comparing all overlapping *m*-bit patterns.
- Block Frequency (BF) – A test which evaluates the proportion of 1's in *m*-bit blocks.
- Cumulative Sums, Forward (CSF), Reverse (CSR) – Evaluates whether the maximal cumulative sum of partial sequences is outside the range for expected behavior of a random sequence.
- Discrete Fourier Transform (FFT) – Implemented as a Fast Fourier Transform, detects repeating or periodic features that are near to each other.
- Frequency (FREQ) – Evaluates the frequency of 1's and 0's in the entire sequence.
- Longest Run – Comparison of longest contiguous run of 1's in *m*-bit blocks to expected frequency of same.
- Rank – The rank of disjoint sub-matrices within the entire sequence.
- Runs – Finds and evaluates the longest sequence of contiguous 1's in the entire sequence and compares the oscillation between 1's and 0's to a standard frequency.
- Serial – Compares the frequency of all *m*-bit overlapping patterns in the full sequence. Two variations are applied.

[1] The seed vector affects the square at a rate of $4l(n-l)/n^2$ where *l* is the seed vector length in bytes and *n* is the quasigroup order. For example, a seed vector of 16 bytes (128 bits) should sufficiently reconfigure the LOQG PRNG. Consequently, 128 bits are produced from SHA-256 or other hash functions, allowing timestamps to become a good seed.

Figure 1 compares the NIST-STS analysis of the LOQG PRNG order 256 output with analysis of BSD *arc4random*, OpenSSL *rand*, Java *SecureRandom*, and Microsoft.Net *RandomNumberGenerator* outputs. The figure shows the number of successes (pass/fail according to NIST-STS) for 1000 runs of the random number generators. Given any single PRNG suite, the proposed algorithm outperforms it in majority of the tests.

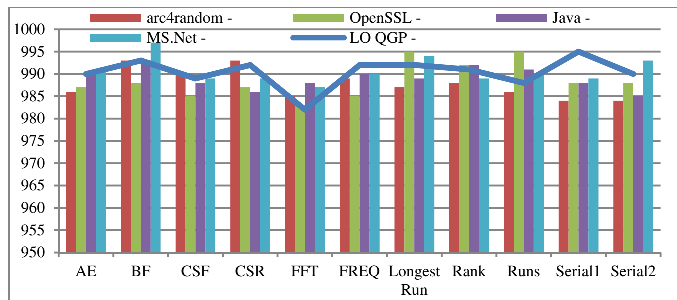


Figure 1. LOQG PRNG vs. Commercial PRNGs: This chart compares the LOQG PRNG with commercial PRNGS. The vertical axis is the number of successful tests for each NIST test area. One thousand (1000) tests were conducted for each PRNG.

Figure 2 compares Godavarty and MGK QG PRNGs with the LOQG PRNG of order 256. The two variations of the LOQG PRNG are shown to demonstrate differences between different QG sizes in the LOQG. Results of the Dimitrova and Markovski [17] were so low, that they obscured results from the other four, and have not been displayed. The Dimitrova and Markovski algorithm failed the Approximate Entropy, Rank, Serial and Longest Run tests; of the others, none scored above the $34^{th}$ percentile. Results show that the LOQG PRNG performs as well or better than Godavarty and MGK.

**Encryption using proposed random number generator:** We take plain text and combine it with random values from the LOQG PRNG algorithm using a quasigroup stream cipher as discussed in section 2. Therefore, the operands on the left hand side of the equation are 8-bit words from the plaintext and 8-bit words from the stream of random numbers. The result value is the cipher text. When decoding, an identical stream of random values must be recombined with the cipher text to reproduce the plain text.

We encrypted PCM audio [22] then evaluated the encrypted cipher text with the NIST-STS suite for randomness. Here we compare P-values produced by the NIST-STS suite. P-value is the probability that an examined stream of numbers is as random as a "perfect" random sequence. TABLE 2 shows that the P-value of the original audio was nearly zero in all cases. However, the P-values for the random numbers and the ciphertext were very close to each other showing that the ciphertext had properties similar to random numbers, essentially "destroying" the information contained within the plaintext.

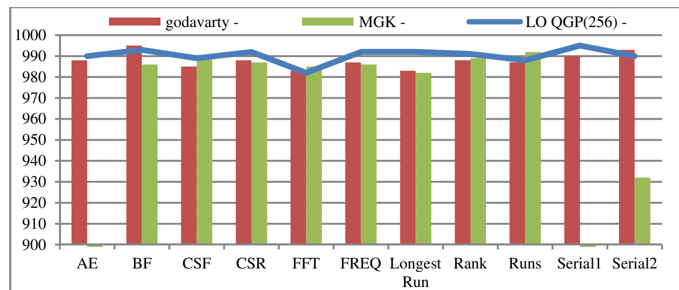


Figure 2. LOQG PRNG vs. Other QG PRNGS: This chart compares the Godavarty [19] and MGK [18] QG PRNGS with LOQG PRNG order 256. The vertical axis is the number of successful tests for each NIST test areas. One thousand (1000) tests were conducted for each PRNG. The Dimitrova-Markovski QG PRNG is not included because its performance maximized at 33.9%, with complete failure on Serial 1 and 2.

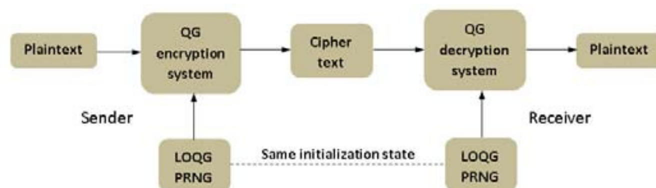


Figure 3. Use of proposed quasigroup random number generator to encrypt a stream of audio data. The operation can be inverted if the receiver knows the initialization state for the random number generator.

**Compression tests:** The GNU-zip utility was chosen to test the compressibility of data produced by the PRNGs. Figure 3 demonstrates results from compressing each of the 1000, $2^{22}$ bit data files with the GNU-zip compression utility. The compression rates are a ratio of the compressed data size versus the original data size, shown are the average, minimum and maximum compression rates across the 1000 samples from each algorithm. With the exception of the Dimitrova-Markovski output, each PRNG produced data, which when compressed became larger instead of smaller (the Dimitrova-Markovski output compressed to approximately $1/1000^{th}$ of the original size). Consequently, fig. 4 is actually measuring the expansion of data when run through GNU-zip.

## VI. CONCLUSIONS

We have presented a novel quasigroups based low overhead pseudo random number generator. The algorithm requires the storage of $2n \cdot ceiling(log_2 n)$ bits, where $n$ is the order of the quasigroup. The algorithm is computationally efficient, as it requires matrix lookup operations and limited number of writes to memory. The quality of random numbers produced by the proposed algorithm is compared against other well-known PRNGs and the results show that the proposed algorithm outperforms any given PRNG in majority of the tests. We also presented the results of using the stream of random numbers generated to encrypt audio data.

TABLE 2. AUDIO ENCRYPTION SUCCESS RATES FOR 1000 RUNS. RAND IS THE STREAM OF RANDOM NUMBERS AND C-TEXT IS THE CIPHER TEXT

| | **Org Audio** | **Average** | **P-Value** | **StdDev-P** | | **Var-P** | | **Number of successes** | |
|---|---|---|---|---|---|---|---|---|---|
| | **P-value** | **Rand** | **C-Text** | **Rand** | **C-Text** | **Rand** | **C-Text** | **Rand** | **C-Text** |
| AF | 0.0000 | 0.4796 | 0.5071 | 0.2837 | 0.2879 | 0.0805 | 0.0829 | 990 | 989 |
| BG | 0.0000 | 0.5177 | 0.4977 | 0.2837 | 0.2843 | 0.0805 | 0.0808 | 993 | 992 |
| CS-F | 0.0000 | 0.5042 | 0.5091 | 0.2852 | 0.2942 | 0.0813 | 0.0866 | 989 | 987 |
| CS-R | 0.0000 | 0.5101 | 0.5076 | 0.2872 | 0.2934 | 0.0825 | 0.0861 | 992 | 988 |
| FFT | 0.0102 | 0.5056 | 0.4962 | 0.2983 | 0.2936 | 0.0890 | 0.0862 | 982 | 990 |
| FREQ | 0.0000 | 0.4965 | 0.4983 | 0.2805 | 0.2885 | 0.0787 | 0.0833 | 992 | 988 |
| Longest Run | 0.0000 | 0.5031 | 0.4908 | 0.2841 | 0.2805 | 0.0807 | 0.0787 | 992 | 994 |
| Rank | 0.0000 | 0.4905 | 0.4949 | 0.2800 | 0.2863 | 0.0784 | 0.0819 | 991 | 982 |
| Runs | 0.0000 | 0.5014 | 0.4951 | 0.2906 | 0.2877 | 0.0845 | 0.0828 | 988 | 987 |
| Serial1 | 0.0000 | 0.5162 | 0.5079 | 0.2861 | 0.2866 | 0.0819 | 0.0822 | 995 | 995 |
| Serial2 | 0.0000 | 0.5153 | 0.5048 | 0.2834 | 0.2821 | 0.0803 | 0.0796 | 990 | 988 |

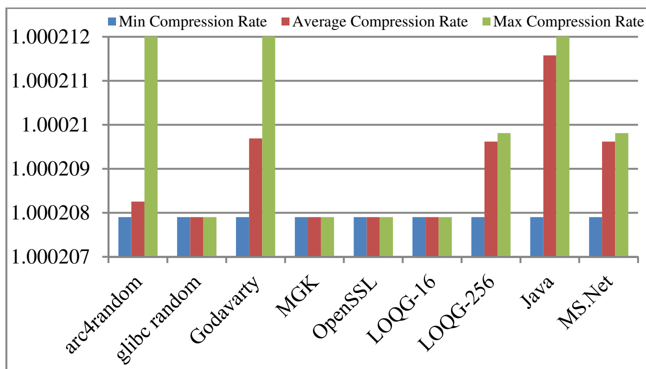


Figure 3. This graph shows minimum, average, and maximum compression rates of the pseudo random data generated by the tested suites. The GNU-libc (glibc) random data, which was used to drive the MGK suite, is also included.

## REFERENCES


[1] TIBOE Software, "TIBOE Programming Community Index for May 2012," 1 May 2012. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. [Accessed 21 May 2012].

[2] Microsoft, "RandomNumberGenerator," [Online]. Available: http://msdn.microsoft.com/en-us/library/system.security.cryptography.randomnumbergenerator.aspx. [Accessed 25 May 2012].

[3] Microsoft, Inc., "CryptGenRandom Function," [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa379942%28v=vs.85%29.aspx. [Accessed 21 May 2012].

[4] Sun Inc./Oracle Inc., "CryptoSpec.thml -- SHA1PRNG," [Online]. Available: http://docs.oracle.com/javase/1.4.2/docs/guide/security/CryptoSpec.html#AppA. [Accessed 21 May 2012].

[5] OpenSSL.org, "rand(3)," [Online]. Available: http://www.openssl.org/docs/crypto/rand.html. [Accessed 21 May 2012].

[6] Oracle/Sun, "SecureRandom," [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html. [Accessed 25 May 2012].

[7] BSD Library Functions Manual, "arc4random(3)," Apple, Inc., [Online]. Available: http://developer.apple.com/library/ios/#documentation/System/Conceptual/ManPages_iPhoneOS/man3/arc4random.3.html. [Accessed 21 May 2012].

[8] NIST, "FIPS Home Page," [Online]. Available: http://www.itl.nist.gov/fipspubs/index.htm. [Accessed 21 June 2012].

[9] NIST, "Secure Hash Standard (FIPS 180-4)," March 2012. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf. [Accessed 21 May 2012].

[10] NIST, "Digital Signature Standard; FIPS 186," [Online]. Available: http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf. [Accessed 21 June 2012].

[11] L. Dorrendorf, Z. Gutterman and B. Pinkas, "Cryptanalysis of the random number generator of the Windows operating system," *ACM Transactions on Information and System Security,* vol. 13, no. 1, pp. 10:1-10:30, 2009.

[12] S. Paul and B. Preneel, "A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher," *Fast Sfotware Encryption 2004 : Lecture notes in Computer Science,* vol. 3017, pp. 245-259, 2004.

[13] NIST, "NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and the Continued Security Provided by SHA-1," NIST, 2004.

[14] M. Battey and A. Parakh, "Efficient quasigroup block cipher for sensors networks," in *To appear in proceedings of 21st International Conference on Computer Communication Networks (ICCCN 2012)*, Munich, Germany, 2012.

[15] M. Satti and S. Kak, "Multilevel indexed quasigroup encryption for data and speech," *IEEE Transactions on Broadcasting,* vol. 55, no. 2, pp. 270-281, 2009.

[16] D. Gligoroski and S. Markovski, *Cryptographic potentials of quasigroup transformations,* Skopje, Republic of Macedonia: University "St. Cyril and Methodious", Institute of Informatics.

[17] J. Dimitrova and V. Markovski, "On quasigroup pseudo random sequence generators," in *1st Balkan Conference in Informatics*, Thessaloniki, pp. 393-401, 2003.

[18] S. Markovski, D. Gligoroski and L. Kocarev, "Unbiased random sequences from quasigroup string transformations," *Fast Software Encryption: 12th International Workshop,* Paris, France, pp. 163-180, 2005.

[19] V. K. Godavarty, "Using quasigroups for generating pseudorandom numbers," CoRR, arXiv:1112.1048v1, 2011.

[20] A. Rukhin and et. al., "SP800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," NIST, 2010.

[21] J. Soto, "Statistical Testing of Random Number Generators," NIST.

[22] "Waveform Audio Format - 11,025 Hz 16 bit PCM audio file," [Online]. Available: http://www.nch.com.au/acm/11k16bitpcm.wav. [Accessed 21 July 2012].