MSTg: Cryptographically strong pseudorandom number generator and its realization

Pavol Svaba
Institute of Business Information Technology
Lucerne University of Applied Sciences and Arts
Zentralstrasse 9
CH-6002 Lucerne, Switzerland

Pascal Marquardt and Tran van Trung
Institut für Experimentelle Mathematik
Universität Duisburg-Essen
Ellernstraße 29
45326 Essen, Germany

Abstract

Random covers for finite groups have been introduced in [5, 9], and [18], and used for constructing public key cryptosystems. The primer [10] introduces a new approach for constructing pseudorandom number generators, called MSTg, based on random covers for large finite groups. In particular, on the class of elementary abelian 2-groups they allow a very efficient realization. The crucial point that makes random covers useful for group based cryptography is the fact, that the problem of finding a factorization with respect to a randomly generated cover is presumed intractable. In other words, random covers induce functions which possess the features of one-way functions. These in addition, as shown by results of statistical tests, behave randomly. We extend the preliminary tests done in [10] and investigate the statistical properties of the output sequences generated by MSTg on proposed class of groups and with different parameters, in particular the block size. We investigate the security of the system and claim that MSTg generators are suitable for cryptographic applications. Finally, we study the possibilities of parallelization of the generator algorithm and propose a method of using MSTg in practice.

Keywords. Finite group, parallelization, pseudorandom number generator, random cover, statistical test of randomness

***AMS Classification*—65C10**

1 Introduction

Random number generators are useful for a variety of purposes. As an alternative to often unpractical and rather inefficient, mostly of physical nature based schemes called *True random number generators* (TRNG), a special class of deterministic algorithms has been created to satisfy constantly growing demand after random numbers. These, so called *Pseudorandom*

number generators (PRNG), make use of mathematical formulae to produce sequences of numbers that appear random. As the word "pseudo" suggests, the output sequences of PRNG are not random in the way you might expect. On the other hand, they are usually very efficient, meaning they can produce many numbers in a short time, and deterministic, and therefore the output sequences can be reproduced at any time. For that purpose, a small initial information which is unique for a given sequence, a so called *seed*, is required. These characteristics make PRNGs suitable for applications where large amounts of random data are required, and where it is useful that the same sequence can be replayed if necessary. Popular examples of such applications are simulation and modeling applications. In other applications like gambling or in the field of cryptography, for example for generating key materials of encryption schemes, is generating random bit sequences an important problem and many schemes depend on the unpredictability of this process. We suggest that the reader consults [4, 13] for information about pseudorandom number generators.

The concept of random covers for finite groups was introduced recently as a basic tool for a new approach to designing public key cryptosystems based on finite groups [5, 9]. The most striking property of random covers regarding cryptographic applications is that they induce a large class of functions that behave randomly. In [6, 8], a special type of covers, called logarithmic signatures, for permutation groups are used to construct pseudorandom number generators and to generate random permutations in the symmetric group.

In this paper we introduce a new approach to designing PRNGs based on *random covers* for finite groups. On a certain class of groups, this new type of generators, called $MSTg$, turns out to be highly efficient and produces high-quality random bit sequences. More importantly, we provide evidence that this class of generators is suitable for cryptographic applications. We study the possibilities to parallelize the generator algorithm with the goal of a performance boost.

The paper is organized as follows: In Section 2, we summarize some basic facts about random covers and their induced mappings. In Section 3, we present a new approach to designing pseudorandom number generators based on random covers. The realization of the generator on the class of elementary abelian 2-groups is introduced in Section 4, in particular, its default version with two random covers. In Section 5, we present results of a thorough statistical test of these generators. The test was carried out using the NIST Statistical Test Suite and the Diehard battery of statistical tests developed by George Marsaglia. In Section 6, we discuss the security of the system and the right choice of parameters for usage in cryptographic applications. Section 7 points out some benefits and interesting practical hints of the usage of $MSTg$ generators. Section 8 investigates the possibilities of parallelization of the generator algorithm and presents the data of performance. The paper closes with a conclusion.

2 Random covers for finite groups and their induced mappings

In this section we briefly present definitions and some basic facts about covers for finite groups and their induced mappings. For more details the reader is referred to [5, 9, 18]. The notation used about groups is standard and may be found in any textbook of group theory.

Let $\mathcal{G}$ be a finite abstract group, we define the *width* of $\mathcal{G}$ as the positive integer $w =$

$\lceil \log |\mathcal{G}| \rceil$. Let $\mathcal{S}$ be a subset of $\mathcal{G}$. Suppose that $\alpha = [A_1, A_2, \ldots, A_s]$ is an ordered collection of subsets $A_i \subseteq \mathcal{G}$, such that $\sum_{i=1}^{s} |A_i|$ is bounded by a polynomial in the width of $\mathcal{G}$. We call α a *cover* for $\mathcal{S}$ if every product $a_1 \cdot a_2 \cdots a_s$ lies in $\mathcal{S}$ and each element $h \in \mathcal{S}$ can be written as a product of the form

$$h = a_1 \cdot a_2 \cdots a_{s-1} \cdot a_s \tag{1}$$

with $a_i \in A_i$. In particular, if each $h \in \mathcal{S}$ can be expressed in exactly one way by Equation (1), α is called a *logarithmic signature* for $\mathcal{S}$. If the elements in each set A_i, $i = 1, \ldots, s$, are chosen at random from the elements in $\mathcal{G}$, we refer to α as a *random cover*. If $\mathcal{S} = \mathcal{G}$, α is called a cover for $\mathcal{G}$. Normally, the context precludes ambiguity and sometimes a cover for a subset of $\mathcal{G}$ will be called a cover for $\mathcal{G}$. The A_i are called the *blocks*, and the vector $(r_1, \ldots, r_s)$ with $r_i = |A_i|$ the *type* of α. We say that α is *nontrivial* if $s \geq 2$ and $r_i \geq 2$ for $1 \leq i \leq s$; otherwise α is said to be *trivial*. A cover α is called *factorizable* if the factorization in Equation (1) can be achieved in time polynomial in the width w of $\mathcal{G}$, for all but a negligible number of elements of $\mathcal{G}$, otherwise it is called *non-factorizable*.

In general, the problem of finding a factorization as in Equation (1) with respect to a randomly generated cover is presumed intractable. There is strong evidence of the hardness of the problem. For example, let $\mathcal{G}$ be a cyclic group and g be a generator of $\mathcal{G}$. Let $\alpha = [A_1, A_2, \ldots, A_s]$ be any cover for $\mathcal{G}$, for which the elements of A_i are written as powers of g. Then the factorization with respect to α leads to solving the Discrete Logarithm Problem (DLP) in $\mathcal{G}$.

Remark 2.1 *It is worth noting that the problem of how to generate random covers for finite groups of large order is treated in [17]. Probabilistic methods show that the generation of random covers for groups of large order can be done with high efficiency and at minimum cost.*

Let $\alpha = [A_1, A_2, \ldots, A_s]$ be a random cover of type $(r_1, r_2, \ldots, r_s)$ for $\mathcal{G}$ with $A_i = [a_{i,1}, a_{i,2}, \ldots, a_{i,r_i}]$ and let $m = \prod_{i=1}^{s} r_i$. Let $m_1 = 1$ and $m_i = \prod_{k=1}^{i-1} r_k$ for $i = 2, \ldots, s$. Let τ denote the canonical bijection from $\mathbb{Z}_{r_1} \otimes \mathbb{Z}_{r_2} \otimes \cdots \otimes \mathbb{Z}_{r_s}$ on $\mathbb{Z}_m$; i.e.

$$\begin{aligned} \tau &: \mathbb{Z}_{r_1} \otimes \mathbb{Z}_{r_2} \otimes \cdots \otimes \mathbb{Z}_{r_s} \longrightarrow \mathbb{Z}_m \\ \tau(j_1, j_2, \ldots, j_s) &:= \sum_{i=1}^{s} j_i m_i. \end{aligned}$$

Using τ we now define the surjective mapping $\breve{\alpha}$ induced by α.

$$\begin{aligned} \breve{\alpha} &: \mathbb{Z}_m \longrightarrow \mathcal{G} \\ \breve{\alpha}(x) &:= a_{1,j_1} \cdot a_{2,j_2} \cdots a_{s,j_s}, \end{aligned}$$

where $(j_1, j_2, \ldots, j_s) = \tau^{-1}(x)$. Since τ and τ^{-1} are efficiently computable, the mapping $\breve{\alpha}(x)$ is efficiently computable.

Let $\gamma : 1_{\mathcal{G}} = \mathcal{G}_0 < \mathcal{G}_1 < \cdots < \mathcal{G}_s = \mathcal{G}$ be a chain of subgroups of $\mathcal{G}$, and let B_i be an ordered, complete set of right (or left) coset representatives of $\mathcal{G}_{i-1}$ in $\mathcal{G}_i$. It is clear that $[B_1, \ldots, B_s]$ forms a logarithmic signature for $\mathcal{G}$, called a *transversal logarithmic signature*.

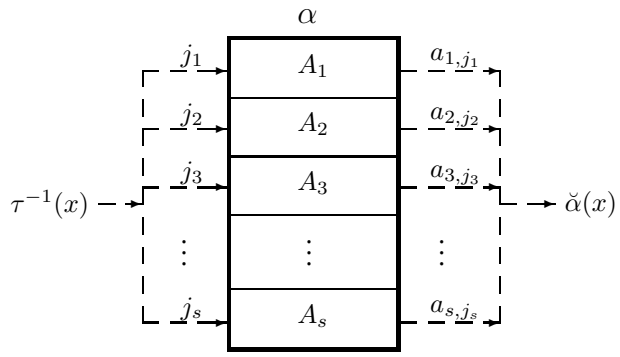

Figure 1: Mapping induced by cover α.

Transversal logarithmic signatures are an important example of factorizable logarithmic signatures [9]. In case that β is a factorizable logarithmic signature for a group $\mathcal{G}$ the induced function

$$\breve{\beta}: \mathbb{Z}_m \longrightarrow \mathcal{G}$$

is a bijection, which is efficiently invertible. Conversely, given a cover α and an element $y \in \mathcal{G}$, to determine any element $x \in \breve{\alpha}^{-1}(y)$ it is necessary to obtain any one of the possible factorizations of type (1) for y and determine indices $j_1, j_2, \ldots, j_s$ such that $y = a_{1,j_1} \cdot a_{2,j_2} \cdots a_{s,j_s}$. This is possible if and only if α is factorizable. Once a vector $(j_1, j_2, \ldots, j_s)$ has been determined, $\breve{\alpha}^{-1}(y) = \tau(j_1, j_2, \ldots, j_s)$ can be computed efficiently. We make use of the following cryptographic hypothesis that if $\alpha = [A_1, A_2, \ldots, A_s]$ is a random cover for a "large" subset $\mathcal{S}$ of a group $\mathcal{G}$, then finding a factorization in (1) is an intractable problem. In other words, the mapping

$$\breve{\alpha}: \mathbb{Z}_m \longrightarrow \mathcal{S}$$

induced by α with $m = \prod_{i=1}^{s} |A_i|$ is a one-way function.

3 PRNG based on random covers

In this section we present the basic principle of building a generic PRNG based on random covers for finite groups. There are no restrictions on the group structure for the PRNG in this generic case. We call our random cover based pseudorandom number generator $MSTg$. From now on let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two chosen finite groups with $|\mathcal{G}_1| = n$, $|\mathcal{G}_2| = m$ and $n \geq m$. In case that $\mathcal{G}_1$ contains a subgroup of order m, then we will choose $\mathcal{G}_2$ as such a subgroup. In particular, if $n = m$ we have $\mathcal{G}_1 = \mathcal{G}_2$.

Let ℓ be an integer such that $\ell \geq n$. Let $k \geq 0$ be a fixed integer. Let α be a random cover of type $(u_1, u_2, \ldots, u_t)$ for $\mathcal{G}_1$ with $\prod_{i=1}^{t} u_i = \ell$. Let $\alpha_1, \ldots, \alpha_k$ be a set of random covers of type $(r_1, r_2, \ldots, r_s)$ for $\mathcal{G}_1$ with $\prod_{i=1}^{s} r_i = |\mathcal{G}_1|$. Let $\gamma = [H_1, H_2, \ldots, H_s]$ be a random cover of type $(r_1, r_2, \ldots, r_s)$ for $\mathcal{G}_2$. We assume that there are bijective mappings $f_1: \mathcal{G}_1 \to \mathbb{Z}_n$ and $f_2: \mathcal{G}_2 \to \mathbb{Z}_m$, which efficiently identify elements of $\mathcal{G}_1$ with numbers in $\mathbb{Z}_n$ and elements of $\mathcal{G}_2$ with numbers in $\mathbb{Z}_m$. We define a function

$$F: \mathbb{Z}_\ell \longrightarrow \mathbb{Z}_m$$

as a composition of mappings as follows.

$$\mathbb{Z}_\ell \xrightarrow{\breve{\alpha}} \mathcal{G}_1 \xrightarrow{f_1} \mathbb{Z}_n \xrightarrow{\breve{\alpha}_1} \mathcal{G}_1 \xrightarrow{f_1} \mathbb{Z}_n \longrightarrow \cdots \xrightarrow{\breve{\alpha}_k} \mathcal{G}_1 \xrightarrow{f_1} \mathbb{Z}_n \xrightarrow{\breve{\gamma}} \mathcal{G}_2 \xrightarrow{f_2} \mathbb{Z}_m, \qquad (A)$$

Another alternative to define the function $F : \mathbb{Z}_\ell \longrightarrow \mathbb{Z}_m$ is as follows.

$$\mathbb{Z}_\ell \xrightarrow{\breve{\alpha}} \mathcal{G}_1 \xrightarrow{f_1} \mathbb{Z}_n \xrightarrow{\breve{\gamma}} \mathcal{G}_2 \xrightarrow{f_2} \mathbb{Z}_m \xrightarrow{\breve{\delta}_1} \mathcal{G}_2 \xrightarrow{f_2} \mathbb{Z}_m \longrightarrow \cdots \xrightarrow{\breve{\delta}_k} \mathcal{G}_2 \xrightarrow{f_2} \mathbb{Z}_m, \qquad (B)$$

where $\delta_1, \ldots, \delta_k$ are a set of random covers of type $(v_1, v_2, \ldots, v_w)$ for $\mathcal{G}_2$ with $\prod_{i=1}^{w} v_i = |\mathcal{G}_2|$. It turns out that the function F has a strong random behavior, when the involved covers are randomly generated. Using F we define the $MSTg$ in the following algorithm. The $MSTg$

Algorithm 1 MSTg: Pseudorandom number generator based on covers for finite groups

Input: Integers ℓ, m, function $F : \mathbb{Z}_\ell \longrightarrow \mathbb{Z}_m$ as defined above, a random and secret seed $s_0 \in \mathbb{Z}_\ell$, a constant $C \in \mathbb{Z}_\ell$.

Output: t pseudorandom numbers $z_1, \ldots, z_t \in \mathbb{Z}_m$

1: **for** $i \leftarrow 1$ t **do**
2: $\quad s_i = (s_{i-1} + C) \bmod \ell$.
3: $\quad z_i = F(s_i)$.
4: **end for**
5: **return** $(z_1, \ldots, z_t)$

as presented in Algorithm 1 uses a simple counter mode to generate output sequences of pseudorandom numbers via the function F. However, we see that any other suitable mode can be used in place of the counter mode in the algorithm. Since the function F is the core of $MSTg$, which is designed for cryptographic applications, great care must be taken in the generation of F. In other words we need a good (not necessarily cryptographically secure) TRNG or PRNG, for example the Blum-Blum-Shub, or Mersenne Twister, or even a fixed $MSTg$ (as shown in later sections) to create the random covers involved in F. Assume that we have chosen such a generator. Using a randomly selected seed, the generator will generate random group elements for the covers involved in defining F. For example, to generate k random covers α_i of type $(r_1, \ldots, r_s)$ for $\mathcal{G}_1$ the generator will generate an output of $k \sum_{i=1}^{s} r_i$ numbers in $\mathbb{Z}_n$. By using $f_1^{-1} : \mathbb{Z}_n \longrightarrow \mathcal{G}_1$ we obtain from these numbers $k \sum_{i=1}^{s} r_i$ group elements in $\mathcal{G}_1$, which are then used to form the k random covers α_i.

It is worth mentioning that the induced mapping $\breve{\beta} : \mathbb{Z}_n \longrightarrow \mathcal{G}_1$, as described in Section 2, where β is a transversal logarithmic signature of $\mathcal{G}_1$, could be used for f_1^{-1}, see for instance [5, 9]. For a certain class of abelian groups the mapping $f_1 : \mathcal{G}_1 \longrightarrow \mathbb{Z}_n$ becomes the trivial identity mapping, as we will see in the next section, and hence each number in $\mathbb{Z}_n$ may be used as a group element as well.

Suppose the groups $\mathcal{G}_1$ and $\mathcal{G}_2$ of order n and m have been chosen. How large does the value k need to be? The question is inherently related to the performance and quality of $MSTg$. For if k would need to be large, then a large amount of storage would be required to set up the function F and consequently the speed of computation with F would necessarily be reduced. Surprisingly, our experimental investigation based on statistical tests shows that when $\mathcal{G}_1$ and $\mathcal{G}_2$ are elementary abelian 2-groups (i.e. $\mathcal{G}_1$ and $\mathcal{G}_2$ may be viewed as vector spaces over $\mathbb{F}_2$), then $k = 0$ or 1 are sufficient. It means only two or three covers are required

for constructing function F. The implication is that $MSTg$ is, in fact, a good pseudorandom number generator. Moreover, for these groups we may even ignore the mappings f_1 and f_2, therefore the performance of the generator is highly efficient, as presented in the subsequent section.

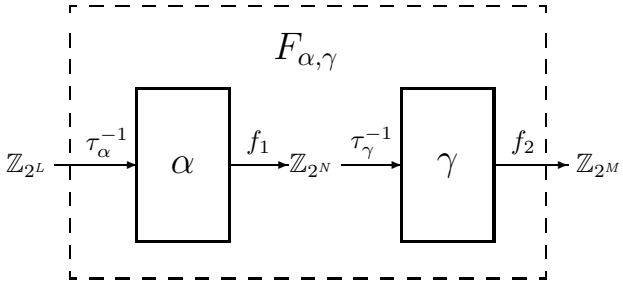

Figure 2: Two cover MSTg.

4 Realization on elementary abelian 2-groups

In this section we present the realization of $MSTg$, for which the underlying groups are elementary abelian 2-groups. These groups may be viewed as the additive groups of the finite fields of characteristic 2 and therefore they are suitable for computer implementation.

From now on we assume that $\mathcal{G}_1$ is an elementary abelian 2-group with $|\mathcal{G}_1| = n = 2^N$ and $\mathcal{G}_2$ is a subgroup of $\mathcal{G}_1$ with $|\mathcal{G}_2| = m = 2^M$. The group operation will be written additively as XOR in this case, so that the groups may be realized as vector spaces over $\mathbb{F}_2$ and their elements as binary vectors of length N and M respectively. We study $MSTg$ based on this class of groups.

4.1 Two covers $MSTg$

Let α be a random cover for $\mathcal{G}_1$ of type $(u_1, \ldots, u_t)$ with $\prod_{i=1}^{t} u_i = \ell = 2^L$. Also let γ be a random cover for $\mathcal{G}_2$ of type $(r_1, \ldots, r_s)$ with $\prod_{j=1}^{s} r_j = 2^N$. Typically we choose $L \geq N > M$ (i.e. Type (A), see previous section). We construct two covers $MSTg$ where function $F_{\alpha,\gamma}$ (see Figure 2) is defined using

$$\begin{aligned} \tau_\alpha^{-1} &: \mathbb{Z}_{2^L} \longrightarrow \mathbb{Z}_{u_1} \otimes \mathbb{Z}_{u_2} \otimes \cdots \otimes \mathbb{Z}_{u_t} \\ f_1 &: \mathcal{G}_1 \longrightarrow \mathbb{Z}_{2^N} \\ \tau_\gamma^{-1} &: \mathbb{Z}_{2^N} \longrightarrow \mathbb{Z}_{r_1} \otimes \cdots \otimes \mathbb{Z}_{r_s} \\ f_2 &: \mathcal{G}_2 \longrightarrow \mathbb{Z}_{2^M} \end{aligned}$$

As the block lengths $u_1, \ldots, u_t, r_1, \ldots, r_s$ are all powers of 2, the computation of the required block indices received as image of τ_α^{-1} (τ_γ^{-1} resp.) can be realized as splitting of the preimage, represented in its binary form, into parts of the appropriate length $\log_2 u_i$, ($\log_2 r_j$, resp.). For example, taking $\alpha = [a_{ij}]$ of type $(2^2, 2^3, 2^2, \ldots, 2)$ and an input of the form $x = 1110100\ldots1$, the received indices (in binary form) are $\tau_\alpha^{-1}(x) = (11, 101, 00, \ldots, 1)$, and therefore $\breve{\alpha}(x) = a_{1,3} \cdot a_{2,5} \cdot a_{3,0} \cdots a_{t,1}$. Moreover, as the cover entries (elements of $\mathcal{G}_1$ and $\mathcal{G}_2$) are represented as

binary vectors, the mappings f_1, f_2 are trivial and can be ignored, so the only mathematical operation required is bitwise XOR of vectors on positions given by these indices.

Seeding the generator with s_0, we can efficiently generate a pseudorandom sequence of M-bit numbers z_1, z_2, z_3, etc., where $z_i = F_{\alpha,\gamma}((s_0 + i \cdot C) \bmod 2^L)$.

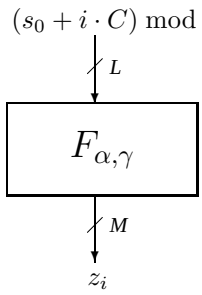

Figure 3: Counter mode MSTg.

5 Statistical testing of $MSTg$

In this section we investigate the randomness of output bit sequences from $MSTg$ with two covers (as well as with one cover in a special case). Two significant randomness test suites are used for this purpose. The first one is the NIST Statistical Test Suite in the Special Publication 800-22 Revision 1a (Revised: April 2010) issued by the National Institute of Standards and Technology. The second is the DIEHARD Statistical Tests developed by George Marsaglia. The aim of these tests is to investigate the quality of the approach of using covers for building $MSTg$ based on elementary abelian 2-groups. The test results presented in this paper extend the initial tests published in [10]. We particularly aim at testing $MSTg$ with various parameters such as the order of underlying groups $\mathcal{G}_1, \mathcal{G}_2$ (i.e. input / output size) and cover block size.

5.1 NIST Statistical Tests Suite

To begin with we make a brief description of the strategy for statistical analysis of a random number generator by the NIST test, see [16]. The NIST Statistical Test Suite consists of 15 core tests that, by reason of different parameter inputs, can be considered as 188 statistical tests.

(1) For each statistical test and each binary sequence (of length N), a P-value is computed from the test statistic of this specified statistical test. A success/failure is assigned to this P-value on the basis of the significance level.

(2) For each statistical test and each sample two further evaluations are made. The first one is the proportion of binary sequences passing the statistical test. The range of acceptable proportions is determined by the significance level and the sample size M. The second evaluation is an additional P-value computed on the basis of χ^2 test applied to the P-values in the entire sample. This additional P-value is examined to ensure the uniformity of the test sequences and is computed, based on the distribution of P-values

obtained for the statistical test on the 10 equally divided sub-intervals between 0 and 1.

(3) A sample is considered to have passed a statistical test if the proportion in step (2) is in the interval of acceptance and the additional P-value in step (2) exceeds a certain value. If either of the two evaluations i n step (2) is not fulfilled, then the sample is labeled as suspect. If this occurred, additional samples would need to be tested. Otherwise, we say that the sample passed the test.

We fix the input parameters for each statistical test such as sequence length $N = 10^7$ (of the test binary sequence), sample size $M = 100$ (the number of binary sequences), and significance level equal to 0.01. The strategy of our evaluation consists in generating a random[1] $MSTg$ that, in turn by using a random seed, generates one sample of bit sequences of $N \cdot M$ bits. The sample is then tested against all 15 tests in the suite simultaneously. This process is repeated 1056 times, each time using freshly generated $MSTg$. In other words, each tested sample comes from different generator. The percentage of samples (generators $MSTg$) that passes all the tests including other data obtained from the tests will be then used to compare with other known pseudorandom number generators under the same defined condition. For this purpose, we have chosen widely used generators Blum-Blum-Shub [2], ARC4 [15], and Mersenne Twister [12]. For example, we randomly generate 1056 Blum-Blum-Shub PRNGs, each of them will generate a sample of $M \cdot N$ bits. This sample is tested under the same conditions as those for $MSTg$. This way provides an adequate comparison of the test results of $MSTg$ with other generators. Table 1 presents the results of the NIST statistical test of different versions of $MSTg$ including the generators Blum-Blum-Shub (BBS), ARC4 and Mersenne Twister (MT). Notice that Mersenne Twister is not suitable for cryptographic applications. Some explanations need to be included for the reading of the table. As described above all 15 NIST tests on a sample of 100 bit sequences of size 10^7 each reports a set of 188 P-values, for short we call it P-value set. Thus there are 1056 P-value sets. The column f_0 gives the total number of P-value sets that pass the NIST tests. The column f_i, $i = 1, 2, 3, 4$, records the entire number of P-value sets, which have exactly i "defect" P-values; column f_{5+} shows number of P-value sets of 5 or more "defect" P-values. The column f_{max} gives the maximum number and f_{aver} the average number of defect P-values (for a P-value set) for each NIST test. Column #c gives the number of covers used in the corresponding version of $MSTg$. Column C gives the prime constant used in the counter mode.

5.2 DIEHARD Battery of Tests

The Diehard Battery of Tests of Randomness is provided by George Marsaglia [11]. The Diehard Test Suite is composed of 18 tests and as the number of P-value varies over tests, it provides 219 P-values entirely. For each test we generate a random[2] $MSTg$ and use it to produce binary sequence of fixed length 10 MBytes which is tested against all 18 test simultaneously. We rerun this process 50160 times, using a newly generated $MSTg$ for each

[1] It is well-known that the Blum-Blum-Shub generator produces high quality bit sequences, so we use it to initialize covers for each tested $MSTg$. Parameters of the generator are the same as of the tested version, see Table 1 (BBS). Blum-Blum-Shub, however, can be replaced by any good (not necessarily cryptographically secure) PRNG or TRNG, as pointed out in later sections.

[2] The same Blum-Blum-Shub generator as for NIST test is used to initialize $MSTg$ for Diehard test.

repeat. Contrary to the NIST test suite, the Diehard test suite does not suggest a method of how to interpret the test results. These are often evaluated (and interpreted) differently. We make a stringent condition to interpret a generator as having passed the Diehard test by requiring that all 219 P-values for a given (fixed) sequence have to belong to some chosen interval I_1, see Table 2. This is in fact a strong requirement, because for a truly random sequence, not all of its P-values would necessary belong to such a fixed interval. Under such a criterion the proportion of sequences passing the Diehard test will strongly be reduced. We compare the proportion of passed sequences using different conditions (intervals) produced by $MSTg$ with the outputs from Blum-Blum-Shub (BBS), ARC4 and Mersenne Twister (MT). The results are summarized in the Table 2. Column FB records the total number of the so called "FAILS BIG" P-values, i.e. P-values that are not in I_6.

6 Security of $MSTg$

The test results in the previous section show in particular that all tested $MSTg$ generate good output sequences that pass all the NIST and Diehard tests. For cryptographic applications we suggest that two or three random covers with block size at least 16 should be used for each $MSTg$. We prefer to use prime numbers for the constant C in the counter mode if two covers are used, even though it is not necessary. For one cover version, the usage of primes is virtually essential, see also notes in Table 1.

For the completeness, we recall the argumentation about the security of $MSTg$ from [10]. Let $\mathcal{G}_1$ be an elementary abelian 2-group of order $|\mathcal{G}_1| = n = 2^N$ having XOR as the group operation. Let $\mathcal{G}_2$ be a subgroup of $\mathcal{G}_1$ with $|\mathcal{G}_2| = m = 2^M$. Suppose we construct a random $MSTg$ of Type (A) as shown in Section 3. For simplicity we choose, for example, $\ell = n = |\mathcal{G}_1|$, $k = 0$ and we generate two random covers α and γ for $\mathcal{G}_1$ and $\mathcal{G}_2$. Then $F = \breve{\gamma} \circ \breve{\alpha} : \mathbb{Z}_{2^N} \longrightarrow \mathbb{Z}_{2^M}$, the composition of $\breve{\alpha} : \mathbb{Z}_{2^N} \longrightarrow \mathbb{Z}_{2^N}$ and $\breve{\gamma} : \mathbb{Z}_{2^N} \longrightarrow \mathbb{Z}_{2^M}$ is the function presenting the generator. Recall that the bijections f_1 and f_2 for the identification of $\mathcal{G}_1$ with $\mathbb{Z}_{2^N}$ and $\mathcal{G}_2$ with $\mathbb{Z}_{2^M}$ become the identity functions and therefore can be ignored.

One of the main questions regarding the security of the generator is whether we are able to determine the seed from a given output sequence. Actually, for $MSTg$, this is equivalent to the problem of invertibility of the function F. In the following we present argumentation showing that for $MSTg$ based on sufficiently large groups this problem is computationally infeasible. If we would make use of the cryptographic hypothesis from Section 2 stating, that random covers induce one-way functions, we would readily have the answer. Instead, we provide a proof without using the hypothesis when covers are used in an appropriate manner.

To begin with, we first make a simple but important remark that we can ensure that α and γ together cannot be replaced by a single cover β for $\mathcal{G}_2$, having the same type as α. This can be checked efficiently by computing at most $\sum_{i=1}^{t} u_i$ appropriate chosen inputs, where $(u_1, \ldots, u_t)$ is the type of α (for more details see below). The probability that the randomly generated covers α and γ can be replaced by a single cover β is actually negligible.

Now the mapping $\breve{\gamma} : \mathbb{Z}_{2^N} \longrightarrow \mathbb{Z}_{2^M}$ could be considered as a compression function of N bit to M bit. Since this function is constructed using random group elements of $\mathcal{G}_2$, it is expected that for each output $z \in \mathbb{Z}_{2^M}$ there are on average 2^{N-M} elements $y \in \mathbb{Z}_{2^N}$ such that $\breve{\gamma}(y) = z$, see [17]. Let $Y \subset \mathbb{Z}_{2^N}$ denote the set of 2^{N-M} preimages y of z. It should be noted that the set Y is not determined and experimental results also show that Y behaves like

PRNG	input size ($\log_2 \ell$) [bit]	output size ($\log_2 m$) [bit]	block size	#c	C^*	f_{aver}	f_0/R_N	f_{max}	f_0	f_1	f_2	f_3	f_4	f_{5+}
$MSTg$	512	256	256	2	p_{512}	0,757	0,473	5	499	383	122	38	12	2
			16			0,772	0,473	5	500	364	136	48	5	3
			4			0,764	0,486	7	513	346	148	36	10	3
	256	192	256	2	p_{256}	0,746	0,479	6	506	373	135	28	11	3
			16			0,774	0,473	5	499	367	133	47	7	3
			4			0,747	0,473	6	499	377	140	33	4	3
	256	128	256	2	p_{256}	0,759	0,470	6	496	382	131	34	10	3
			16			0,748	0,485	8	512	359	140	35	8	2
			4			0,769	0,457	5	483	399	122	41	9	2
	192	128	256	2	p_{192}	0,696	0,512	6	541	349	126	28	11	1
			16			0,766	0,477	5	504	356	149	34	12	1
			4			0,736	0,500	6	528	341	140	38	5	4
	128	64	256	2	p_{128}	0,736	0,501	7	529	351	120	43	10	3
			16			0,724	0,479	6	506	385	125	32	7	1
			4			0,768	0,457	5	483	386	144	36	6	1
	64	64	256	2	p_{64}	0,736	0,489	7	516	361	133	38	6	2
			16			0,723	0,494	6	522	362	129	31	10	2
			4			0,746	0,491	5	519	355	126	46	7	3
	64	64	256	1	p_{64}	0,721	0,485	5	512	385	110	42	5	2
			16			0,751	0,482	5	509	357	143	39	7	1
			4			0,797	0,475	8	502	357	135	44	10	8
BBS	1024	1	-	-	-	0,759	0,474	5	501	370	134	41	9	1
ARC4	256	8	-	-	-	0,719	0,500	6	528	361	120	38	2	7
MT	64	64	-	-	-	0,712	0,491	6	518	371	130	30	5	2

All statistical tests have been carried out on the CRAY XT6m of the University of Duisburg-Essen. For each PRNG we carried out 1056 tests using the NIST Statistical Test Suite. Each test analyzed an output sequence of 10^9 bits generated by using a unique random seed. Moreover, each $MSTg$ output is created by a different generator, filled with randomly generated entries. On a single processor one NIST test of a 10^9 bit sequence took approximately 5 hours.

$R_N = 1056$,
$p_{512} = 8099619925894334754391218150220793215016775072154324214616073694588666106395-8845662083170150318243310546588563971899027711952142568870471145525912271517 37$,
$p_{256} = 4093868575373206380882448599758645819911219040395778584980736465259745305 2901$,
$p_{192} = 4915468052689477438875902348727949892504679620891044976651$,
$p_{128} = 237285956799898977113083501268719939217$,
$p_{64} = 11895088228400396813$.

* We have experimentally evaluated the primes C for most of the tested versions of $MSTg$. Used values keep $|L - \log_2 C| \approx 1 \pm 0.7$, where L is the input size in bits ($L = \log_2 \ell$). The "optimal" value for C modifies each of the pointers $(j_1, \ldots, j_s)$ after every cycle and simultaneously keeps the use of all cover entries approximately uniform. As the input is computed mod 2^L, the prime is used to prevent the generator to run into a cycle with the period smaller than 2^L. To improve the performance, the constant $C = 1$ could be used instead of a prime.

Table 1: The test results of NIST.

a random set taking from the universe of size 2^N. Define $X := \breve{\alpha}^{-1}(Y)$. Again, on average, we may expect that $|X| = 2^{N-M-\delta}$ with $0 \leq \delta \leq 2$, [18]. This set X must contain the seed s_0, for which $F(s_0) = z$. An exhaustive search needs to be done to determine s_0, and this

PRNG	input size ($\log_2 \ell$) [bit]	output size ($\log_2 m$) [bit]	block size	#c	C	I_1	I_2	I_3	I_4	I_5	I_6	FB
$MSTg$	512	256	256	2	p_{512}	3040	25504	34116	46504	48897	49439	724
			16			3159	25413	34066	46447	48880	49436	728
			4			3089	25271	33994	46329	48878	49452	712
	256	192	256	2	p_{256}	3078	25348	34099	46452	48873	49454	710
			16			3056	25294	34100	46345	48828	49390	773
			4			3109	25349	34063	46436	48878	49462	708
	256	128	256	2	p_{256}	3049	25327	34058	46374	48868	49459	704
			16			3065	25361	34045	46374	48826	49441	721
			4			3048	25212	34024	46493	48885	49472	697
	192	128	256	2	p_{192}	2993	25204	34026	46501	48886	49435	732
			16			3119	25350	34062	46418	48913	49435	732
			4			3084	25279	34031	46395	48875	49425	738
	128	64	256	2	p_{128}	3000	25253	33900	46482	48901	49437	732
			16			2984	25219	34003	46534	48916	49447	715
			4			3011	25241	34117	46471	48836	49436	728
	64	64	256	2	p_{64}	3061	25199	33996	46376	48855	49423	739
			16			3005	25199	34020	46487	48877	49455	711
			4			3061	25300	33970	46437	48900	49471	692
	64	64	256	1	p_{64}	2968	25230	34157	46442	48819	49406	759
			16			3137	25341	33996	46417	48871	49442	724
			4			3108	25392	33999	46486	48893	49475	693
BBS	1024	1	-	-	-	3053	25231	34070	46431	48886	49442	720
ARC4	256	8	-	-	-	3054	25368	33971	46361	48920	49449	712
MT	64	64	-	-	-	3044	25236	34000	46446	48873	49427	736

All statistical tests have been carried out on the CRAY XT6m of the University of Duisburg-Essen. For each PRNG we carried out 50160 tests using the Diehard Battery of Tests. Each test analyzed an output bit sequence of size 10 MBytes, generated by using a unique random seed. Each $MSTg$ output is generated from a different generator.

$R_D = 50160$,
$I_1 = [0.005, 0.995]$,
$I_2 = [0.001, 0.999]$,
$I_3 = [0.0005, 0.9995]$,
$I_4 = [0.00005, 0.99995]$,
$I_5 = [0.000005, 0.999995]$,
$I_6 = [0.000001, 0.999999]$,
$p_{512}, \ldots, p_{64}$, see Table 1.

Table 2: The test results of Diehard.

search has a complexity of size $\mathcal{O}(2^{N-M-\delta-1})$. Thus if $N-M$ is sufficiently large, say larger than 100, it is computationally impossible to determine s_0. The one-way-ness of F provides sufficient evidence about the security of $MSTg$ against the determination of the seed (when long output sequences are produced). This is similar to the case where strong encryption systems, for example AES, RSA, are used to build pseudorandom number generators.

An interesting property of the mappings $\breve{\alpha}$ and $\breve{\gamma}$ is that their outputs should have a strong independency. For example, consider the mapping $\breve{\alpha}$. Let x and x' be two different

given input for $\breve{\alpha}$. Let $\alpha = [A_1, \ldots, A_t]$ with $A_i = \{a_{i\,1}, \ldots, a_{i\,j_{r_i}}\}$, $i = 1, \ldots, t$. Assume $\breve{\alpha}(x) = \sum_{i=1}^{t} a_{i\,j_i}$ and $\breve{\alpha}(x') = \sum_{k=1}^{t} a_{i\,k_i}$. Then there exists at least a value i such that $j_i \neq k_i$. On the other hand since the elements for the cover are randomly generated from a large group $\mathcal{G}_1$ and since the proportion $\sum_{i=1}^{t} u_i/|\mathcal{G}_1|$ is very small, it is with a very high probability that $a_{i\,j_i} \neq a_{i\,k_i}$ (a precise formula for the probability can be derived by using the results from the classical occupancy problem, see for example [13], p. 53). This implies that each bit-position of outputs $\breve{\alpha}(x')$ and $\breve{\alpha}(x)$ differ with the probability 1/2, or put in another way, both outputs differ in about half of their bits. The fact that any $MSTg$ with one random cover passes the randomness tests as shown above (i.e. their output bit sequences are not distinguishable from a truly random sequence) is a further indication of this property.

Return to the case of $MSTg$ with two random covers. Let $z_1 = F(s_0)$ and $z_2 = F((s_0 + C) \bmod 2^N)$ be two outputs of $MSTg$, for which s_0 is unknown. From the discussion above $y_1 = \breve{\alpha}(s_0)$ and $y_2 = \breve{\alpha}((s_0 + C) \bmod 2^N)$ are independent, thus the information of z_1 and z_2 does not in general reduce the complexity of finding the correct preimage y_1 of z_1, i.e. $y_1 = \breve{\gamma}^{-1}(z_1)$, which is then used to determine s_0. At last, but not least, notice that the composition of $\breve{\alpha}$ and $\breve{\gamma}$ makes use of two incompatible operations. The first one is the XOR operation applied on cover elements and the second is the partition of a given bit sequence of length N into s subsequences of size $\log_2 r_1, \ldots, \log_2 r_s$, where each subsequence is used as a pointer to the block with randomly generated elements (picking one).

Determining an accurate complexity of computing the seed s_0 for a given output sequence $z_1, \ldots, z_t$ of $MSTg$ appears to be very difficult, but such a rigorously mathematical result would prove the security of the generator.

6.1 The reconstruction of β (with $\breve{\beta} \simeq \breve{\gamma} \circ \breve{\alpha}$)

In a special case, multiple covers, say α and γ, can together be replaced by a single cover β. In other words, there is a cover β such that $\breve{\beta}(x) = \breve{\gamma} \circ \breve{\alpha}(x)$ for all inputs $x \in \mathbb{Z}_n$, where α is a random cover of type $(u_1, \ldots, u_t)$ for $\mathcal{G}_1$, with $\prod_{i=1}^{t} u_i = n$, and γ is a random cover for $\mathcal{G}_2$. The cover β for $\mathcal{G}_2$ is of the same type as α. However the probability for the existence of such a β is very small. Actually, we may carry out a simple test to rule out this case. Firstly, we use the three steps below to build the cover β. By using the two sided transformation on β, see [9], we may assume that the first $t-1$ blocks contain the identity element, denoted here as $\mathbb{1}$. We construct $\beta := [B_1, \ldots, B_t] = (b_{ij})$ of type $(u_1, \ldots, u_t)$ as follows:

(1) Set all $b_{i,1} = \mathbb{1}$, for all $i = 1, \ldots, t-1$.

(2) For each $j_t \in \mathbb{Z}_{u_t}$ construct $x_{j_t} \longmapsto (1, 1, \ldots, 1, j_t)$ and set $b_{t,j_t} = \breve{\gamma}(\breve{\alpha}(x_{j_t}))$. After this step, block B_t is fully assembled. Set $z = \breve{\gamma}(\breve{\alpha}(x_1))$.

(3) Construct blocks $B_{t-1}, \ldots, B_1$. For block B_i use $x_{j_i} \longmapsto (1, \ldots, 1, j_i, 1, \ldots, 1)$, for all $j_i \in \mathbb{Z}_{u_i} \setminus \{1\}$, and set $b_{i,j_i} = \breve{\gamma}(\breve{\alpha}(x_{j_i})) \cdot z^{-1}$.

Then we generate a reasonable amount of random inputs x and verify whether $\breve{\beta}(x) \stackrel{?}{=} \breve{\gamma} \circ \breve{\alpha}(x)$. As the covers used in $MSTg$ are all randomly generated and the block size is at least sixteen, see Remark 6.2, the actual probability of the success is negligible. Practically, the first randomly chosen input fails the verification step already.

Remark 6.1 *The goal of this remark is to show, that random covers of type* $(2, 2, \ldots, 2)$ *should not be used for MSTg in cryptographical applications. Let* α *be a cover of type* $(r_1, \ldots, r_n) = (2, \ldots, 2)$ *for an elementary abelian 2-group* $\mathcal{G}$ *of order* 2^n*. We may identify* $\mathcal{G}$ *with the vector space of dimension* n *over* $\mathbb{F}_2$*, denoted by* V_n*. Without loss of generality, we can assume that the first element of each block of* α *is the identity, i.e. zero vector (see normalization and translation operations in [19]). The set of all second entries of the blocks of* α *span a subspace* U *of* V_n*. With the probability* $\prod_{k=1}^{n}(1 - 2^{-k})$*, which is roughly* $1/4$*,* U *equals* V_n*, i.e. the set of the second elements of* α *forms a basis for* V_n*. In such a case,* α *is a tame logarithmic signature and its induced mapping is linear and therefore the Basis attack in Section 6.4 can be applied. Even if* $U \neq V_n$ *the covers with block size* 2 *are close to the "linear case", i.e. all the elements of* U *can be efficiently factorized.*

Remark 6.2 *Because of the previous remark, we may assume that the size of the blocks of* α *is at least* 4*. In this case we show with a following small example, that a cover* β *with* $\breve{\beta} \simeq \breve{\gamma} \circ \breve{\alpha}$ *can not generally be constructed. Let* α *for* $\mathcal{G}_1 = \mathbb{F}_2^4$ *and* γ *for* $\mathcal{G}_2 = \mathbb{F}_2^3$ *be random covers with two blocks of size* 4*. Without loss of generality, we assume that the first element of each block of* α *and* γ *is the identity, i.e. zero vector, denoted here as* 0000 *(resp.* 000*). The remaining elements of* α *and* γ *are randomly generated and therefore independent. Let* α, γ *and a cover* β *constructed after the three steps of the algorithm above be as follows.*

$$\alpha = \left[\begin{array}{c} 0000 \\ * \\ * \\ a \\ \hline 0000 \\ * \\ * \\ b \end{array}\right], \quad \gamma = \left[\begin{array}{c} 000 \\ h_{1,2} \\ h_{1,3} \\ h_{1,4} \\ \hline 000 \\ h_{2,2} \\ h_{2,3} \\ h_{2,4} \end{array}\right], \quad \beta = \left[\begin{array}{c} 000 \\ * \\ * \\ \breve{\gamma}(a) \\ \hline 000 \\ * \\ * \\ \breve{\gamma}(b) \end{array}\right]$$

Take an input $x = 15$*, i.e.* $x \longmapsto (11, 11)$ *in binary. Then*

$$\breve{\gamma}(\breve{\alpha}(11\,||\,11)) = \breve{\gamma}(a \oplus b) \stackrel{!}{=} \breve{\gamma}(a) \oplus \breve{\gamma}(b) = \breve{\beta}(11\,||\,11).$$

Let $a = (a_1\,||\,a_2)$*,* $b = (b_1\,||\,b_2)$*, where* $a_i, b_i \in \mathbb{F}_2^2$*,* $i = 1, 2$*. Now consider the case such that* $a_i \neq b_i$*,* $a_i, b_i \neq 00$*, for* $i = 1, 2$*. (Clearly, this condition cannot be satisfied if block size is* 2*. With block size 4 it happens in* 6 *of* 16 *cases for each block.) Therefore*

$$\begin{aligned} \breve{\gamma}(a_1 \oplus b_1\,||\,a_2 \oplus b_2) &= \breve{\gamma}(a_1\,||\,a_2) \oplus \breve{\gamma}(b_1\,||\,b_2) \\ h_{1,(a_1 \oplus b_1)} \oplus h_{1,(a_2 \oplus b_2)} &= h_{1,a_1} \oplus h_{2,a_2} \oplus h_{1,b_1} \oplus h_{2,b_2}, \end{aligned}$$

where both a_i, b_i *are different pointers to the block* i *of* γ*, each indexing a randomly generated non-identity element. The sum* $a_i \oplus b_i$ *gives the pointer to the remaining non-identity element of the block* i*. Thus* $\breve{\beta} \simeq \breve{\gamma} \circ \breve{\alpha}$ *implies a linear relation among the elements of* γ*, which is, however, in contradiction to the assumption that all (non-identity) elements of* γ *are independent.*

In the following, we describe some attacks on one cover version of $MSTg$ by exploiting the properties of the system. None of these attacks really succeeds in breaking an $MSTg$ in practice, and therefore, they are more of theoretical value. In general, these techniques cannot be applied if two (or more) covers are used. The main reason is the application of a non-compatible operation which is used to transform the output of the first cover into indices (pointers to positions) in the second cover.

6.2 The Birthday attack

We consider the attempt to find a preimage (seed) for a given one cover generator output, i.e. to factorize with respect to a randomly generated cover. Let $\gamma := [H_1, \ldots, H_s] = (h_{ij})$ be a random cover of type $(r_1, \ldots, r_s)$ for $(\mathcal{G}, \cdot)$, $|\mathcal{G}| = m$, and $\breve{\gamma} : \mathbb{Z}_n \longrightarrow \mathcal{G}$ with $n = \prod_{i=1}^{s} r_i$. Assume that $z = \breve{\gamma}(x)$ is given for some $x \in \mathbb{Z}_n$ with $\tau_\gamma^{-1}(x) = (j_1, \ldots, j_s)$, $j_i \in \mathbb{Z}_{r_i}$. We define $\gamma_1 := [H_1, \ldots, H_{\lceil s/2 \rceil}]$ and $\gamma_2 := [H_{\lceil s/2 \rceil + 1}, \ldots, H_s]$. In other words, we split γ into two covers by taking the first $\lceil s/2 \rceil$ blocks to the first one and the rest to the second. Then $\breve{\gamma}(x) = \breve{\gamma}_1(x_1) \cdot \breve{\gamma}_2(x_2)$, where $\breve{\gamma}_1(x_1) = \prod_{i=1}^{\lceil s/2 \rceil} h_{i\,k_i}$ and $\breve{\gamma}_2(x_2) = \prod_{i=\lceil s/2 \rceil + 1}^{s} h_{i\,k_i}$, with $x_1 \longmapsto (k_1, \ldots, k_{\lceil s/2 \rceil})$ and $x_2 \longmapsto (k_{\lceil s/2 \rceil + 1}, \ldots, k_s)$, for some $k_i \in \mathbb{Z}_{r_i}$ and $i = 1, \ldots, s$. Notice that in general $k_i \neq j_i$ (z may have more than one factorization with respect to γ). First construct a table T of pairs (u, v) with randomly chosen $u = (u_1, \ldots, u_{\lceil s/2 \rceil})$, $u_i \in \mathbb{Z}_{r_i}$, and $v = \breve{\gamma}_1(u)$. The attack works as follows: for random $w = (w_{\lceil s/2 \rceil + 1}, \ldots, w_s)$, $w_i \in \mathbb{Z}_{r_i}$, compute a product $g = z \cdot \breve{\gamma}_2(w)^{-1}$. If there is a pair (u, v) in T such that $g = v$, then we have found a factorization $x' = u \| w$, i.e. $x' \mapsto (u_1, \ldots, u_{\lceil s/2 \rceil}, w_{\lceil s/2 \rceil + 1}, \ldots, w_s)$. Using the well known Birthday paradox we see that if the size of T is roughly $\mathcal{O}(\sqrt{m})$, then the probability of such collision is about $1/2$. As a result, we have found x' such that $\breve{\gamma}(x') = z$, but not necessarily $x' = x$. In that case, to find the correct seed, we have to rerun the attack.

For the Birthday attack to work, the elements $\breve{\gamma}_1(u)$, $\breve{\gamma}_2(w)$, must be randomly chosen from $\mathcal{G}$. Using the results from the previous section we assume the elements to be random if proportions of the covers are appropriate. However, the domain of both $\breve{\gamma}_i$ is roughly $\sqrt{n}$. If we choose n such that $\sqrt{n} \approx m$, we expect that on average about $1/e$ or 37% of elements are not generated at all by each of $\breve{\gamma}_i$ (see, for example, [3]). Therefore, for this attack to work, we require that $n > m^2$. This implies, that the probability that the found preimage is equal to x (the correct seed) is on average m/n, i.e. smaller than $1/|\mathcal{G}|$.

6.3 The Meet-in-the-middle attack

The following brute-force type of attack is a modification of the previous method. It allows an attacker to recover the seed by using a time and memory trade-off and is generally called the Meet-in-the-middle attack. Let $\gamma := [H_1, \ldots, H_s] = (h_{ij})$ be a random cover of type $(r_1, \ldots, r_s)$ for $\mathcal{G}$. Assume that $z = \breve{\gamma}(x)$ is given for some x with $\tau_\gamma^{-1}(x) = (j_1, \ldots, j_s)$, $j_i \in \mathbb{Z}_{r_i}$. As before, we define γ_1, γ_2, and construct table T of all possible pairs (u, v) with $u = (u_1, \ldots, u_{\lceil s/2 \rceil})$, $u_i \in \mathbb{Z}_{r_i}$, and $v = \breve{\gamma}_1(u)$. Here the size of T is roughly $\mathcal{O}(\sqrt{n})$, where $n = \prod_{i=1}^{s} r_i$. Now for each chosen $w = (w_{\lceil s/2 \rceil + 1}, \ldots, w_s)$, $w_i \in \mathbb{Z}_{r_i}$, compute the product $g = z \cdot \breve{\gamma}_2(w)^{-1}$. If there is a pair (u, v) in T such that $g = v$, then we have found a factorization $x' \mapsto (u_1, \ldots, u_{\lceil s/2 \rceil}, w_{\lceil s/2 \rceil + 1}, \ldots, w_s)$. In summary, this attack requires $\mathcal{O}(\sqrt{n})$ memory and $\mathcal{O}(\sqrt{n})$ time. Compared to the Birthday attack above, this method is deterministic, and therefore allows to find the correct value of the seed x. More precisely, it finds all preimages

for a given output z.

6.4 The Basis attack

We consider a trivial attempt to find a preimage (seed) for a given generator output if elementary abelian 2-groups are used as underlying groups. Let $MSTg$ be a one cover generator based on γ of type $(r_1, \ldots, r_s)$ for $\mathcal{G}_2$, with $|\mathcal{G}_2| = 2^M$ and $\breve{\gamma} : \mathbb{Z}_{2^N} \longrightarrow \mathbb{Z}_{2^M}$. Define $\mathcal{S} := \breve{\gamma}(\mathbb{Z}_{2^N})$. Thus $\mathcal{S}$ is a subset of $\mathcal{G}_2$ and the ratio $\rho = 2^N/|\mathcal{S}|$ may be viewed as the average number of representations for each element of $\mathcal{S}$ with respect to γ. Due to the connection between the generation of random covers and the occupancy problem, see [17], we can derive an approximation for the ratio given by the following formula

$$\rho \approx \lambda\big(\frac{e^\lambda}{e^\lambda - 1}\big)$$

where $\lambda = 2^N/|\mathcal{H}|$, and $\mathcal{H}$ is the smallest subgroup of $\mathcal{G}_2$ containing $\mathcal{S}$. As a matter of linear algebra, we may find a maximal subset of linearly independent vectors which come from all the blocks of γ. By using the two sided transformation on γ, see [9], we may assume that the first $s-1$ blocks contain the zero vector. The linearly independent vectors together with the zero vectors form a cover which allows an efficient factorization of a certain number of products of γ. This amount is approximated by $\frac{1}{\rho}\prod_{i=1}^{s}(k_i+1)$, where $k_i = \lceil \log_2 r_i \rceil$. Thereby we assume, that $\log_2 r_i$ linearly independent vectors are chosen from the block with size r_i. Therefore the probability that a given seed could be computed correctly is given by

$$\approx \frac{1}{\rho}\prod_{i=1}^{s}\frac{(k_i+1)}{r_i}$$

As a result, if ρ or/and r_i are increased, this probability decreases. The usage of more than one covers decreases the probability significantly. More precisely, it equals the product of probabilities for all covers. Also, as mentioned before, if parameters of the generator are chosen properly (ρ is large), the probability that the correct seed could be found using this method is negligible.

6.5 The one cover one bit attack

All attacks mentioned above study the security from the static point of view – the output of a single generator cycle. This attack exploits the relationships between bits (patterns) in the output sequence from a one cover version of $MSTg$ if elementary abelian 2-groups are used. Let γ be a random cover of type $(r_1, \ldots, r_s)$ for $\mathbb{Z}_{2^M}$, $\prod_{i=1}^{s} r_i = 2^L$, used to construct $MSTg$. We define γ_u to be a cover constructed by taking only the u-th bit of every element of γ, for chosen $u \in \{1, \ldots, M\}$. Clearly, such cover induces a function $\breve{\gamma}_u : \mathbb{Z}_{2^L} \longrightarrow \mathbb{Z}_2$. Notice, that the factorization of g with respect to γ results in a subset of preimages given by the factorization of the u-th bit of g with respect to γ_u. More precisely the first mentioned equals the intersection of factorizations with respect to all γ_u through $u = 1, \ldots, M$. For the attack described here, we assume, that the canonical bijection is used to pair input x with indices $(j_1, \ldots, j_s)$, where least significant bits are used for j_s and most significant bits for j_1, see Section 4.1. The output of γ_u after first t cycles of generator could be received from

the output sequence by collecting bits $(u + k \cdot M)$, for $k = 1, 2, \ldots, t$. Denote those bits by $z^{(1)}, \ldots, z^{(t)}$. Also denote by h_{ij} the bit on the j-th position in block i of γ_u.

Firstly, we consider the case, where $C = 1$ is the constant used to update the generator input. Let $x^{(0)} \longmapsto (j_1^{(0)}, \ldots, j_s^{(0)})$ be the seed s_0 to be recovered. Then for $x^{(k)} \longmapsto (j_1^{(k)}, \ldots, j_s^{(k)})$ in cycle k, we compute

$$z^{(k)} = \breve{\gamma}_u(x^{(k)}) = \underbrace{\Big(\bigoplus_{i=1}^{s-1} h_{i\, j_i^{(k)}}\Big)}_{=:\, b^{(k)}} \oplus h_{s\, j_s^{(k)}}$$

where $j_s^{(k)} = (j_s^{(0)} + k) \bmod r_s$. Until $j_s^{(0)} + k < r_s$, the value of $b^{(k)}$ remains constant, i.e. 0 or 1. Let's assume, that $j_s^{(0)} + t < r_s$ for some small $t > 1$. Then the output sequence $z^{(1)}, z^{(2)}, \ldots, z^{(t)}$ is equal to either $h_{s\, j_s^{(1)}}, h_{s\, j_s^{(2)}}, \ldots, h_{s\, j_s^{(t)}}$ or $\overline{h}_{s\, j_s^{(1)}}, \overline{h}_{s\, j_s^{(2)}}, \ldots, \overline{h}_{s\, j_s^{(t)}}$, where $\overline{y}$ denotes the negation. In other words, assuming that $j_s^{(0)} + t < r_s$ for some small chosen parameter t, if the collection of successive elements h_{sj} in the last block of γ_u is equal to $z^{(1)}, \ldots, z^{(t)}$ or $\overline{z}^{(1)}, \ldots, \overline{z}^{(t)}$, then $j_s^{(0)}$ is equal to the value one less than the position of the first element $h_{s\, j^{(1)}}$, i.e. it is equal to the index of element just before $h_{s\, j^{(1)}}$. Similarly, by collecting bits $u, (u + r_s \cdot M), (u + r_s^2 \cdot M), \ldots$, we may recover the value of $j_{s-1}^{(0)}$. In general, to recover $j_s^{(0)}, j_{s-1}^{(0)}, \ldots, j_{s-c}^{(0)}$, we require a sequence of length $M \cdot \prod_{i=0}^{c} r_{s-i}$ bits, i.e. the output of $(r_s \cdot r_{s-1} \cdots r_{s-c})$ cycles. Notice, that for properly chosen parameters, for example input size of 256 bits, the size of the output sequence required to recover only half of the seed bits is $M \cdot 2^{128}$, and therefore infeasible.

If $C \neq 1$ the attack works similarly, but the choice of the output bits used to recover $j_i^{(0)}$ must be done according to the constant C. Therefore if, as proposed, a large prime is used for C, each of the pointers to the blocks may be changed after a single cycle. Then, only to recover $j_s^{(0)}$, i.e. to construct the series of outputs from cycles k such that the value $b^{(k)}$ is fixed, may require an output sequence of quite extensive length.

7 More on $MSTg$

An interesting implication of the results of the previous sections is the fact that $MSTg$ can be used to initialize $MSTg$. This may remind us to the well known "hen and egg" problem, but there is a very important difference. For the initialization of $MSTg$ we require only that the used generator has good statistical properties. Actually, this condition may possibly be lessened and we may use some "weaker" generator. This hypothesis, however, needs to be properly tested before applying, and as there are enough good alternatives, we do not recommend it. The volume of random data used to initialize covers is largely restricted, on the other hand the volume of data produced by $MSTg$ is variable and mostly extensive. Moreover, the generator used for initialization of $MSTg$ could be completely known (including seed), i.e. covers used in $MSTg$ may be made public. Keeping this additional information (or part of it) private will harden the attacks on $MSTg$.

7.1 Initialization with $MSTg$

Here we present a possible solution for problem of initializing $MSTg$ with another $MSTg$. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be elementary abelian 2-groups as defined before. Let GEN be a given (fixed) $MSTg$ seeded with a truly random seed s_I. The output bitstream from the GEN is cut into bit sequences of appropriate size and used as entries in covers α, $\alpha_1, \ldots, \alpha_k$ and γ for $\mathcal{G}_1$ and $\mathcal{G}_2$ to initialize $MSTg$. The information required to recover the given sequence using such fixed system are two seed values s_I and s_0. Typically, we would keep both seeds private and use some standard built-in all-round GEN. Notice that GEN does not need to be cryptographically secure, so we may use generator with parameters much weaker than for the secure usage, e.g. we may use $MSTg$ with only one cover and 64 bit input. Also any other suitable (pseudo-)random bit generator can be used to initialize $MSTg$ instead. From the attacker point of view, the only known information are the cover(s) used in GEN and the output sequence from $MSTg$. For him, to start the attack on s_0, he first have to recover generator $MSTg$ which is equivalent to finding the seed s_I for GEN. We may also envisage this as following. By choosing a seed s_I for GEN, say, for example, 128 bits long, we effectively choose one from 2^{128} possible generators. Additionally, each such chosen $MSTg$ itself is seeded with a seed s_0.

7.2 Updating covers of $MSTg$

Firstly, we have to mention that the most practical PRNGs used nowadays are based on a fixed algorithm or function. In the case of $MSTg$ we are actually able to use each time newly initialized and therefore different generator. We recommend this as a good practice. The covers used in $MSTg$ are initialized from a random stream, so we may consider the generator function F as a random function, although F is fixed for a given $MSTg$. An innovative idea is to randomly change F after reaching some predefined number of cycles. This can be achieved by the construction with a feedback from the outputstream used directly (replace some/all entries in covers with output bit sequence), indirectly (use output bits to reseed GEN and generate sequence used to replace some/all entries in covers), see Figure 4, or by using additional external seeding information. Notice that we may use the output of $MSTg$ to update also the entries of GEN, in which case the whole process of generating $MSTg$ does not rely on any fixed generator or any fixed set of data. Clearly, without any additional external data (e.g. another seed for GEN), the entropy of the system remains constant. Updating however increases the complexity of possible attacks, and especially provides the control over the length of the sequence generated from a static system.

7.3 $MSTg$-based one-time-pad

As a special case, this method of using $MSTg$ allows to simulate the concept of one-time-pad. Namely, to obtain a sequence of random bits of a certain length, we randomly generate an $MSTg$, which is then used once to generate the bit sequence. Thereafter, the covers of the generator are discarded. For generating a new bit sequence, a new instance of $MSTg$ has to be created. Also the GEN is changed each time.

The generated one-time-pad can be easily used, for example, to encrypt stream of cleartexts (divided in M-bit blocks) as given in the Figure 5. This is a known concept of a stream cipher. The initialization of the generator (the seed values at least) must differ for each single

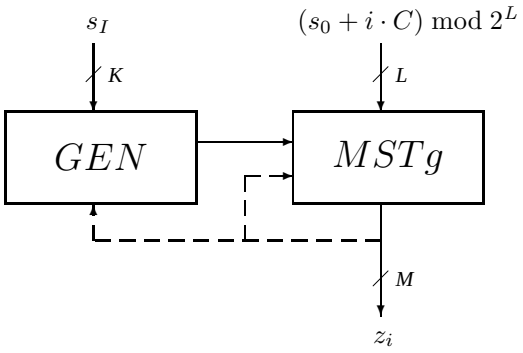

Figure 4: MSTg with GEN initialization (and feedback).

use (e.g. document, time period, etc.), so that no two data share the same key. Here under the key we understand the output sequence of the generator which is in one-to-one correspondence with the seed(s) of the generator. If we would use the generator with the same initialization to encrypt two different cleartexts, the trivial attack by XORing both outputs allows the adversary to recover the difference of both cleartext sequences. Then any partial information about one cleartext reveals the appropriate part of the second cleartext sequence. Notice that usage of the updating feedback from an encrypted output stream increases the entropy of the system. In such a case, the resulting keystreams for two distinct cleartexts differ even if the same seed is used.

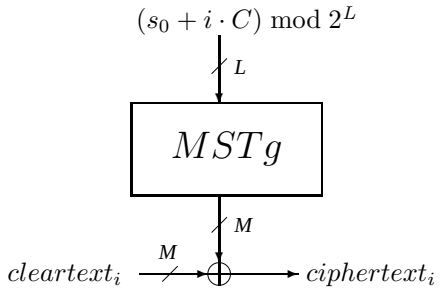

Figure 5: MSTg-based one-time-pad.

Remark 7.1 *To reduce the space needed to store covers, we consider the construction of two covers $MSTg$, with $F = \breve{\gamma} \circ \breve{\alpha}$, by using a single random cover $\alpha = (a_{ij})$ for $\mathcal{G}_1$. Firstly, we choose a surjection $f : \mathcal{G}_1 \longrightarrow \mathcal{G}_2$. For example, if elementary abelian 2-groups are used as proposed, we take f which maps each N-bit vector in an M-bit vector by taking its first M bits (or any other fixed M positions). We construct cover $\gamma = (h_{ij})$ for $\mathcal{G}_2$ (of the same type as α) by taking $h_{ij} = f(a_{ij})$ for each a_{ij} in α. In all tested two covers versions of $MSTg$ this reduces the memory demand by 2/6 and may improve the performance as well, e.g. in cases where reduced memory requirements imply storing of α in CPU cache memory. Another possibility to reduce the cover size and increase the performance is given by the choice of the parameters ℓ, n, m. As an example we take the version with input size 256 bits and output size 128 bits. If $\mathcal{G}_1, \mathcal{G}_2$ are given with $|\mathcal{G}_1| = n = \ell = 2^{256}$ and $|\mathcal{G}_2| = m = 2^{128}$, the generator with block size 256 requires 384 kB of memory, with block size 16 then 48 kB, see Table 3.*

A change of $\mathcal{G}_1$ with $|\mathcal{G}_1| = n = 2^{192}$ reduces the memory demand for the version with block size 256 to 320 kB, with block size 16 to 40 kB. This makes a difference of 1/6, though the security level remains unchanged. What will alter is the the entropy of the system, which however remains large enough for the change to be noticed. Of course, to achieve a desired result both mentioned methods may be combined together.

8 Parallelization of $MSTg$

In recent years, the development of modern graphics cards shows a significant growth in performance, programmability, and capacity for general usage. The performance potential of massively parallel Graphics Processing Units (GPU) often exceeds the potential of conventional processors. Therefore its application appears in cryptography as an intuitive idea. Several authors indeed show impressive results of their GPU implementation of various cryptographic techniques, such as DES and AES. Also known is the use of these systems in cryptanalysis. In this section we study the possibilities of parallelization of the generator for the implementation on GPU.

As before, let $\mathcal{G}_1, \mathcal{G}_2$ be elementary abelian 2-groups, $\mathcal{G}_2 \subseteq \mathcal{G}_1$, with $|\mathcal{G}_1| = 2^N, |\mathcal{G}_2| = 2^M$. For simplicity we set $k = 0$ and generate two random covers α and γ for $\mathcal{G}_1$ and $\mathcal{G}_2$, where α is of type $(u_1, \ldots, u_t)$ and $\prod_{i=1}^{t} u_i = 2^L$, and cover γ of type $(r_1, \ldots, r_s)$ and $\prod_{j=1}^{s} r_j = 2^N$. The generating function $F = \breve{\gamma} \circ \breve{\alpha}$ is the composition of induced mappings $\breve{\alpha}$ and $\breve{\gamma}$. We consider the parallelization of $MSTg$ on three possible levels.

1. Parallelization of function F

The generator algorithm, see Algorithm 1 from Section 3, operates in so-called counter mode, where in each cycle i the output z_i is computed via the generator function F and input s_i, i.e. $z_i = F(s_i)$. Afterwards, the value of s_i is updated by adding the constant C, which is typically some large prime. As we can see, for a fixed F the output z_i depends only on the value s_i. Thus, instead of a single generating loop, we may run k identical generators with initial values $s_1, \ldots, s_k$, where $s_j = (s_0 + j \cdot C) \bmod 2^L$, in parallel, and update inputs s_j in every cycle with the value $k \cdot C$. The k outputs are concatenated to a single output bitstream, as shown in the Figure 6.

Notice that, as the produced bit chunks are all random and independent from each other, see Section 6, the outputs can be concatenated in an arbitrary order, i.e. not necessarily in the order of their indices. However, if they are used to encrypt data, as described in the previous section, this arrangement must be deterministic to enable a correct decription.

In the experiment we investigate the speed-up of the generator if run as multiple independent threads (65536) on a single graphics card. Each thread executes a small number of generator cycles. Threads are put to a waiting queue, from which they are taken and executed by one of the processing units (cores). The usage of GPU raises additional load caused by data transfer between CPU and graphics card. Also level of optimization by the compilation of GPU source is far lower than for CPU. On account of this, the CPU-optimized version runs more than 40 times faster than the version on a single GPU core, as shown in Table 3. Here we see the greatest potential for increasing the performance of GPU code. Further improvement can be achieved by optimizing the usage of graphics card memory or by using

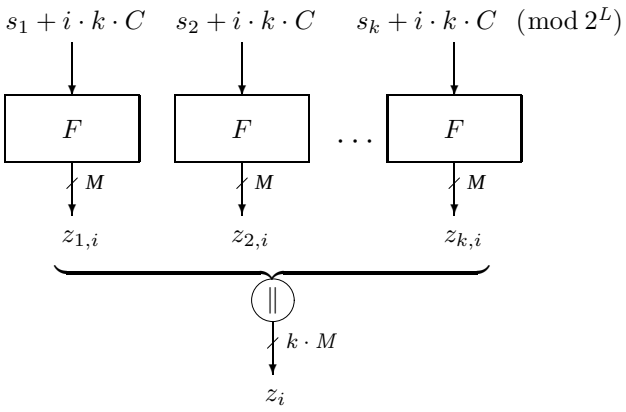

Figure 6: Parallel MSTg.

a higher range hardware. Table 3 summarizes the experimental data with several parameter settings. Notice, that by using the graphics card with 128 cores, the speed of implemented GPU-$MSTg$ is roughly some hundreds of megabytes per second.

2. Bitslicing

Bitslicing was first suggested by Biham for fast software implementation of DES [1]. This technique, instead of handling M-bit words in one operation, processes M one-bit operations simultaneously on a SIMD parallel computer (Single Instruction Multiple Data). In $MSTg$ with one cover only, say γ, each of the output M bits can be produced independently. This is due to the fact that the group operation is bitwise XOR. We can envisage the bitslicing within γ as construction of M parallel one-bit generators with covers $\gamma_1, \ldots, \gamma_M$ for $\mathbb{Z}_2$, where γ_j is filled with j-th bits of all entries of γ. In other words, if γ is stored as a matrix, the γ_j corresponds to its j-th column. Each γ_j is then assigned to a single processing unit. As γ_j is represented as a bit-vector, the computing $\breve{\gamma}_j(x)$ is realized as XOR of the bits on positions given by the input x (more precisely by $\tau_\gamma^{-1}(x)$). This is done independently on all $\gamma_1, \ldots \gamma_M$. Clearly, each processing unit has to start with the same seed s_0 and to update its input in the identical way. This matches exactly the basic principle of SIMD computation, i.e. an execution of identical instructions on different data. The outputs of all processing units are concatenated into a single output stream. Similarly, as with the previous method, the order in which the produced bits are arranged together does not need to coincide with their index j. Actually, instead of collecting M one-bits after each round, we may concatenate chunks, say bytes, assembled on processing units after some fixed number of rounds.

If more than one cover is used, say two, with the generator function $F = \breve{\gamma} \circ \breve{\alpha}$, then each output $F(x)$ depends on the value $\breve{\alpha}(x)$ and this further on the value of input x. We still may apply bitslicing within each of the covers, but before proceeding with the computation on γ, the value of $\breve{\alpha}(x)$ has to be fully assembled and accessible to all processing units (all γ_j slices). In GPU environment this leads to delays caused by the necessary synchronization of all processing units and to the serialization due to a simultaneous access to the single memory bank (which holds the value of $\breve{\alpha}(x)$). This implies, that the bitslicing technique is more suitable for one cover versions of $MSTg$.

input size ($\log_2 \ell$) [bit]	output size ($\log_2 m$) [bit]	block size	blocks	covers size [kByte]	security level * [bit]	CPU performance [MByte/s]	GPU ** 1 Core performance [MByte/s]	GPU ** 128 Cores performance [MByte/s]
512	384	256	64	1792	128	39	0,90	120
		16	128	224		-	0,90	324
		4	256	112		-	0,52	356
384	256	256	48	960	128	52	0,96	146
		16	96	120		-	0,90	455
		4	192	60		-	0,52	466
256	128	256	32	384	128	61	1,02	80
		16	64	48		-	1,01	407
		4	128	24		-	0,55	527
192	64	256	24	192	128	56	0,84	83
		16	48	24		-	0,73	495
		4	96	12		-	0,39	552
64	64	256	8	32	not cryptogr.	357	2,75	620
		16	16	24	secure	-	2,29	1180

All tested versions use two covers and an appropriate constant C from $p_{64}, \ldots, p_{512}$, see Table 1.

* As the security level we define the logarithm of the average number of preimages (seeds) for an output of a single generator cycle, see Section 6. We simply consider here the complexity of the brute-force attack recovering the preimage for a given output of the compression mapping from $\mathbb{Z}_\ell$ into $\mathbb{Z}_m$. Assuming the cryptographic hypothesis from Section 2 is true, the complexity of the known attacks is obviously a lot higher.

** Modern graphics cards vary massively by various parameters such as memory, performance, power consumption and price. Supported by the programming tools are the products from manufacturers Nvidia and ATI. For testing purposes, an Intel-based computer was equipped with a graphics card in lower price range (about 130 USD) and installed with CUDA C programming environment from NVIDIA [14]. Further technical details are given in the following table.

PC	Intel 2.13 GHz, 2MB Cache, 4GB RAM
OS	64-bit Ubuntu Server, 12.04 LTS
Graphics Card (GPU)	GeForce GTX 550Ti (PCI-Express)
RAM	1 GB DDR5
Clock	2 GHz
Max. Threads per Block	1024
Multicores/Cores	4/128
CUDA version	2.1

Table 3: The performance of MSTg on CPU and GPU.

We have done small experiment with one cover version of $MSTg$ with 256 bit input, 128 bit output, and block size 256, by using a single computing unit of GPU. The bitsliced version requires on average $1.83\,\mu s$ per cycle, compared with $7.20\,\mu s$ per cycle when all 128 bits are computed at once by using the Type 1. parallelization. The latter corresponds to about $0.056\,\mu s$ per cycle and bit. This shows that for $MSTg$ the bitslicing technique is less efficient than the previous parallelization method. The reason for this performance loss is the fact, that in the bitsliced version all processing units carry out the identical computation to update the input (modulo 2^L addition), i.e. one update per single bit created. On the other hand,

in Type 1. parallelized version we update input once per 128 bits created.

However, the bitslicing implies that a single processing unit does use (and needs to store) only a "slice" of the cover, e.g. $1/M$ of the cover data. This could be useful in cases where devices with restricted resources are used, or in systems of low bit-width architecture, e.g. microcontrollers, smart cards, etc.. Clearly, more than one bit could be taken as a slice, or both parallelization techniques could be combined together to optimize performance of an embedded system.

3. Multiple GPUs

The last, and somehow trivial possibility is the construction of a GPU cluster (client/server network; multi-GPU system) with parallel operating GPUs. Nowadays, such systems are already used in cryptanalysis, for example to crash MD5 hashes or to recover passwords by using brute-force methods.

9 Conclusion

In this paper we introduce a new approach to designing pseudorandom number generators. We define a class of PRNGs, called $MSTg$, based on random covers for finite groups. The reason of using random covers is that these induce functions that behave randomly, which means that they possess the properties of one-way functions. In particular, the class of elementary abelian 2-groups has been proposed for an efficient realization of $MSTg$. For this class of groups, we have made extensive tests of randomness of the outputs generated by $MSTg$ by using the NIST Statistical Test Suite and the Diehard Battery of Tests of Randomness. The results of all tested versions with all three tested block sizes indicate the high quality of the output bit sequences of $MSTg$. We have discussed the security of these generators, providing the argumentation proving that they are secure for use in cryptographic applications and proposed method of using them in practice. The high potential of these very simple, efficient and strong pseudorandom number generators is the fact that they possess further unique features allowing any user to generate his own instance. For this purpose we propose a method for initializing $MSTg$ with another $MSTg$. Users are allowed to choose custom parameters for the set-up, such as input, output and block size, number of covers and underlying groups used, or security level. We have also investigated possibilities for parallelization of the generator algorithm, showing that $MSTg$ is well suited for implementation on multiprocessing systems, like GPUs. The experiments evidence a significant increase in the performance if implemented on a graphics card.

References

[1] E. Biham, A fast new DES implementation in software, *Lecture Notes in Computer Science*, 1267, pp. 260–272, 1997.

[2] L. Blum, M. Blum, and M. Shub, A simple unpredictable pseudo-random number generator, *SIAM J. Computing*, 15, pp. 364–383, 1986.

[3] W. Feller, An Introduction to Probability Theory and Its Applications, John Wiley & Sons, Vol. 1, New York, 1957.

[4] D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3^{rd} Edition, Addison-Wesley, Reading, Massachusetts, 1998.

[5] W. Lempken, S.S. Magliveras, Tran van Trung, W. Wei, A public key cryptosystem based on non-abelian finite groups, *J. Cryptology*, 22, pp. 62–74, 2009.

[6] S. S. Magliveras, B. A. Oberg and A. J. Surkan, A New Random Number Generator from Permutation Groups, In *Rend. del Sem. Matemat. e Fis. di Milano*, LIV, pp. 203–223, 1984.

[7] S. S. Magliveras, A cryptosystem from logarithmic signatures of finite groups, In *Proceedings of the 29'th Midwest Symposium on Circuits and Systems*, Elsevier Publishing Company, pp. 972–975, 1986.

[8] S. S. Magliveras and N. D. Memon, Random Permutations from Logarithmic Signatures, *Computing in the 90's, First Great Lakes Comp. Sc. Conf., Lecture Notes in Computer Science* Springer-Verlag, 507, pp. 91–97, 1989.

[9] S. S. Magliveras, D. R. Stinson and Tran van Trung, New approaches to designing public key cryptosystems using one-way functions and trapdoors in finite groups, *J. Cryptology*, 15, pp. 285–297, 2002.

[10] P. Marquardt, P. Svaba and Tran van Trung, Pseudorandom number generators based on random covers for finite groups, *Des. Codes Cryptogr.*, 64, pp. 209–220, 2012.

[11] G. Marsaglia, The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness, 1995. http://www.stat.fsu.edu/pub/diehard/

[12] M. Matsumoto and T. Nishimura, Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator, *ACM Transactions on Modeling and Computer Simulation*, 8(1), pp. 3–30, 1998.

[13] A. Menezes, P. van Oorschot and S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

[14] NVIDIA Corporation, NVIDIA CUDA, Developer Zone, 2012. http://developer.nvidia.com/category/zone/cuda-zone

[15] R. L. Rivest The RC4 Encryption Algorithm, RSA Data Security, Inc., Mar. 1992.

[16] A. Rukhin, et. al., Statistical test suite for random and pseudorandom number generators for cryptographic applications, *NIST Special Publication 800-22*, Revised April 2010, National Institute of Standards and Technology, http://csrc.nist.gov/rng

[17] P. Svaba and Tran van Trung, On generation of random covers for finite groups, *Tatra Mt. Math. Publ.*, 37, pp. 105–112, 2007.

[18] P. Svaba and Tran van Trung, Public key cryptosystem MST_3: cryptanalysis and realization, *J. Math. Cryptol.*, 4, pp. 271–315, 2010.

[19] P. Svaba, Covers and Logarithmic Signatures of Finite Groups in Cryptography, *University of Duisburg-Essen, Germany*, PhD. thesis, 2011.