Pseudorandom Bit Generation with Asymmetric Numeral Systems

Josef Pieprzyk[1,2], Marcin Pawłowski[1], Paweł Morawiecki[1], Arash Mahboubi[3], Jarek Duda[4], and Seyit Camtepe[2]

[1] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
[2] Data61, CSIRO, Sydney, Australia
[3] School of Computing and Mathematics, Charles Sturt University, Port Macquarie, Australia
[4] Institute of Computer Science and Computer Mathematics, Jagiellonian University, Cracow, Poland

Abstract. The generation of pseudorandom binary sequences is of a great importance in numerous applications stretching from simulation and gambling to cryptography. Pseudorandom bit generators (PRBGs) can be split into two classes depending on their claimed security. The first includes PRBGs that are provably secure (such as the Blum-Blum-Shub one). Security of the second class rests on heuristic arguments. Sadly, PRBG from the first class are inherently inefficient and some PRBG are insecure against quantum attacks. While, their siblings from the second class are very efficient, but security relies on their resistance against known cryptographic attacks.
This work presents a construction of PRBG from the asymmetric numeral system (ANS) compression algorithm. We define a family of PRBGs for 2^R ANS states and prove that it is indistinguishable from a truly random one for a big enough R. To make our construction efficient, we investigate PRBG built for smaller $R = 7, 8, 9$ and show how to remove local correlations from output stream. We permute output bits using rotation and Keccak transformations and show that permuted bits pass all NIST tests. Our PRBG design is provably secure (for a large enough R) and heuristically secure (for a smaller R). Besides, we claim that our PRBG is secure against quantum adversaries.

Keywords: Pseudorandomness, Entropy Encoding, Compression, Asymmetric Numeral Systems, Indistinguishability, ANS, PRBG, PRNG, Keccak

1 Introduction

Since ancient times, people have been engaged in gambling, where tossing a coin or rolling a dice is a source of randomness. The study of physics has led to discovery of many processes, whose behaviour is inherently probabilistic. Particle diffusion, thermal and shot noises or radioactive decay are good examples of such processes. Random bit generators (RBGs) built using physical probabilistic processes are able to produce sequences of bits with a very high rate. Their main two drawbacks, however, are the need for a periodic calibration and the fact that randomness is accessible locally only. If random bits need to be shared by two or more parties, this becomes a nontrivial problem. An obvious solution is to apply a deterministic algorithm, which takes a short random bit sequence (also called "seed") and extends it to a much longer one.

A source of such binary sequences is called a pseudorandom bit generator (PRBG). Sharing generated bits becomes much easier as it is enough for parties to agree on a common seed and an algorithm. Unfortunately, generated bits are no longer truly random. However, for some applications, it suffices if pseudorandom sequences "look" random or more precisely, they pass some statistical tests (such as the ones recommended by NIST [17]). First PRBG solutions are based on linear feedback registers and linear congruences [16]. They are very fast but once you know enough bits, it is possible to calculate their seeds and consequently predict the next bits. The current explosion of Internet services has changed the randomness landscape dramatically. To guarantee an appropriate level of security, the services are built using a variety of cryptographic tools. To function properly, such tools consume large volumes of (pseudo)random bits. In many circumstances, generated randomness needs to be replicated in a few geographically distant locations. This puts security considerations at the forefront.

There is an interesting question about how to verify randomness of PRBG or in other words, which statistical tests need to be passed so a tested PRBG can be claimed to behave as a truly random generator. Alternatively, one can ask if there is a universal test, which, when passed, assures that all other statistical tests hold. This question has been answered in the affirmative by Yao [23]. This is the so-called *next-bit test*. Given an adversary with polynomially-bounded computing resources who can observe a polynomial-size output sequence generated by PRBG. It is said that PRBG passes the next-bit test if the adversary is able to predict the next bit with probability $1/2 + \varepsilon$, where ε is negligible. A *distinguisher* is an algorithm that implements the next-bit test. PRBG is called cryptographically strong (or CSPRBG) if it passes the next-bit test or alternatively, there is no distinguisher that can tell apart it from a truly random source. There are two classes of CSPRBG: one based on a heuristic argument and the other on an intractability assumption. The first class includes numerous designs based on nonlinear feedback shift registers (NFSR). Trivium, Snow and Sober (see eStream portfolio *https://www.ecrypt.eu.org/stream/*) are good examples of such solutions. The second class includes a RSA-based PRBG that assumes intractability of integer factorisation [1] and a Bum-Blum-Shub PRBG, whose security rests on intractability of quadratic residuosity [4]. Despite a well-developed theoretical framework and extensive range of designs, there are many examples of security failures due to the usage of a weak or small-entropy source. An early version of Netscape secure socket layer (SSL) encryption turns out to be completely insecure due to weak randomness of pseudorandom number generator (PRNG), which is fed by a highly predictable seed [12]. Similarly, a PRBG based on elliptic curves (intended to be a NIST standard) has been found to be compromised due to special selection of seed constants [20].

Asymmetric numeral systems (ANS) is a relatively new family of compression algorithms invented by Jarek Duda [11]. It has taken the IT industry by storm. ANS is being used by major IT players (such as Apple, Facebook, Google, Linux) as a preferred compression algorithm. Duda in his work [11] suggests that ANS can be a source of cryptographically strong pseudorandom bits. An attractive feature of ANS is that it generates binary encodings with different lengths, which complicates security analysis. Besides, for each fixed-length ℓ, it can output any ℓ-bit sequence (the actual bit string depends on the ANS state and input symbol). In fact, an ANS encoding table can be seen as a large and dynamic S-box, whose outputs (bit sequences of various lengths) are controlled by input symbols and internal states.

Motivation. Let us point at possible applications of ANS for pseudorandom bit generation.

- Truly random bit generators based on physical processes (such as thermal noise or radioactive decay) need to be calibrated from time to time [2]. An expensive calibration can be avoided by using ANS, which removes redundancy/bias. In other words, calibration can be replaced by an "intelligent" ANS that senses the probability fluctuations and automatically adjusts compression parameters.
- Similarly, software-generated randomness from a local entropy of operating system events can be compressed giving a very close to uniformly random probability distribution [10]. As a choice of such events and their statistics can vary from time to time, ANS parameters can be adjusted accordingly.
- The above two points talk about local randomness that does not need to be shared. There are circumstances when two or more parties wish to agree on common randomness by, for example, observing the same fragment of the moon or collecting signals from a chosen pulsar. ANS can remove redundancy from shared randomness and if ANS applies a common pre-agreed secret, the parties can establish a common randomness (that is also secret).

- ANS compresses a sequence of symbols of an arbitrary probability distribution into a binary sequence with a low (perhaps negligible) redundancy. Intuitively, such binary sequence should behave as a truly random one.
- ANS can translate uniform probability distribution into an arbitrary probability distribution by switching encoder and decoder. In fact, a concatenation of ANS compression and decompression can translate an arbitrary probability distribution to another arbitrary one. This application is very useful in simulations, where there is a need to model events that do not follow a uniform probability distribution.
- As ANS is very fast and widely used, it is interesting to investigate how competitive it is against other cryptographically strong PRBGs.

Contribution. The work investigates an application of ANS-based compression for pseudorandom-bit generation. In particular, we

- examine inherent ANS properties that make it an excellent candidate for pseudorandom bit generation. We show that if symbols fed into ANS follow probabilities that are natural powers of 1/2, then ANS states happen with a uniform probability distribution,
- define a family of ANS-based PRBG and prove that no distinguisher can tell it apart from truly random generators for a big enough parameter R,
- design a family of PRBGs that uses ANS whose symbol spread function is chosen at random. An ANS input symbol frame is randomly shuffled. Both random selection of the symbol spread function and symbol frame shuffling are controlled by a cryptographic key,
- analyse the security of our design and show that any algebraic analysis over a non-binary field requires an adversary to make guesses about binary encodings produced by ANS,
- present a fast PRBG with two ANS instances, whose outputs are interleaved. The gain in efficiency goes together with the increase of security (as an adversary needs to guess the relation between binary encodings and ANS instances),
- implement our ANS-based PRBG to verify its efficiency and security (by running the NIST statistical tests). Table 1 compares ANS-based PRBG with other constructions.

PRBG	Efficiency Mb/s	Security Proof	PQ Secure	NIST Tests
AES-based[1]	261.6 [21]	✗	✓	✓
SHA3-based[2]	1280	✗	✓	✓
ChaCha-based[3]	1336 [21]	✗	✓	✓
BBS [4]	0.134 [19]	✓	✗	✓
ANS-based	733	✓	✓	✓

Table 1. Summary of results (PQ stands for post-quantum), (1) the textbook version of AES without hardware optimisation, (2) based on Keccak-f[1600] with 24 rounds for initialization and 6 rounds per output, and (3) the /dev/urandom implementation from Linux kernel version 4.10.0.

2 Pseudorandomness

Knuth in his work [16] summaries early methods for generation of pseudorandom numbers. A typical solution applies the *linear congruential method*, which generates a sequence of integers x_i according to the congruence

$$x_{i+1} \equiv a \cdot x_i + c \bmod N, \tag{1}$$

where N is an positive integer, $0 \leq a, c \leq N$ and $i \in \mathbb{N}$. The congruence needs the so-called *seed* x_0, which provides a starting point. Note that the sequence of integers is periodic. The choice of the modulus N and the multiplier a forces the length of the period. The maximum length of the period is N. Assuming an adversary can observe integers x_i (for $i = 1, 2, \ldots$), then they are able to recover a, c and the seed x_0 after three observations and alternatively generate all further integers.

A generalisation of this approach is a generation of pseudorandom bits using a linear feedback shift register (LFSR) with a linear filter (see Figure 1).

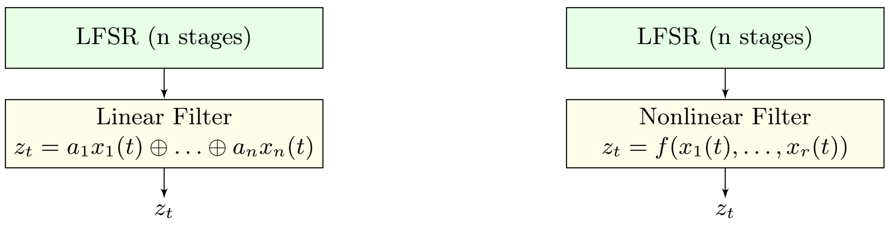
LFSR (n stages)
Linear Filter
$z_t = a_1x_1(t) \oplus \ldots \oplus a_nx_n(t)$
z_t
LFSR (n stages)
Nonlinear Filter
$z_t = f(x_1(t), \ldots, x_r(t))$
z_t

Fig. 1. LFSR-based PRBG with linear and nonlinear filters

Note that the binary coefficients a_i are fixed and determine the filter. The register state is $(x_1(t), \ldots, x_n(t))$ at the clock t. A well designed LFSR with n binary stages (variables) can generate sequence of the length $2^n - 1$, which is the maximum possible length. Mersenne Twister (see *https://en.wikipedia.org/wiki/Mersenne_ Twister*) is an example of a such PRBG. It generates all integers except (all) zero and the sequence passes most of the simple statistical tests. If an adversary has access to the output z_t and knows its structure but does not know the seed, from which it has started, then they can completely determine the seed after n observations of the output. This is due to the fact that each observation provides a linear relation (in n variables).

A more secure version of pseudorandom bit generation is LFSR with a nonlinear filter as presented in Figure 1. The nonlinear filter $f()$ takes r inputs, which typically are selected stages (bits) of LFSR and at each clock t, outputs a single bit z_t. The analysis of the generator becomes significantly more complex. However, an adversary can still determine initial state or seed after observing enough output bits. To see how this is possible, it is enough to consider the nonlinear filter and its algebraic degree. Assume that that $\deg f = m$. This means that the filter can produce outputs that depend on single state variables and all monomials of the state up to degree m $(m \leq r)$. In other words, the filter is able to produce $\sum_{i=1}^{m} \binom{n}{i}$ different monomials/terms. Now the adversary can treat them as independent variables, create a linear relation for each observation and solve the system of relations using the Gaussian elimination (see [8]).

It turns out that the algebraic analysis becomes intractable when the states of the generators are modified using nonlinear functions. The resulting constructions also called nonlinear shift registers (NLSRs) offer excellent statistical properties and are difficult to “break". The eStream project completed in 2008 (see http://www.ecrypt.eu.org/stream/) recommends many good quality stream ciphers based on NLFSRs. Trivium [9] is a good example of an elegant design and a strong resistance against cryptanalysis.

An alternative approach to the above heuristics is to design pseudorandom generators using intractable mathematical problems such as factorization or discrete logarithm. This leads us to computationally secure pseudorandomness, which guarantees that an adversary with polynomial computing resources cannot construct an efficient distinguisher (or a statistical test that the generator fails). Alexi et al. [1] show how to use the RSA exponentiation to generate pseudorandom sequences. Blum et al. [4] study pseudorandom sequences based on squaring assuming that quadratic residueosity is intractable. In general, computationally secure pseudorandom generators allow users to increase the level of security by selecting more difficult instances of the intractable problem. A price to pay, however, is the efficiency of sequence generation. In 1997 Shor [18] has proven that factorization and discrete logarithm are easy on a quantum computer. This makes all cryptographic algorithms based on these (and other related) problems insecure in the post-quantum world. It is worth noting that Grover's algorithm [13] reduces security level of all symmetric key cryptosystems by half. It means that to maintain the same security level for symmetric encryption in the post-quantum world, it is enough to double the length of cryptographic keys.

3 Asymmetric Numeral Systems

ANS uses an integer $x \in \mathbb{N}$ to encode a symbol $s \in \mathbb{S}$ into an integer $x' \in \mathbb{N}$, where s happens with probability p_s. This implies that $\lg_2(x') \approx \lg_2 x + \lg_2 1/p_s$. Alternatively, we can say that we are looking for an encoding function $C(x, s)$ such that $x' = C(x, s) \approx x/p_s$. To illustrate the idea, consider a binary case, when a symbol $s \in \{0, 1\}$ happens with probability p_s. An encoding function $C(x, s) = x' = 2x + s$ and its decoding function $D(x') = (x, s) = (\lfloor x'/2 \rfloor, x' \bmod 2)$. Assuming uniform probability distribution $p_s = 1/2$, a standard binary coding assigns even integers to $s = 0$ or $\mathbb{I}_0 = \{x' | x' = C(x, s = 0); x \in \mathbb{N}\}$. Odd integers carry $s = 1$ or $\mathbb{I}_1 = \{x' | x' = C(x, s = 1); x \in \mathbb{N}\}$ – see Table 2. Consider an example. Let us encode a sequence

$s \setminus x'$	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$s = 0$	0		1		2		3		4		5		$\cdots$
$s = 1$		0		1		2		3		4		5	$\cdots$

Table 2. Encoding table for binary symbols with probabilities $p_s = 1/2$

of symbols 1011. We start from $x = 0$ and follow the process shown below

$$x = 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 5 \xrightarrow{1} 11$$

Decoding process starts from $D(11) = (\lfloor 11/2 \rfloor, 11 \bmod 2) = (5, 1)$; $D(5) = (\lfloor 5/2 \rfloor, 5 \bmod 2) = (2, 1)$; $D(2) = (\lfloor 2/2 \rfloor, 2 \bmod 2) = (1, 0)$; and $D(1) = (\lfloor 1/2 \rfloor, 1 \bmod 2) = (0, 1)$.

The binary case can be extended to the one with an arbitrary probability distribution $p_s \neq 1/2$. We need to modify sets $\mathbb{I}_s$ so the cardinality of the set $\mathbb{I}_s \cap [0, x\rangle$ follows closely $p_s \cdot x$, where $[0, x\rangle$ denotes a set of all integers between 0 and x (including 0). Let us consider an example for two symbols $s \in \{0, 1\}$ that occur with probabilities $p_0 = 1/4$ and $p_1 = 3/4$. Table 3 shows an encoding function $x' = C(x, s)$. Now we encode the same sequence 1011 as follows

$s \setminus x'$	0	1	2	3	4	5	6	7	8	9	$\cdots$
$s = 0$	0				1				2		$\cdots$
$s = 1$		0	1	2		3	4	5		6	$\cdots$

Table 3. Encoding table for binary symbols with probabilities $p_0 = 1/4$ and $p_1 = 3/4$

$x = 0 \xrightarrow{1} 1 \xrightarrow{0} 4 \xrightarrow{1} 6 \xrightarrow{1} 9$. Clearly, this encoding is shorter from the previous one.

The idea can be generalised for an arbitrary set of symbols not necessarily binary. Natural numbers $\mathbb{N}$ are divided into intervals, each containing 2^R integers, where R is an integer parameter big enough so probabilities of symbols can be well approximated. This means that for each symbol, there are $L_s \approx 2^R p_s$ integers/states in an interval and $\sum_s L_s = 2^R$. Given an interval of 2^R integers, whose entries are labelled by consecutive integers from 0 to $2^R - 1$. Then integers assigned to s are put in L_s consecutive locations from $[c_s, c_{s+1}\rangle$, where c_s is the first location, $c_{s+1}-1$ is the last location and $c_s = \sum_{i=0}^{s-1} L_i$. For example consider blocks of 4 columns (intervals) from Table 3. For $s = 1$, each block contains states at locations $[1, 4\rangle$, where $L_0 = 1$ and $L_1 = 3$. We can construct an appropriate encoding table that has $n = |\mathbb{S}|$ rows and enough column so you can process a long enough sequence of symbols. Note that encoding can be performed very efficiently. Given an integer $x \in \mathbb{N}$ and a symbol $s \in \mathbb{S}$, then an encoding function $C(x, s)$ is calculated according to the following steps:

- Identify a block/interval that contains x. The integer $2^R \lfloor x/L_s \rfloor$ points to first x' of the block.
- Compute an offset (within L_s block locations), which is $(x \bmod L_s)$.
- Find c_s, which gives the location of the first state associated with s in the block.
- Determine $C(x, s) = 2^R \lfloor x/L_s \rfloor + (x \bmod L_s) + c_s$

The corresponding decoding function $D(x') = (x, s)$ recovers an integer x and a symbol s carried by x' and $D(x') = \big(L_s \lfloor x'/2^R \rfloor + x' \bmod 2^R - c_s, s\big)$, where s is identified by checking if $c_s \le x \bmod 2^R \le c_{s+1}$.

This means that compression operations can be defined for an appropriate interval/block of the length 2^R. Observe that while encoding, the final integer/state x' increases very fast as $\lg_2(x') \approx \ell H(\mathbb{S})$, where $H(\mathbb{S})$ is a symbol entropy. Handling very large integers becomes a major efficiency bottleneck. ANS deals with this problem using the so-called *re-normalisation.* Re-normalisation keeps a state x within an interval of a fixed length, for instance $x \in \mathbb{I} = [2^{k_s}, 2^{k_s+1}\rangle$. If ANS is in a state x and gets a sequence of symbols so $x' \ge 2^{k_s+1}$, then it outputs bits $x' (\bmod\ 2^{k_s})$ (as partially compressed bits) and reduces the state $x \longleftarrow \lfloor x'/2^{k_s} \rfloor$. The re-normalisation operation is reversible, i.e. knowing a pair $(x' (\bmod\ 2^{16}), \lfloor x'/2^{16} \rfloor)$, it is easy to reconstruct x'. In practice, ANS applies $\mathbb{I} = [2048, 4096\rangle$ for 256 symbols. Using re-normalisation allows ANS to achieve efficient compression and also it can be conveniently represented as an encoding table. Description of ANS algorithms together with a simple example is given in Appendix.

Duda in his work [11] suggests using ANS as a source of cryptographically strong pseudorandom bits. We develop the idea further. The main test that any CSPRBG has to pass is the test of indistinguishability. PRBG passes the test if no adversary is able to distinguish PRBG from a truly random source. Note that by its nature ANS achieves a close to optimal compression. A residual redundancy can be used to launch a distinguishing attack against an ANS-based PRBG. Let us take a closer look at a relation between entropy and probability for a binary case. Assume that bits are slightly biased, i.e. $P(0) = 1/2 + \varepsilon$ and $P(1) = 1/2 - \varepsilon$. Strightforward calculations show that

$$\Delta H \approx \frac{2}{\ln 2} \varepsilon^2$$

where ΔH is the entropy difference between the uniform and biased distributions and we use the Taylor series approximation of logarithm. Note, for instance, that if $\Delta H = 2^{-20}$, then $\varepsilon \approx 2^{-11}$. The bad news is that entropy differences are translated into much bigger probability biases. The good news, however, is that it is unlikely that the redundancy ΔH converts into a bit bias. More likely, the output bits includes sort of "parity-check" bits. Nevertheless, to be on the safe side, it is reasonable to assume that any redundancy creates the maximum bias $\varepsilon \approx \sqrt{\Delta H}/1.7$.

The well-known fact is that to identify a bias ε, an adversary needs to collect $\approx \varepsilon^{-2}$ bits/observations. For example, for any $\varepsilon \approx 2^{-30}$, it is necessary to observe around 2^{60} bits but this corresponds to a small $\Delta H \approx 2^{-59}$. Assume that ANS uses a random process, which serves as a symbol source and whose statistics is well defined. To design an ANS-based PRBG, we can

1. choose a big enough number of states so the redundancy of binary output is smaller than $\Delta H = 2^{-64}$ as distinguishing attack does not work; or alternatively
2. optimise ANS so it is close to the optimal solution (or the entropy difference between the optimal and design ones is smaller than $\Delta H = 2^{-64}$, for instance).

3.1 ANS with High Compression Rates

Ideally, our problem is solved if we are able to design ANS that is optimal, i.e. compression rate equals to the entropy of a symbol source. Let us investigate this aspect. We start from the following theorem.

Theorem 1. **[7]** *Given a plain ANS as described in Appendix. Then, for a symbol $s \in \mathbb{S}$ (which occurs with probability p_s), ANS generates binary encodings whose length is*

- *k_s bits if $p_s = 2^{-k_s}$ and k_s is a positive integer. Binary encoding entries of encoding table contain k_s-bit sequences for all states $x \in \{2^R, \ldots, 2^{R+1} - 1\}$,*
- *either k_s or $(k_s + 1)$ bits if $2^{-k_s} > p_s > 2^{-(k_s+1)}$. Encodings are k_s-bit long for all states $x \in \{2^R, \ldots, 2^{k_s+1}L_s - 1\}$ while the remaining encodings are (k_s+1)-bit long for $x \in \{2^{k_s+1}L_s, \ldots, 2^{R+1} - 1\}$, where $L_s = 2^R p_s$ and k_s is a positive integer.*

Corollary 1. *Theorem 1 leads to the following conclusions.*

- *Given a source whose symbols occur with probabilities that are natural powers of $1/2$ and ANS with a big enough parameter R so L_s is an integer for any $s \in \mathbb{S}$. Then the average length of binary encodings is equal to the symbol source entropy $H(\mathbb{S})$. In other words, ANS is optimal.*
- *ANS is optimal if and only if for each symbol $s \in \mathbb{S}$, the average encoding length equals to $\log_2 p_s$, where $2^{-k_s} > p_s > 2^{-(k_s+1)}$. In other words, the following relation is true*

$$\sum_{i=2^{k_s+1}L_s}^{2^{R+1}-1} P(x = i) = \log_2 p_s^{-1} - \lfloor \log_2 p_s^{-1} \rfloor = \log_2 p_s^{-1} - k_s. \qquad (2)$$

  To make any statement about ANS optimality, we need to know a probability distribution of ANS states or $\{P(x)|x \in \mathbb{I}\}$.
- *There is an interesting case when $k_s = 0$ or $2^0 > p_s > 2^{-1}$. Symbol encodings for $x \in \{2^R, \ldots, 2L_s - 1\}$ are empty bits $\varnothing$. Encodings for $x \in \{2L_s, \ldots, 2^{R+1} - 1\}$ are single bits.*

Below we reflect on the above fact and its relation to the well-known Huffman code (HC) [15].

Corollary 2. *Both ANS and HC produce optimal entropy encoding for symbols whose probabilities are natural powers of $1/2$ but*

- *HC encoding produces a prefix code. For each symbol, it assigns a unique binary encoding. Note that encodings for a prefix code are never a prefix of any other encoding. This guarantees unique decoding.*
- *ANS assigns an encoding to a symbol that depends on its current state. So it produces different encodings for the same symbol. The decoding recovers the correct symbol frame if the process starts from the correct ANS state.*

Example 1. Given a symbol source $\mathbb{S} = \{s_1, s_2, s_3\}$, where $p_{s_1} = 1/2$ and $p_{s_2} = p_{s_3} = 1/4$. HC for the source is as follows: $s_1 \to 1$, $s_2 \to 01$ and $s_3 \to 00$. For a parameter $R = 2$, ANS is described by the following table.

$p_{s_i}\backslash x_i$	4	5	6	7
$p_{s_1}=1/2$	$\binom{4}{0}$	$\binom{4}{1}$	$\binom{6}{0}$	$\binom{6}{1}$
$p_{s_2}=1/4$	$\binom{5}{00}$	$\binom{5}{01}$	$\binom{5}{10}$	$\binom{5}{11}$
$p_{s_3}=1/4$	$\binom{7}{00}$	$\binom{7}{01}$	$\binom{7}{10}$	$\binom{7}{11}$

Table 4. ANS instance for $R = 2$ and three symbols, where $\binom{x_{i+1}}{b_i}$ means that the next state is x_{i+1} and the encoding is b_i

3.2 Uniform Probability Distribution of ANS States

It turns out that for symbols with probability distribution that follows natural powers of 1/2, probability distribution of ANS states is uniform. Let us start our discussion from few observations about ANS. Consider Algorithm 4. Given an ANS state $x \in \mathbb{I}$ and a symbol $s \in \mathbb{S}$, then x transits to a new state $x' = C(s, \lfloor x/2^k \rfloor)$ with probability p_s. The number of distinct transitions equals to the number n of symbols in $\mathbb{S}$ or $n = |\mathbb{S}|$. Take into account Example B. The encoding table of the ANS shows all possible transitions. For a fixed state, there are 5 possibilities each occurring with the symbol probability p_s. For instance, the state 16 goes to either $17, 20, 16, 21$ or 26, with probabilities $1/2, 1/4, 1/8, 1/16, 1/16$, respectively.

Consider Algorithm 5 for ANS decoding. The ANS construction forces that each state is assigned a unique symbol (via its symbols spread function). During decompression, a state $x \in \mathbb{I}$ is first converted into a pair $D(x) = (s, y)$ and then traverses to a new state $x' = 2^k y + b$, where b is an binary string of the length k. Consequently, there are 2^k possibilities for x'. For instance, the state 20 from Example B can transit to $x' = 16 + b$, where $b \in \{0, 1, 2, 3\}$ or to one of the states from $\{16, 17, 18, 19\}$ with probability 1/4. The above analysis brings us to the following lemma.

Lemma 1. *Given ANS as described by Algorithms 3, 4 and 5. Then an ANS state $x \in \mathbb{S}$ transits to a state*

- $x' \in \{C(s, \lfloor x/2^k \rfloor); s \in \mathbb{S}\}$ *during compression, where probability* $P(x'|x) = p_s$,
- $x' \in \{2^k y + b; b = 0, 1, \ldots, 2^k - 1\}$ *during decompression, where b is an binary string of the length k and all states x' are equally probable or* $P(x'|x) = 1/2^k$.

A close look at the lemma leads us the following remarks.

- The evolution of states during a single compression/decompression step can be presented as a graph, whose nodes are states and edges are labelled by probabilities of state transitions.
- A graph representation of ANS state transitions is generic, i.e. its probabilistic properties do not depend on a particular symbol spread function deployed by ANS.
- The ANS probabilistic characteristics hold for any source whose symbols occur with probabilities that are natural powers of 1/2.

To illustrate our arguments, consider the ANS from Table 4. Figure 2 depicts its state transition probabilities. The left-hand side of the graph consists of input states whose transition probabilities are equal to $(1/2, 1/4, 1/4)$. Each node has three outbound edges. The output states accept 2^k incoming edges, where the parameter $k = 1, 2$ is determined by the ANS symbol spread function.

We explore Markov chains in order to investigate the asymptotic behaviour of ANS states [14]. In general, ANS state probability distribution is not uniform and can be approximated by $P(x) \approx 1/x$, where $x \in \mathbb{I}$. This is not the case when symbol probabilities are natural powers of 1/2. Let us consider ANS from Table 4. The ANS probability transition matrix is

$$P = \begin{bmatrix} p_{4,4} & p_{4,5} & p_{4,6} & p_{4,7} \\ p_{5,4} & p_{5,5} & p_{5,6} & p_{5,7} \\ p_{6,4} & p_{6,5} & p_{6,6} & p_{6,7} \\ p_{7,4} & p_{7,5} & p_{7,6} & p_{7,7} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{4} & 0 & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}$$

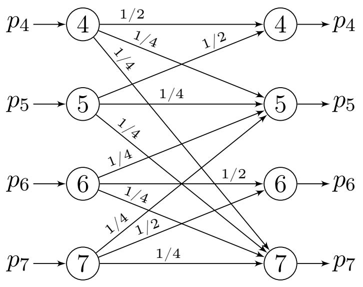

Fig. 2. Transition probabilities for ANS with $R = 2$

where $p_{i,j}$ is probability that a state i moves to a state j; $i, j \in \{4, 5, 6, 7\}$ (see Figure 2). It is easy to verify that a Markov chain created by concatenation of the matrix P converges to $\lim_{n\to\infty} P^n = [1/4]$, where $[1/4]$ is a matrix with all entries equal to 1/4. As determined by symbol probabilities, there are three possible output states for each input state. However, looking at output states, one can quickly identify that states 4 and 6 can be reached from $\{4, 5\}$ and $\{6, 7\}$, respectively with probability 1/2. The output states 5 and 7 can be reached from all input states with probability 1/4. In the equilibrium state, input and output states must occur with the same probabilities – see Figure 2. This leads us to the following four relations

$$\begin{aligned}
p_4 &= \frac{1}{2}(p_4 + p_5) \\
p_5 &= \frac{1}{4}(p_4 + p_5 + p_6 + p_7) = \frac{1}{4} \\
p_6 &= \frac{1}{2}(p_6 + p_7) \\
p_7 &= \frac{1}{4}(p_4 + p_5 + p_6 + p_7) = \frac{1}{4}
\end{aligned}$$

The solution is $p_4 = p_5 = p_6 = p_7 = 1/4$.

The following theorem summarises the above deliberations.

Theorem 2. *Given ANS for the parameter R and the symbol probabilities that are natural powers of $1/2$. Then probabilities of ANS states converge asymptotically to the uniform distribution or $P(x) = (1/2)^R$ for $x \in \mathbb{I}$.*

Proof. Denote asymptotic probabilities by p_x, where $x \in \mathbb{I}$ (as shown in Figure 2). In equilibrium, the probabilities are the same for both input and output states. Consider an output state $x' \in \mathbb{I}$. According to Lemma 1, the state x' can be reached from 2^k input states $x \in \{2^k y + b; b = 0, 1, \ldots, 2^k - 1\}$, where x' is assigned (by a symbol spread function) to a symbol with probability 2^{-k}. This immediately produces the following relation:

$$p_{x'} = \frac{1}{2^k} \sum_{x\in\{2^k y+b;b=0,1,\ldots,2^k-1\}} p_x$$

In total, we get 2^R such relations - one per output state. Each relation requires that probability $p_{x'}$ has to be equal to the average of probabilities p_x, where $x \in \{2^k y + b; b = 0, 1, \ldots, 2^k - 1\}$. Moreover, by construction, each input state appears in precisely one relation for a given symbol. This is a remarkable property of ANS states as they exhibit the same probabilistic behaviour. This implies that in the equilibrium state, all probabilities have to be the same. This concludes our proof.

Example 2. Consider ANS from Table 4 and its transition probabilities given in Figure 2. Assume input state probabilities $p_4 = \frac{1}{4} + \varepsilon$, $p_5 = \frac{1}{4}$, $p_6 = \frac{1}{4} - \varepsilon$ and $p_7 = \frac{1}{4}$. Then each compression step reduces ε by half or

$$\begin{aligned} p_4 = \frac{1}{4} + \varepsilon &\longrightarrow \frac{1}{4} + \frac{\varepsilon}{2} \\ p_5 = \frac{1}{4} &\longrightarrow \frac{1}{4} \\ p_6 = \frac{1}{4} - \varepsilon &\longrightarrow \frac{1}{4} - \frac{\varepsilon}{2} \\ p_7 = \frac{1}{4} &\longrightarrow \frac{1}{4} \end{aligned}$$

Clearly, all probabilities asymptotically converge to 1/4.

4 Design Methodology of PRBGs with ANS

Our PRBG design is guided by the following principles. It should

- be as fast as an original ANS;
- produce a non-repeating sequence of bits (i.e. a sequence cycle has to be long enough so it is impossible to repeat it, say 2^{128});
- generate uncorrelated and balanced sequence of bits;
- be indistinguishable from a truly random bit source;
- resist against any adversary, whose computing resources are limited to $\Theta(2^{128})$. It means that our target is 128-bit security, i.e. we apply a 128-bit cryptographic key. Note that to maintain the 128-bit security level for a quantum adversary, we need 256-bit keys;
- be scalable, i.e. easily adjustable to a required level of security.

Our PRBG design uses the following basic blocks:

- ANS – we choose a generic ANS designed for a source with symbols occurring with probabilities that are natural powers of 1/2. In fact, we deal with a family of ANS_{k_s}. Each ANS_{k_s} is an instance designed for a source $\mathbb{S}_{k_s} = \{s_1, {}_2, \ldots, s_{k_s}, s_{k_s+1}\}$, where a symbol s_i occurs with probability $p_{s_i} = 2^{-i}$ for all $i \leq k_s$ and $p_{s_{k_s+1}} = p_{s_{k_s}}$. Note the following important properties of the family:
  1. it is possible to define ANS_{k_s} for arbitrarily large parameter k_s that directly determines the size of the encoding table (parameter R),
  2. each ANS_{k_s} is optimal, i.e. it squeezes out all redundancy and generates a sequence of uncorrelated and unbiased bits,
  3. it is easy to calculate the bit generation rate.
- SF – this is a file or array, where a symbol frame (that is going to be compressed by ANS_{k_s}) is stored. Symbols in the frame must follow the probability distribution determined by the parameter k_s. The size of the symbol frame should not be too large but at the same time, we would like to generate an arbitrarily long sequence of PRBG bits. A simple solution is to reuse the frame as many times as necessary. To avoid a cycle, we need to permute symbols of the frame every time it is reused.
- Permutation $\pi \in Sym(\ell)$ – it is necessary to reshuffle ℓ frame symbols, where $Sym(\ell)$ is a symmetric group defined for ℓ elements. It is easy to determine a set of generators g_i of $Sym(\ell)$ with a help of Magma [6], for instance. Clearly, we need to choose π randomly using a relatively short secret key K.

4.1 Generic PRBG Designs

Figure 3 illustrates our generic PRBG design. A symbol frame (SF) contains symbols that occur with an assumed probability distribution. Symbols of SF are permuted according to π, which is controlled by a secret key K. Symbols are processed by a keyed ANS, which produces a sequence of pseudorandom bits. The ANS symbol spread function is indexed by the secret key K. The

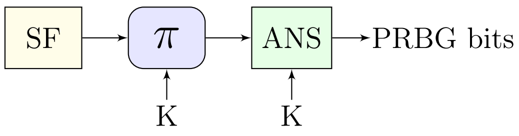

Fig. 3. Generic PRBG with keyed ANS

function can be implemented by first permuting 2^R states according to the secret key K and then by splitting the permuted states into subsets $\mathbb{L}_i$ whose cardinalities L_i reflect probabilities of symbols s_i. Clearly, for a given symbol s_i, the states from $\mathbb{L}_i$ must be arranged in the increasing order.

4.2 Provable Security

A concept of provable security relates to a family of PRBG indexed by a parameter that allows a user to choose different instances with different security levels. RSA and BBS pseudorandom generators (see [1, 4]) are good examples of such families. Their index is the length of the modulus used for computations. To guarantee a required level of security λ, it is enough to choose a long enough modulus N so the corresponding factorisation (for RSA) or quadratic residuosity (for BBS) instance requires 2^λ computation steps.

Definition 1. *ANS-based PRBG is a family of ANS algorithms indexed by their parameter R or $\mathcal{F}_{ANS} = \{ANS_R | R \in \mathbb{N}\}$. Each ANS_R instance is designed for 2^R states, where its symbol spread function is chosen at random. It is fed by a symbol source $\mathbb{S}$, where each symbol is chosen independently, randomly and whose probability is a natural power of* $1/2$.

Theorem 3. *(Provable Security) Given*

(A1) a symbol source $\mathbb{S}$, where each symbol is chosen independently, randomly and whose probabilities are natural powers of $1/2$ *and*
(A2) an instance of ANS_R designed for $\mathbb{S}$ and whose states occur uniformly at random.

Then ANS generates binary encodings

(P1) whose average length is equal to the symbol source entropy $H(\mathbb{S})$*,*
(P2) whose bits are independent and uniformly random,
(P3) that pass all statistical tests or alternatively, no distinguisher is able to tell apart an ANS bit stream from a random one.

Proof. P1 is a direct conclusion from Theorem 1 and also Corollary 1. This statement is equivalent to the claim that ANS is optimal. *P2* follows from the assumption that ANS chooses its states with uniform distribution and the fact that every row of ANS encoding table contains 2^{R-k_s} copies of all possible binary strings of the length k_s - see Theorem 1. *P3* is a conclusion from the statements *P1* and *P2*.

The above theorem is true under the assumption that ANS_R states occur uniformly at random. Theorem 2 asserts that probabilities of ANS_R states converge to a uniform distribution. This leads us to the following conclusion.

Corollary 3. *Sequences generated by the family $\mathcal{F}_{ANS} = \{ANS_R | R \in \mathbb{N}\}$ are indistinguishable from truly random ones for all big enough R.*

The above considerations tell us that the family $\mathcal{F}_{ANS}$ has a "sound structure". The optimistic conclusion needs to be confronted by the fact that in practice, we are interested in the implementation of an instance ANS_R (rather than the whole family $\mathcal{F}_{ANS}$). For our ANS_R, we would like to choose a relatively small R so our PRBG is efficient. Most importantly, the assumptions *A1* and *A2* of Theorem 3 have to hold. Consider the assumption *A1*. Instead of symbols generated by the source $\mathbb{S}$, we take a symbol frame of a fixed length whose statistics follows the required one and whose content is shuffled (periodically) by a cryptographic key. The assumption *A2* requires a uniform state distribution. Theorem 2 assures us that ANS state probability distribution asymptotically converges to a uniform Markov-chain equilibrium. It is reasonable to expect that asymptotic uniform probability distribution does not guarantee "local" uniformity especially for a small R. ANS state probability distribution fluctuations are quite obvious due to a cyclic nature of ANS (see [7] for details). Needless to say, the state probability fluctuations produce a correlation amongst local binary encodings. In other words, it is possible to design a distinguisher that targets the correlation (see Section 4.3). There are a few options to eliminate or reduce the output bit correlation.

- Construction of PRBG with a few parallel ANS (threads) whose outputs are merged by interleaving their individual encodings.
- XOR-ing PRBG output bits with a pseudorandom sequence derived from a cryptographic key K. For instance, the sequence can be a rotated K, where the rotation offset is calculated from a current ANS state.
- ANS state probability distribution can be made locally uniform (and consequently force uniformity of binary encodings) by XOR-ing a pseudorandom sequence with the ANS full/partial state.
- A well-through-out design of ANS with long cycles of states. Note that while the parameter R grows, the probability of short cycles drops in a natural way. This means that for a big R, local correlations occur with a negligible probability [5]. This inevitably leads us to the theory of random graphs, which is beyond the scope of the paper. This interesting aspect of ANS is left for a future research.

It is worth noting that security of ANS-based PRBGs does not rely on any intractability assumption (such as integer factoring). In contrast, it depends on randomness used to construct the symbol spread function of ANS. We claim that a quantum adversary can obviously apply the Grover algorithm [13] and reduce the randomness length by half. This means that instead of the exhaustive search of the whole space 2^{λ}, the adversary is able to reduce search to $2^{\lambda/2}$, where λ is the security level.

4.3 Security Analysis

We analyse an ANS-based generator, whose symbol spread function is chosen at random (controlled by a secret key K, see Figure 3). Additionally, we make the following assumptions:

- The source generates symbols whose probabilities are natural powers of 1/2, i.e. $p_{s_i} = (1/2)^i$ for $i = 1, 2, \ldots$ and $\sum_i p_{s_i} = 1$.
- ANS is designed for a parameter R, i.e. the number of states is 2^R.

- SF contains symbols that follow the assumed probability distribution. The symbols are shuffled well enough so we consider the permutation π_K to be truly random.

Without loss of generality, we describe the encoding table by the following two functions (see Appendix):

$$C_s(x) = C(s, \lfloor x/2^{k_s} \rfloor) = c_{s,0} + c_{s,1}x + \ldots + c_{s,\alpha_s}x^{\alpha_s};$$
$$B_s(x) = x \mod 2^{k_s} = d_{s,0} + d_{s,1}x + \ldots + d_{s,\beta_s}x^{\beta_s};$$

where the functions are presented over $GF(2^R)$. Our goal is to design a distinguisher, which is able to tell apart our PRBG from a truly random one. As we are not able to observe symbols, the second best we can do is to guess a long enough sequence of the most probable symbol s_1 that occurs with probability 1/2. For this case, we know that

$$C(x) = c_0 + c_1x + \ldots + c_{k_s}x^{\alpha};$$
$$B(x) = x \mod 2 = \sum_{i=0}^{R-1} d_i x^{2^i}$$

where $k_s = 2^{R-1}-1$ and as $B(x)$ is linear, it can be represented by a combination of linear function over $GF(2^R)$. For instance, $B(x) = x \pmod 2 = x + x^2 + x^4$ in $GF(2^3)$. Now, suppose that we observe n output bits and know the initial state x_0. So we can write a system of (nonlinear) equations as follows:

$$x_i = C(x_{i-1})$$
$$b_i = B(x_{i-1}) \text{ for } i = 1, \ldots, n$$

As we have $2^{R-1} + n$ unknowns, we are not able to solve the system (even if it were linear). Note that the output bits b_i give a single bit of information about the states. Moreover, the system of equation is true with the probability $(1/2)^n$. This approach is doomed to fail for a large enough n.

Note that for a sequence of the symbol s_1, $C(x)$ has to be cyclic. The longest possible length is the number of all states 2^R. In general, there can be few disjoint cycles/paths. Clearly, for loops, $x = C(x)$ and $b = B(x)$. This means that ANS produces a sequence of the same bits. Let us explore an impact of a state loop on the output bit statistics. We use the following simple test.

Loop Test

```
Input: A stream of bits $\mathbf{b} = (b_1, \ldots, b_n)$.
Output: An integer $LT$ that indicates existence of loops.
Steps: Initialize $LT = 0$;
    for $i = 1$ to $n - 1$
    {
    if $b_i = b_{i+1}$ then $LT++$;
    }
    return($LT$);
```

Behaviour of the test for a truly random sequence $\mathbf{b}$ is described by the following lemma.

Lemma 2. *Given a truly random binary sequence $\mathbf{b} = (b_1, \ldots, b_n)$. Then the loop test returns a random variable LT_n with the probability distribution described by the following recursion:*

$$P(LT_n = i) = \begin{cases} (\frac{1}{2})^{n-1} & \text{if } i \in \{0, n-1\}; \\ \frac{1}{2}\left(P(LT_{n-1} = i-1) + P(LT_{n-1} = i)\right) & \text{if } i \in \{1, \ldots, n-2\}, \\ P(LT_2 = 0) = P(LT_2 = 1) = \frac{1}{2} & \text{Stopping Case} \end{cases}$$

where LT_n is a random variable for n-bit strings.

Proof. It is easy to check that for 2-bit random sequences, the test returns $LT_2 = 0$ for sequences 01 and 10. $LT_2 = 1$ for sequences 00 and 11. The two events are equally probable. The recursive relation can be proven by observing that $LT_n = i$ when either

- $LT_{n-1} = i - 1$ and the n-th bit matches the $(n-1)$-th bit (so the test executes $LT++$) or
- $LT_{n-1} = i$ and the n-th bit does not match the $(n-1)$-th bit (the count LT stays the same).

The two above events are exclusive and this concludes the proof.

As the number n of bits grows, the average of LT_n hovers around $(n-1)/2$ as illustrated by the table given below.

$i \rightarrow$	0	1	2	3	4	5
$P(LT_2 = i)$	$\frac{1}{2} \cdot 1$	$\frac{1}{2} \cdot 1$				
$P(LT_3 = i)$	$\frac{1}{4} \cdot 1$	$\frac{1}{4} \cdot 2$	$\frac{1}{4} \cdot 1$			
$P(LT_4 = i)$	$\frac{1}{8} \cdot 1$	$\frac{1}{8} \cdot 3$	$\frac{1}{8} \cdot 3$	$\frac{1}{8} \cdot 1$		
$P(LT_5 = i)$	$\frac{1}{16} \cdot 1$	$\frac{1}{16} \cdot 4$	$\frac{1}{16} \cdot 6$	$\frac{1}{16} \cdot 4$	$\frac{1}{16} \cdot 1$	
$P(LT_6 = i)$	$\frac{1}{32} \cdot 1$	$\frac{1}{32} \cdot 5$	$\frac{1}{32} \cdot 10$	$\frac{1}{32} \cdot 10$	$\frac{1}{32} \cdot 5$	$\frac{1}{32} \cdot 1$

As the integers in the above table are binomial coefficients (part of Pascal's triangle), the probability distribution of LT_n can be represented in the following compact form

$$P(LT_n = i) = \frac{1}{2^{n-1}} \binom{n-1}{i}, \quad i = 0, 1, \ldots, n-1.$$

Let us investigate the impact of loops on the result of the test. Assume that one state x loops when ANS is fed by s_1, where $p_{s_1} = 1/2$. The state loops i times with probability $P(\#loop = i) = (1/2)^i$ $(i = 1, 2, \ldots)$. On the average, the state adds ≈ 2 to the loop count LT. If ANS contains many loops, then the loop test is likely to produce a much higher value for LT than the expected $(n-1)/2$. This is to say that our loop test may produce an effective distinguisher.

If we treat ANS as a finite state machine (FSM) and draw a graph to illustrate state transitions, then we could see the full complexity of the cyclic structure of ANS. Apart from loops, one can see cycles of order 2 and higher. The cycles are produced by the same patterns of symbols and output binary sequences of a characteristic pattern. Note that for any given cycle, one can design a test that targets it. The good news is that longer cycles (that involve symbols that occur with small probabilities) have negligible impact on output statistics. Tests that target such cycles are likely to fail. The bad news, however, is that an adversary has a large collection of possibilities/tests to explore the cyclic nature of ANS.

The above considerations lead us to the following conclusion. To obscure the cyclic nature of ANS, we need to apply a keyed permutation for the output bits. Its primary goal is to spread bits that may be part of a loop/cycle around a larger block of bits. For instance, bits produced by a loop will be separated by other output bits making the loop count similar to a truly random one.

4.4 Building Blocks

At the initialisation stage, we need to construct one or more ANS instances. Below we give an algorithm that permutes $2^R = 128$ integers (ANS states) from the set $\{0, \ldots, 127\}$ according to the key K. The permutation is a variant of the key scheduling algorithm of RC4 [22]. As we permute 7-bit elements, we need to split K into 7-bit slices. We get $K[i]$; $i = 0, \ldots, 18$, where

Algorithm 1: Permutes 128 states and produces ANS symbol spread function

```
Input: An array $X[i]$ $(i = 0, \ldots, 127)$ and $K[i]$ $(i = 0, \ldots, 18)$ .
Output: ANS symbol spread function for eight symbols $s_i$ occurring with probabilities
        $\{1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/128\}$, where $i = 0, \ldots, 7$.
Steps:
for $i := 0$ to 127 do                                                    initialisation
    $X[i] := i$
$j := 0$;
for $i := 0$ to 127 do                                                    mixing entries of $X$
    $j := (j + X[i] + K[i \bmod 19]) \bmod 128$;
    swap $X[i]$ and $X[j]$;
$\mathbb{L}_0 = \{X[0], \ldots, X[63]\}$; $\mathbb{L}_1 = \{X[64], \ldots, X[95]\}$;    symbol spread function
$\mathbb{L}_2 = \{X[96], \ldots, X[111]\}$; $\mathbb{L}_3 = \{X[112], \ldots, X[119]\}$;
$\mathbb{L}_4 = \{X[120], \ldots, X[123]\}$; $\mathbb{L}_5 = \{X[124], X[125]\}$, $\mathbb{L}_6 = \{X[126]\}$; $\mathbb{L}_7 = \{X[127]\}$;
```

the last 7-bit $K[18]$ consists of the two last and five first bits of K. Note that states in $\mathbb{L}_i$ $(i = 1, \ldots, 7)$, are used by ANS in increasing order. If we need more ANS instances, we use Algorithm 1 many times, where the initial permutation for the current run of the algorithm is taken from the previous one.

We also need to pre-compute one or more symbol frames. Algorithm 2 gives details.

Algorithm 2: Produces a symbol frame permutation SF

```
Input: An array $S[i]$ $(i = 0, \ldots, 2^{14} - 1)$ and 128-bit cryptographic key $K[i]$ $(i = 0, \ldots, 18)$ .
Output: Permutation $S[i]$, where $i = 0, \ldots, 2^{14} - 1$.
Steps:
for $i := 0$ to $2^{14} - 1$ do                                           initialisation
    $S[i] := i$
$j := 0$;
for $i := 0$ to $2^{14} - 1$ do                                           mixing entries of $S$
    $j := (j + S[i] + K[i \bmod 19]) \bmod 2^{14} - 1$;
    swap $S[i]$ and $S[j]$;
Return $SF := S$;                                                         frame permutation
```

A symbol frame SF is constructed in two steps. Given an array $S[i]$ of 7-bit strings, where $i = 0, \ldots, 2^{14} - 1$. In the first step, the entries of $S[i]$ are permuted using the key K in a similar way as in Algorithm 1. The second step converts 7-bit entries of $S[i]$ into symbols s_i that follow the required probability distribution $\{1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/128\}$. The frame permutation is now converted to symbol frames or $S \to SF$. Each entry of S is a 7-bit string while each entry of SF needs to be a symbol s_i; $i = 0, \ldots 7$. We use the following coding.

- If $S[i] = 0 \pmod 2$, then $SF[i] = 0 \equiv s_0$. A half of entries contains zeros so $P(s_0) = 1/2$.
- If $S[i] = 1 \pmod 4$, then $SF[i] = 1 \equiv s_1$. A quarter of entries contains one so $P(s_1) = 1/4$.
- If $S[i] = 2^{\ell-1} - 1 \pmod{2^\ell}$, then $SF[i] = \ell \equiv s_{\ell-1}$, where $\ell = 3, 4, 5, 6$. The symbol $s_{\ell-1}$ occurs with probability $\frac{1}{2^\ell}$.
- If $S[i] = 63$ or 127, then $SF[i] = s_7$ and s_8, respectively. The symbols happen with probability $\frac{1}{128}$.

5 Instantiations of PRBG

Our main construction is given in Figure 4. ANS is fed by symbol frames SF_i that are produced by permuting an initial frame SF with an assumed symbol statistics, i.e. $SF_i = \pi_K^i(SF)$, where

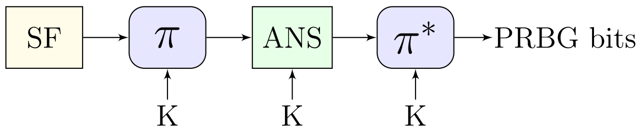

Fig. 4. PRBG with plain ANS and bit stream permutation π^*_K

$i = 1, 2, \ldots$ and K is a cryptographic key. ANS symbol spread function is controlled by the key K. A permutation π^*_K spreads around bits of possible loops and short cycles.

5.1 Threaded ANS-based PRBG

This version of PRBG is designed for speed. The generic construction from Figure 4 is modified so the operations can be executed as fast as possible. Figure 5 illustrates our design. The design

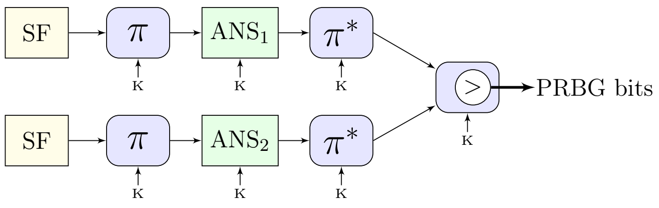

Fig. 5. Two threaded ANS based PRBG

speeds up output bit generation by processing multiple frames in parallel. Thread outputs are concatenated and they form a final PRBG bitstream. In principle, concatenation can be controlled by the cryptographic key K. Consequently, the total PRBG throughput is determined by the sum of single thread contributions. Clearly, a speedup multiplication factor is limited by the number of physically available threads.

5.2 Multi-Frame Threaded ANS-based PRBG

The first version of a threaded ANS-based PRBG applies multiple frames that are fed to the ANS algorithm. It is illustrated in Figure 6. Instead of permuting a symbol frame SF for each frame

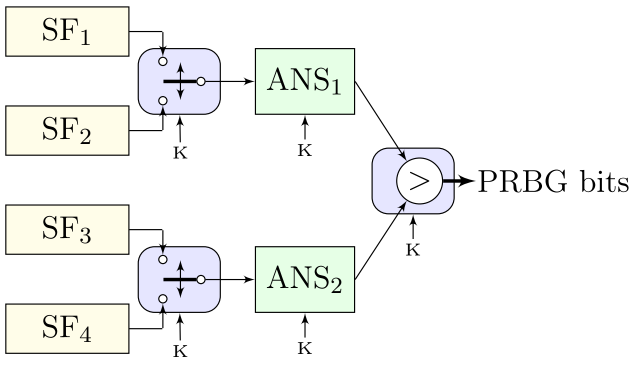

Fig. 6. Dual-frame threaded ANS-based PRBG

iteration, we use four different permuted frames SF_i; $i = 1, 2, 3, 4$, that are fixed for a generation session. They can be pre-computed well before the session. Symbol frames are paired. The first pair (SF_1,SF_2) feeds ANS_1 and the second (SF_3,SF_4) – ANS_2. For each symbol frame pair, there is a switch that selects a currently active frame, from which symbols are drawn. The switches are controlled by a cryptographic key K. Note that once a frame becomes active, all its symbols are processed. After finishing with the current frame, the switch selects a new active one. The heart of the design is two instances of ANS, whose symbol spread functions are controlled by K. The permutation π^* from Figure 4 is replaced by a block that interleaves binary encodings generated by ANS algorithms. To make PRBG fast, we use ANS algorithms for a relatively small number of states. We assume the parameter $R = 7$, which produces a 128-state ANS.

Initialisation follows the steps described below.

1. Construct an instance of 128-state ANS with its symbol spread function chosen at pseudorandom by a 128-bit key K. Note that the symbol probabilities are $\{1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/128\}$. We can adapt the key-scheduling algorithm of the well-known Rivest RC4 cipher to select the symbol spread functions of the two ANS algorithms.
2. Prepare two pairs of symbol frames SF_i; $i = 1, 2, 3, 4$. Their lengths should be different but they have to be multiples of 128. Each frame must contain the number of symbols that conforms with the symbol statistics. The frames are loaded with symbols according to the order of their probabilities, then their contents are shuffled using the key K. This can be done in a similar way to the RC4 cipher, for instance. The shuffled frames SF_i are stored and their contents are fixed. This is done at the very beginning and once per session only.
3. Design a simple key scheduling, which takes a session key K and rotates it by κ positions (it does not matter left or right as long as rotation is consistent). It is denoted by $K := K_{Rot\ \kappa}$. Notation $K[i]$ means the i-th bit of K; $i = 0, \ldots, 127$.

PRBG generation proceeds as follows (see Figure 6). The two algorithms ANS_1 and ANS_2 are run as parallel threads, whose binary outputs (encodings) are interleaved (merged) into a single outputs binary stream. Below we describe the upper thread for ANS_1. The lower thread works in a similar fashion and its description is skipped.

1. Choose a frame SF_i $(i = 1, 2)$ using the 128-bit key K. If this is the first frame, we compute $K[0] \oplus X[6]$, where $X[6]$ is the second most significant bit of an initial ANS_1 state X. Note that $X[7] = 1$. If $K[0] \oplus X[6] = 1$, select SF_1, otherwise take SF_2. If this is the n-th frame, we calculate $K[n \pmod{128}] \oplus X[6]$, where X is a current ANS_1 state. If the result is 1 then we choose SF_1, otherwise SF_2.
2. Compress the chosen frames by running both upper and lower threads. Their encodings are interleaved and merged into a single output stream.

5.3 ANS-based PRBG with Key Rotation

The second version of our threaded ANS-based PRBG uses a key rotation to mask ANS binary encodings. They are then merged with the order of concatenation determined by K. The design is presented in Figure 7. As before, a symbol frame SF is permuted ahead of a bitstream generation session. The same frame SF is used throughout the whole PRBG session. Unlike in the previous version, binary encodings coming out from ANS are XOR-ed with a binary sequence $ROT(K, X)$, where K is a cryptographic key and X is a rotation offset derived from the current ANS state and the length k_s of binary encoding. The output streams from the top and bottom threads are merged (or interleaved) and they form a PRBG output stream. The merging operation is controlled by a cryptographic key K.

We use two versions of key rotation. The first one is a simple scheme that takes only a single 64-bit word as a key. The rotation offset is determined by the 3 least significant bits of the current ANS internal state X. The rotation happens only when we have accumulated 64 output bits of ANS encodings. After rotation, the ANS encodings are XOR-ed with the rotated key. As our

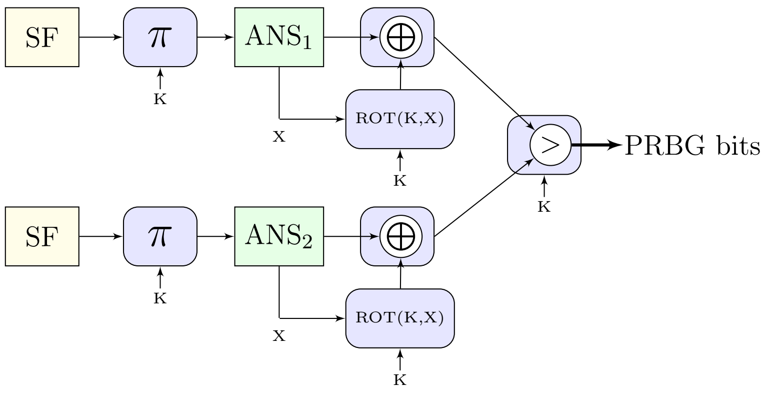

Fig. 7. Two-threaded ANS-based PRBG with key rotation

experiments have shown, this rotation version is weak as it fails to remove a local ANS encoding correlation.

The second version of rotation applies two 64-bit words ($K1$, $K2$) and is described as follows:

$$\begin{aligned} K1 &= ROT(K1, 1); \\ x &= LSB(X, 3); \\ t &= MSB(K2, x) \oplus LSB(K1, x); \\ K2 &= (K2 \ll x)|t; \end{aligned}$$

where X is the current ANS state, $LSB(K, n)$ and $MSB(K, n)$ return the n least or most significant bits of K, respectively, $(K \ll n)$ stands for the left shift of K by n positions and $a|b$ means concatenation of two binary strings a and b. Note that that every application of key rotation updates both keys $K1$ and $K2$.

6 Experiments

We focus our experiments on fast PRBG that are built using ANS with a relatively low number of states (say 128, 256 or 512). Sadly, security of such PRBG is not guaranteed as Theorem 3 works for big enough parameters R, where a local correlation among output bits is not a problem.

6.1 Output Permutation π^*_K

We implement three versions of the basic PRBG illustrated in Figure 4. They are as follows:

- $PRBG_{id}$ – π^*_K is the identity one and does not depend on K;
- $PRBG_{rot}$ – π^*_K is an XOR of ANS encodings with the first version of rotation from Section 5.3;
- $PRBG_{rot2}$ – π^*_K as above except we use the second version of rotation from Section 5.3;
- $PRBG_{keccak}$ – π^*_K is a single round of Keccak permutation [3].

Symbol frames used in our experiments are constructed from a long pseudorandom bit string (10Gb) that passes all NIST tests. We evaluate each PRBG using the NIST test suite [17]. We use 15 versions of ANS. Each version is labelled by a pair (S, R), where $S \in \{8, 16, 32, 64, 128, 256\}$ is the number of symbols handled by ANS and $R \in \{7, 8, 9\}$ is the parameter that determines the number of ANS states, which is 2^R. Instead of 18 possibilities, we experiment with 15 versions only. We experiment with symbol frames of three lengths: 6400, 12800 and 25600 symbols. As expected, all 15 versions of $PRBG_{id}$ fed by the three symbol frames fail almost all NIST tests after generating 10^9 of pseudorandom bits.

Tested Algorithm	Number of Failed Tests				Total Number of PRBG Variants Tested
	0	1	2	≥ 3	
PRBG_{id}	–	–	–	45	45
PRBG_{rot}	15	9	1	20	45
PRBG_{rot2}	20	4	2	19	45
PRBG_{keccak}	10	34	1	–	45

Table 5. Comparison of different implementations of PRBG (output of 1Gb)

Table 5 illustrates our results. Both permutations π^*_K based on rotations and a single round of Keccak produce a subset of PRBGs that passes all NIST tests (20, 15 and 10, respectively). Note that PRBG_{keccak} instances behave more consistently, as they never fail more than 3 tests. Looking at the above results, we see that both key rotating algorithms flawlessly pass more tests than the PRBG_{keccak}. It is interesting to notice that the majority of the failed tests for the PRBG_{rot2} are for PRBG variants that have more than 32 symbols chosen as their ANS parameter. There are only two cases for with more than 2 failed test for PRBG variants with up to 32 symbols.

We take a closer look at PRBG_{keccak} in order to reduce the complexity of the Keccak permutation and improve its efficiency. We try four versions of PRBG_{keccak} that apply the nonlinear layer χ and rotation. Table 6 shows the results.

Tested Algorithm	Number of Failed Tests				Total Number of PRBG Variants Tested
	0	1	2	≥ 3	
$\mathrm{PRBG}_{keccak1}$	–	–	–	45	45
$\mathrm{PRBG}_{keccak2}$	–	1	–	44	45
$\mathrm{PRBG}_{keccak3}$	2	–	1	42	45
$\mathrm{PRBG}_{keccak4}$	–	3	–	42	45

Table 6. Comparison of different implementations of PRBG with Keccak (output of 1Gb)

The χ layer of Keccak is based on operations on a state constructed from 25 64-bit words. We can identify 5 batches of state operations in the χ layer. By reducing the number of batches, we can increase the efficiency of our algorithms. At the same time, the quality of PRBG may suffer. We have designed 4 flavours of the Keccak layer. Each of them has a different number of χ layer operations. It means that $\mathrm{PRBG}_{keccak1}$ has a single batch, $\mathrm{PRBG}_{keccak2}$ has two batches and so forth up to $\mathrm{PRBG}_{keccak4}$ with 4 out of 5 batches of χ-layer operations. The results of our experiments are presented in Table 6. We can see that weakening the χ layer of the Keccak algorithms has a negative impact on the number of passed tests. We have hoped to see a more gradual degradation. Hence, we conclude that these lightwieght versions are not suitable for application.

6.2 Randomness Inflation

In this part, we fix the length of an input frame to 6400 symbols. The frame is then used n times to generate a binary sequence $(z_1, \ldots, z_n)$. When the frame is used n times, a new frame is taken and the process is repeated as many times as necessary to collect a required number of output bits. We test the binary sequence against the NIST tests. We say that PRBG inflates randomness n times with a threshold θ if $(z_1, \ldots, z_n)$ fails θ NIST tests. Table 7 gives the results.

PRBG Variant	ANS Parameters (S, R)	Output Length	Number θ of Failed Tests				
			$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
PRBG$_{rot}$	$(8, 9)$	10^8	159	158	161	145	154
PRBG$_{rot2}$	$(8, 9)$	10^8	154	1	185	160	2
PRBG$_{keccak}$	$(8, 9)$	10^8	0	159	154	158	158
PRBG$_{keccak^2}$	$(8, 9)$	10^8	159	6	154	160	160
PRBG$_{keccak^3}$	$(8, 9)$	10^8	0	0	1	0	3
PRBG$_{keccak^4}$	$(8, 9)$	10^8	154	0	3	3	1

Table 7. Randomness inflation

PRBG$_{keccak}$ allows generating 10^8 output bits by repeating each symbol frame twice. Subsequently, we have analyzed the impact of adding more Keccak rounds of per single output frame. We have denominated them as $PRBG_{keccak^n}$, where $n \in 2, 3, 4$, represents the number of additional rounds. We can see that running 3 or 4 additional rounds significantly improves the statistical quality of the output random bits. All version allow doing the same if the number of analyzed output bits is restricted to 10^7.

6.3 Efficiency

We analyse efficiency of our designs to see how fast they can produce pseudorandom bits. All tests are done using an Intel Core i7-5600U CPU with 12 GB of RAM and a 64bit version of OS. All algorithms are implemented in the Go language (Version 1.16.6). Tests are performed for a single-threaded versions of our PRBG generators and are presented in Table 8.

	$S = 8$, $R = 7$	$S = 32$, $R = 7$	$S = 128$, $R = 8$	$S = 128$, $R = 9$	$S = 256$, $R = 9$
Multi-Frame	184 Mbps	313 Mbps	593 Mbps	378 Mbps	641 Mbps
Key Rotating	282 Mbps	471 Mbps	683 Mbps	547 Mbps	733 Mbps

Table 8. Average throughputs of a **single thread** of a Multi-Frame and Key Rotating ANS-based PRBG achieved for different ANS parameters. S means the number of symbols in the alphabet of a source frame, and R controls the size of the encoding table.

It is easy to see that the speed depends on the number of symbols in the input frame. The bigger number S the faster generation of output bits. If efficiency is the prime concern, then we have at least two options for speeding up generation of pseudorandom bits.

- Instead of a distribution whose probabilities are natural powers of 1/2, we can use a distribution whose probabilities are much smaller than 1/2 and close to each other. Consequently, state equilibrium probabilities are no longer uniform and we need to deal with a persistent bias of encodings.
- We design ANS for a distribution whose probabilities are natural powers of 1/2 and allow symbols to occur with the same probabilities. As an illustration, consider ANS with 8 symbols that occur with probabilities $\{1/2, \ldots, 1/128, 1/128\}$. It generates ≈ 2 bits per symbol. Now if ANS is designed for 128 symbols that happen with uniform probability, then it produces 7 bits per symbol.

Besides, PRBGs may use multiple ANS threads, where its final throughput equals the sum of ANS encodings. An important fact is that security of such PRBGs increases with the number of ANS threads as an adversary needs make guesses about encoding lengths. Our discussion about efficient ANS-based PRBGs barely has scratched the surface. We are confident that we can substantially improve our PRBGs so it can effectively compete with PRBGs based on block ciphers such as Keccak or ChaCha. This challenge is left as a future research direction.

7 Conclusions and Future Research

We investigate generation of pseudorandom bits using the ANS compression algorithm. In principle, any compression algorithm removes all redundancy and should produce randomly looking bits. We show that ANS can produce binary encodings that achieve optimal entropy while compressing symbols whose probabilities are natural powers of 1/2. We prove that by assuming uniform distribution of ANS states, we can get PRBG that passes all statistical tests. Unfortunately, we are able to prove that the uniform probability distribution of states is asymptotic. This means that our construction is provably secure if the number of states is big "enough". It is an interesting research question to pinpoint the smallest ANS instance, whose output bits pass all NIST tests. This question closely relates to a design of random graphs without short cycles.

As efficient pseudorandom bit generation is always in high demand, we consider ANS instances that are very efficient. Clearly, their security becomes heuristic. We show that algebraic cryptanalysis of ANS for $R > 7$ becomes difficult and breaking it involves guessing of partition of binary encodings that are glued together into a single stream. To remove local correlations, we permute output bits using rotation and keccak transformations. To verify the quality of our PRBG variants, we use the NIST test suite. It turns out that the results are quite encouraging. Some variants produce long sequences (say 10^8 or 10^7 bits) that pass all NIST tests.

Our research can be extended by investigating the interaction between two ANS instances executed by two treads in parallel. An obvious advantage of a such PRBG is that merging encodings from the two ANS forces an adversary to guess their lengths. This potentially increases both efficiency and security.

References

[1] Werner Alexi, Benny Chor, Oded Goldreich, and Claus P. Schnorr. Rsa and rabin functions: Certain parts are as hard as the whole. *SIAM Journal on Computing*, 17(2):194–209, 1988.

[2] Ammar Alkassar, Thomas Nicolay, and Markus Rohe. Obtaining true-random binary numbers from a weak radioactive source. In Osvaldo Gervasi, Marina L. Gavrilova, Vipin Kumar, Antonio Laganà, Heow Pueh Lee, Youngsong Mun, David Taniar, and Chih Jeng Kenneth Tan, editors, *Computational Science and Its Applications – ICCSA 2005*, pages 634–646, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[3] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, 1986.

[5] Béla Bollobás. *Random Graphs*. Cambridge University Press, Cambridge, aug 2001.

[6] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[7] Seyit Camtepe, Jarek Duda, Arash Mahboubi, Paweł Morawiecki, Surya Nepal, Marcin Pawłowski, and Josef Pieprzyk. Compcrypt–lightweight ANS-based compression and encryption. *IEEE Transactions on Information Forensics and Security*, 16:3859–3873, 2021.

[8] Nicolas T. Courtois and Willi Meier. Algebraic attacks on stream ciphers with linear feedback. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 345–359, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[9] Christophe De Cannière and Bart Preneel. *Trivium*, pages 244–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] Leo Dorrendorf, Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the windows random number generator. *in ACM Conference on Computer and Communications Security*, pages 476–485, 2007.

[11] Jarek Duda. Asymmetric numeral systems. *Internet Archive*, arxiv-0902.0271:1–47, 2009.

[12] Ian Goldberg and David Wagner. Randomness and the netscape browser. *Dr. Dobb's Journal*, 1(1):1–1, 1996.

[13] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, pages 212–219, ACM, 1996. ACM Press.

[14] O Haggstrom. *Finite Markov Chains and Algorithmic Applications.* London Mathematical Society, London, UK, 2002.

[15] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[16] Donald Knuth. *The art of computer programming, Vol. 2 / Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, USA, 1973.

[17] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. *NIST Special Publication 800-22, Gaithersburg, MD, US,*, 800:163, 05 2001.

[18] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct 1997.

[19] Bijesh Shrestha. Multiprime Blum-Blum-Shub pseudorandom number generator. Thesis, Naval Postgraduate School, Monterey, California, 2016.

[20] Dan Shumow and Niels Ferguson. On the possibility of a back door in the NIST SP800-90 dual Ec PRNG. *Rump Session, Crypto 2007*, 1(1), 2007.

[21] Johannes vom Dorp, Joachim von zur Gathen, Daniel Loebenberger, Jan Lühr, and Simon Schneider. Comparative analysis of random generators. In *Algorithmic Combinatorics: Enumerative Combinatorics, Special Functions and Computer Algebra*, pages 181–196. Springer International Publishing, Berlin, Heidelberg, 2020.

[22] Wikipedia. RC4. `https://en.wikipedia.org/wiki/RC4`, 2021. Accessed Nov 27, 2021.

[23] Andrew C Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, pages 80–91. IEEE, 1982.

A ANS Algorithms

ANS compression is defined by the following three algorithms: initialisation, symbol frame encoding and binary frame decoding. Algorithm 3 shows initialisation that constructs encoding and decoding functions.

Algorithm 3: Initialises ANS

```
Input: A set of symbols $\mathbb{S}$, their probability distribution $p : \mathbb{S} \to [0,1]$ and a parameter $R \in \mathbb{N}^+$.
Output: Instantiation of coding and decoding functions:
  • the encoding functions $C(s,x)$ and $k_s(x)$;
  • the decoding functions $D(x)$ and $k(x)$.
Steps: Initialisation proceeds as follows:
  • calculate the number of states $L = 2^R$;
  • determine the set of states $\mathbb{I} = \{L, \dots, 2L-1\}$;
  • for each symbol $s \in \mathbb{S}$, compute integer $L_s \approx Lp_s$, where $p_s$ is probability of $s$;
  • define the symbol spread function $\overline{s} : \mathbb{I} \to \mathbb{S}$, such that $|\{x \in \mathbb{I} : \overline{s}(x) = s\}| = L_s$;
  • establish the coding function $C(s,y) = x$ for the integer $y \in \{L_s, \dots, 2L_s - 1\}$, which assigns states $x \in \mathbb{L}_s$
    according to the symbol spread function;
  • compute $k_s(x) = \lfloor \lg(x/L_s) \rfloor$ for $x \in \mathbb{I}$ and $s \in \mathbb{S}$. It gives the number of output bits per an encoding step;
  • construct the decoding function $D(x) = (s,y)$, which for a state $x \in \mathbb{I}$ assigns its unique symbol (given by
    the symbol spread function) and the integer $y$, where $L_s \leq y \leq 2L_s - 1$. Note that $D(x) = C^{-1}(x)$.
  • calculate $k(x) = R - \lfloor \lg(x) \rfloor$ or the number of bits that needs to be read out from the bitstream.
```

Algorithm 4 takes a symbol frame and compresses it into a binary frame. The binary frame together with the final ANS state is sent to a receiver.

Algorithm 4: Encodes Symbol Frames

```
Input: A symbol frame $\mathbf{s} = (s_1, s_2, \dots, s_\ell) \in \mathbb{S}^*$ and an initial state $x = x_\ell \in \mathbb{I}$.
Output: An output bit stream $\mathbf{b} = (b_1|b_2|\dots|b_\ell) \in \mathbb{B}^*$, where $|b_i| = k_{s_i}(x_i)$ and $x_i$ is state in $i$-th step.
Steps: for $i = \ell, \ell-1, \dots, 2, 1$ do
    $s := s_i$;
    $k = k_s(x) = \lfloor \lg(x/L_s) \rfloor$;
    $b_i = x \mod 2^k$;
    $x := C(s, \lfloor x/2^k \rfloor)$;
Store the final state $x_0 = x$;
```

Given a binary frame and the final ANS state, the receiver applies Algorithm 5 and reconstructs the corresponding symbol frame.

Algorithm 5: Decodes Binary Frames

```
Input: A binary frame $\mathbf{b} \in \mathbb{B}^*$ and the final state $x = x_0 \in \mathbb{I}$ of the encoder.
Output: Symbol frame $\mathbf{s} \in \mathbb{S}^*$.
Steps: while $\mathbf{b} \neq \emptyset$ do
    $(s,y) = D(x)$;
    $k = k(x) = R - \lfloor \lg(x) \rfloor$;
    $b = MSB(\mathbf{b})_k$;
    $\mathbf{b} := LSB(\mathbf{b})_{|\mathbf{b}|-k}$;
    $x := 2^k y + b$;
```

Note that $LSB(\mathbf{b})_\ell$ and $MSB(\mathbf{b})_\ell$ stand for ℓ least and most significant bits of $\mathbf{b}$, respectively.

B Example of ANS

We consider a symbol source $\mathbb{S} = \{s_1, s_2, s_3, s_4, s_5\}$, where $p_1 = \frac{1}{2}$, $p_2 = \frac{1}{4}$, $p_3 = \frac{1}{8}$, $p_4 = \frac{1}{16}$, $p_5 = \frac{1}{16}$ and free parameter $R = 4$. The number of states $L = 2^R = 16$ and the state set $\mathbb{I} = \{16, 17, \dots, 31\}$. We follow the initialisation.

- Determine symbol spread function $\overline{s} : \mathbb{I} \to \mathbb{S}$ such that

$$\overline{s}(x) = \begin{cases} s_1 & \text{if } x \in \{17, 18, 22, 24, 25, 27, 28, 31\} = \mathbb{L}_1 \\ s_2 & \text{if } x \in \{20, 23, 29, 30\} = \mathbb{L}_2 \\ s_3 & \text{if } x \in \{16, 19\} = \mathbb{L}_3 \\ s_4 & \text{if } x \in \{21\} = \mathbb{L}_4 \\ s_5 & \text{if } x \in \{26\} = \mathbb{L}_5 \end{cases}$$

  where $L_1 = |\mathbb{L}_1| = 8$, $L_2 = |\mathbb{L}_2| = 4$, $L_3 = |\mathbb{L}_3| = 2$, $L_4 = |\mathbb{L}_4| = 1$ and $L_5 = |\mathbb{L}_5| = 1$.
- Write the coding function $C(s, y)$, which can be represented as

$s\backslash y$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s_1	–	–	–	–	–	–	–	17	18	22	24	25	27	28	31
s_2	–	–	–	20	23	29	30	–	–	–	–	–	–	–	–
s_3	–	16	19	–	–	–	–	–	–	–	–	–	–	–	–
s_4	21	–	–	–	–	–	–	–	–	–	–	–	–	–	–
s_5	26	–	–	–	–	–	–	–	–	–	–	–	–	–	–

- Construct the frame encoding table $\mathbb{E}(x_i, s_i) = (x_{i+1}, b_i) \overset{def}{\equiv} \binom{x_{i+1}}{b_i}$ as follows:

$s_i\backslash x_i$	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
s_1	$\binom{17}{0}$	$\binom{17}{1}$	$\binom{18}{0}$	$\binom{18}{1}$	$\binom{22}{0}$	$\binom{22}{1}$	$\binom{24}{0}$	$\binom{24}{1}$	$\binom{25}{0}$	$\binom{25}{1}$	$\binom{27}{0}$	$\binom{27}{1}$	$\binom{28}{0}$	$\binom{28}{1}$	$\binom{31}{0}$	$\binom{31}{1}$
s_2	$\binom{20}{00}$	$\binom{20}{01}$	$\binom{20}{10}$	$\binom{20}{11}$	$\binom{23}{00}$	$\binom{23}{01}$	$\binom{23}{10}$	$\binom{23}{11}$	$\binom{29}{00}$	$\binom{29}{01}$	$\binom{29}{10}$	$\binom{29}{11}$	$\binom{30}{00}$	$\binom{30}{01}$	$\binom{30}{10}$	$\binom{30}{11}$
s_3	$\binom{16}{000}$	$\binom{16}{001}$	$\binom{16}{010}$	$\binom{16}{011}$	$\binom{16}{100}$	$\binom{16}{101}$	$\binom{16}{110}$	$\binom{16}{111}$	$\binom{19}{000}$	$\binom{19}{001}$	$\binom{19}{010}$	$\binom{19}{011}$	$\binom{19}{100}$	$\binom{19}{101}$	$\binom{19}{110}$	$\binom{19}{111}$
s_4	$\binom{21}{0000}$	$\binom{21}{0001}$	$\binom{21}{0010}$	$\binom{21}{0011}$	$\binom{21}{0100}$	$\binom{21}{0101}$	$\binom{21}{0110}$	$\binom{21}{0111}$	$\binom{21}{1000}$	$\binom{21}{1001}$	$\binom{21}{1010}$	$\binom{21}{1011}$	$\binom{21}{1100}$	$\binom{21}{1101}$	$\binom{21}{1110}$	$\binom{21}{1111}$
s_5	$\binom{26}{0000}$	$\binom{26}{0001}$	$\binom{26}{0010}$	$\binom{26}{0011}$	$\binom{26}{0100}$	$\binom{26}{0101}$	$\binom{26}{0110}$	$\binom{26}{0111}$	$\binom{26}{1000}$	$\binom{26}{1001}$	$\binom{26}{1010}$	$\binom{26}{1011}$	$\binom{26}{1100}$	$\binom{26}{1101}$	$\binom{26}{1110}$	$\binom{26}{1111}$

- Build a decoding table. The decoding function $D(x) = (s, y)$, where the integer y is given by the following table

x	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
y	2	8	9	3	4	1	10	5	11	12	1	13	14	6	7	15

  The decoding table $\mathbb{D}(x_i, b_i)$ can be represented as follows

x_i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
s_i	s_3	s_1	s_1	s_3	s_2	s_4	s_1	s_2	s_1	s_1	s_5	s_1	s_1	s_2	s_2	s_1
k	3	1	1	3	2	4	1	2	1	1	4	1	1	2	2	1
x_{i+1}	$16{+}b_i$	$16{+}b_i$	$18{+}b_i$	$24{+}b_i$	$16{+}b_i$	$16{+}b_i$	$20{+}b_i$	$20{+}b_i$	$22{+}b_i$	$24{+}b_i$	$16{+}b_i$	$26{+}b_i$	$28{+}b_i$	$24{+}b_i$	$28{+}b_i$	$30{+}b_i$

  Note that $x_{i+1} = 2^k y + b_i$, where b_i is an integer that corresponds to the binary string.

It is easy to check that our ANS is optimal as the source entropy $H(\mathbb{S})$ equals to the average length of binary encodings.