# ULTRAFAST PSEUDORANDOM NUMBER GENERATION USING PSEUDORANDOM PERMUTATIONS AND MAPPINGS

**by**

JIE LI

A dissertation submitted to the Graduate Faculty in Computer Science
in partial fulfillment of the requirements for the degree of Doctor of Philosophy,
The City University of New York

2013

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirements for the degree of Doctor of Philosophy.

| | |
|---|---|
| Date | Professor **Paula Whitlock**<br>Chair of Examining Committee |
| Date | Professor **Robert M. Haralick**<br>Executive Officer |

Professor **James Cox**, Brooklyn College, CUNY

Professor **Rosario Gennaro**, The City College of New York, CUNY

Professor **Michael Mascagni**, Florida State University

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

# Abstract


Ultrafast Pseudorandom Number Generation Using Pseudorandom Permutations and Mappings

by

Jie Li

Adviser: Professor Paula Whitlock

Pseudorandom numbers have broad applications in science, technology, entertainment, etc. So far many pseudorandom number generators (PRNGs) have been developed, but dedicated high performance high quality PRNGs are still in demand. In light of this, we propose a new design approach which combines byte-oriented pseudorandom permutations and integer-oriented pseudorandom mappings. Pseudorandom permutations are used for state initialization and reseeding. Pseudorandom mappings are used for state transition and pseudorandom number generation. Several PRNGs are designed using this approach. The performance tests show they surpass the existing pseudorandom number generators in both non-cryptographic category and cryptographic category. The proposed non-cryptographic PRNG reaches a generation speed of half clock cycle per byte on an Intel Core i3 processor, and the cryptographically secure PRNG also runs into one clock cycle per byte. They demonstrate excellent randomness properties as attested by the NIST statistical tests, the new Diehard battery of tests, and the TestU01 batteries of tests. The non-cryptographic PRNG is also designed to meet a couple of other requirements, including long period, high-dimensional equidistribution, quick recovery from biased states, and ease of use. For the cryptographically secure PRNG, security has been taken into account throughout the design. Besides the key scheduling algorithm, which has an avalanche effect comparable to that of standard hash functions, a new two-layer design paradigm is adopted, which functionally divides the internal state into two parts, with the first part serving as a source

of entropy and periodically reseeding the second part. The generator has a huge internal state and employs a high quality state update function, which renders a very long expected period. The overall security of the generator is carefully analyzed in the context of various known cryptanalytic attacks, state compromise extension attacks, and next-bit test.

Besides deterministic pseudorandom number generation, the proposed PRNGs can also work in a non-deterministic mode. In this mode, the generators behave like a true random number generator by periodically querying some non-deterministic random sources and using them as unpredictable sources of entropies. Running in this mode has virtually no impact on the cost, performance, availability, or usability of the generators.

*To my parents, my husband, and my sons*

# Acknowledgments

I owe my deepest gratitude to my mentor Professor Paula Whitlock. Her wisdom, kindness, and dedication have helped me to get through many difficult times and led me to achieve my research goals. I feel so lucky and honored to have such a wonderful mentor.

My heart is also full of gratitude to my former mentor Professor Michael Anshel for guiding me to the world of cryptography and security and for his generosity, understanding, and continuous support.

My special thanks go to my examining committee members, Professor James Cox, Professor Rosario Gennaro, and Professor Michael Mascagni, for spending their precious time reviewing my thesis and giving me insightful suggestions and comments.

I greatly appreciate Professor Robert Haralick, Professor Kent Boklan, Professor Myung Lee, and Professor Xiangdong Li for their valuable advice on my research.

My sincere thanks also go to Professor Ted Brown, Professor Robert Feinerman, and Mrs. Esther Owens for providing me the financial support needed for my study and research.

I am very grateful for all the help and support I have received from Janos Pach, Zhigang Zhu, Christina Sormani, Yves Jean, Amotz Bar-Noy, Lina Garcia, Anthony Francis San Lucas, and many other faculty, staff, and students of the City College, Lehman College, John Jay College, and the Graduate Center of the City University of New York.

I am indebted to my amazing husband Jianliang for his love, encouragement, and invaluable discussions. Tons of thanks to my sons, Zhi, Evan, and Chris, for the joy, happiness, and strength they have brought me. I am eternally grateful to my beloved parents and my whole family for believing in me, encouraging me, and supporting me over the years.

# Contents

# List of Figures

# List of Tables

# List of Algorithms (Listings)

# Abbreviations and Acronyms

| | |
|---|---|
| AES | Advanced Encryption Standard |
| ALFG | Additive Lagged Fibonacci Generator |
| BOS | Byte-Oriented State |
| CSPRNG | Cryptographically Secure Pseudorandom Number Generator |
| DES | Data Encryption Standard |
| IOS | Integer-Oriented State |
| ISA | Initialization Shuffling Algorithm |
| IV | Initialization Vector |
| KSA | Key Scheduling Algorithm |
| LCG | Linear Congruential Generator |
| LFG | Lagged Fibonacci Generator |
| LFSR | Linear Feedback Shift Register |
| MARC | Modified Alleged RC4 |
| MARC-bb | MARC as a Building Block |
| MLCG | Multiplicative Linear Congruential Generator |
| MLFG | Multiplicative Lagged Fibonacci Generator |
| MRG | Multiple Recursive Generator |
| MT | Mersenne Twister |
| NIST | National Institute of Standards and Technology |
| PRGA | Pseudorandom Generation Algorithm |
| PRNG | Pseudorandom Number Generator |
| QRNG | Quantum Random Number Generator |

| | |
|---|---|
| SFMT | SIMD-oriented Fast Mersenne Twister |
| SHA | Secure Hash Algorithm |
| SIMD | Single Instruction Multiple Data |
| TRNG | True Random Number Generator |
| WELL | Well Equidistributed Long-period Linear generator |

# Chapter 1

# Introduction

## 1.1 Research Motivation

Random numbers and pseudorandom numbers are widely used in many kinds of applications, for example, simulations, numerical analysis, stochastic optimizations, cryptography, gaming and gambling, and lottery, among others. Their applications can be classified into two categories: non-cryptographic applications and cryptographic applications. Designing pseudorandom number generators (PRNGs) for applications in each category has its challenges. Designing PRNGs for high speed applications such as Monte Carlo simulations and data encryption in cloud computing is even more difficult.

Simulations are among the most important non-cryptographic applications. They are important means for evaluating complex stochastic dynamic systems. A high quality PRNG is the key to the success of these simulations. The quality of a PRNG can be measured using a couple of metrics, including randomness, speed, period, equidistribution, and usability. Although many PRNGs have been developed so far, few render a satisfactory performance in terms of the above metrics. The widely used Linear Congruential Generators (LCGs), Multiple Recursive Generators (MRGs), and Lagged Fibonacci Generators (LFGs), the efficient Xorshift generators [1], the well-known Mersenne Twister (MT) [2], and the two PRNGs designed to supersede MT, Well Equidistributed Long-period Linear generators

(WELL) [3] and SIMD-oriented Fast Mersenne Twister (SFMT) [4], all leave room for improvement in one respect or another.

Pervasive data encryption is among the most important cryptographic applications of pseudorandom numbers. Pervasive data encryption means two things: encrypt all data and encrypt data while they are moving on the wire and at rest as well. Not all data are secrets and need to be secured, but such a need arises when data classification is difficult or expensive (e.g., in cloud computing) or when encryption needs to be done at a layer below the application (e.g., for whole disk or communication channel encryption). Encryption at rest refers to the fact that the data is physically stored in an encrypted format. This is different from encryption in flight, which is only applied to data to be transported. Encryption at rest may affect the usability of some applications, but it is getting more popular nowadays for several reasons. First, most computers and computing devices are connected to networks and face attacks that keep growing in both quantity and complexity. Second, the boom of virtual computing and cloud computing makes it difficult or impossible to put up a physical defense line against attacks. Third, the risk of losing sensitive data increases due to the wide use of portable computing devices, which are more likely to get lost or be stolen. Finally, the breach of data at rest usually has a more serious consequence than the breach of data on the wire, since the former can contain much more information than the latter.

Past several years have witnessed significant advances in data encryption, including the release of Advanced Encryption Standard (AES) [5] and the development of eStream project (`http://www.ecrypt.eu.org/stream/`). Nonetheless, data encryption is still a costly operation and its performance penalty is still too high. For example, the data encryption speed of AES on an Intel Core i3 personal computer is about two times slower than the speed of disk I/O. This means data I/O performance will be degraded by a factor of 3 when data are encrypted using AES. Stream ciphers such as RC4 [6] and eStream finalists (HC-128, Rabbit, Salsa, and Sosemanuk) [7–10] are faster than block cipher AES, but they still slow down data access by a factor of 1.5 to 2. The fast development of the Internet and the ongoing transition from private computing to cloud computing call for more data

encryption and more efficient data encryption.

In summary, although much work has been done and many PRNGs have been proposed, the qualities and performances of existing PRNGs still cannot meet the needs of many important applications.

## 1.2 Contributions of the Thesis

This thesis focuses on the development of ultrafast pseudorandom number generators in both non-cryptographic and cryptographic categories. A new design approach, which combines byte-oriented pseudorandom permutations and integer-oriented pseudorandom mappings, is proposed for high performance high quality pseudorandom number generation. Several PRNGs are designed using this approach and presented in this thesis.

The first PRNG, MARC, is a new variant of RC4 stream cipher. It enhances the security of RC4 by modifying its key scheduling algorithm and improves the performance by modifying its pseudorandom generation algorithm. It retains the simplicity of RC4 and is much faster than RC4. The key scheduling algorithm of MARC has an avalanche effect comparable to that of standard hash functions. In this thesis, a simplified MARC, called MARC-bb, is used as a building block to construct more advanced PRNGs.

The second PRNG, called MaD0, is designed for high speed simulations and other non-cryptographic applications. MaD0 has a state space of 2240 bits and an expected period length around $1.42 \times 10^{337}$. It employs fast integer-oriented pseudorandom mappings for state transition. This makes MaD0 more chaotic than most non-cryptographic PRNGs. MaD0 shows an excellent randomness property as attested by the NIST statistical tests, the new Diehard battery of tests, and the TestU01 batteries of tests. Being a 64-bit generator, MaD0 enjoys a speed far out of the reach of any other 64-bit PRNGs we are aware of and is even faster than the 128-bit SFMT. It also demonstrates a good high-dimensional equidistribution property, fast recovery ability from biased states, and ease of use and error-proofing advantages. All the above mentioned features make MaD0 a unique and attractive

candidate for parallel and distributed simulations and many other serious applications.

The third PRNG, called MaD3, is a cryptographically secure PRNG (CSPRNG). It maintains a small byte-oriented state, whose transition follows pseudorandom permutations, and a large integer-oriented state, whose transition follows pseudorandom mappings. MaD3 generates high quality pseudorandom numbers and runs into one clock cycle per byte on a typical personal computer, which is several times faster than any CSPRNGs we know. It has a state space of 10520 bits and an expected period length around $10^{1783}$. MaD3 resists various cryptanalytic attacks, including special attacks mounted against RC4, time-memory tradeoff attacks, guess-and-determine attacks, algebraic attacks, distinguishing attacks, differential attacks, and side channel attacks. It can also withstand state compromise extension attacks due to the use of non-invertible pseudorandom mappings and the design that the byte-oriented state serves as an unpredictable source of entropy and periodically reseeds the integer-oriented state during pseudorandom number generation. For its excellent statistical property, ultrafast speed, and strong resistance against various attacks, MaD3 is well suited for pervasive data encryption and can be used in a wide range of other cryptographic applications.

Besides deterministic pseudorandom number generation, a non-deterministic pseudorandom number generation scheme is also proposed in this thesis. Both MaD0 and MaD3 can work in non-deterministic mode. In this mode, both generators behave like a true random number generator by periodically querying some non-deterministic sources and using them as unpredictable sources of entropies. Running in this mode has virtually no impact on cost or performance. Nor does it affect the availability or usability of the generators since no dedicated device or special setup is needed.

## 1.3 Outline of the Thesis

This thesis is structured as follows.

- Chapter 1 sets forth the research motivation and the contributions of the thesis.

- Chapter 2 gives an overview on existing pseudorandom number generators, followed by a brief review of (pseudo)random permutations and (pseudo)random mappings.
- Chapter 3 outlines the design criteria, describes the assessment methods, and lists the notations used in the thesis.
- Chapter 4 introduces the pseudorandom number generator MARC.
- Chapter 5 investigates the iteration-reduced version of MARC.
- Chapter 6 describes the non-cryptographic pseudorandom number generator MaD0.
- Chapter 7 presents the cryptographically secure pseudorandom number generator MaD3.
- Chapter 8 provides the non-deterministic pseudorandom number generation scheme.
- Chapter 9 summarizes the thesis.

# Chapter 2

# Background and Literature Review

A short overview of existing pseudorandom number generators is first given in this chapter. The concepts of random permutations and random mappings are then reviewed. The period lengths of random permutations and random mappings are also analyzed.

## 2.1 Pseudorandom Number Generators

A random number generator is a device or algorithm which outputs a sequence of statistically independent and unbiased binary digits [11]. Random number generators can be divided into two categories: true random number generators (TRNGs) and pseudorandom number generators (PRNGs).

True random number generators use physical phenomena such as radioactive decay, thermal noise, and atmospheric noise to generate unpredictable and irreproducible random sequences. However, such physical phenomena may produce biased or correlated outputs, thus some post-processing is usually needed to improve their statistical properties. TRNGs are not widely adopted so far for several reasons: too expensive, relatively slow (although high speed TRNGs such as quantum random number generators are also available), not generally available, and not reproducible.

Pseudorandom number generators, on the other hand, are often used for their speed and their reproducibility. A PRNG uses a deterministic algorithm to generate pseudoran-

dom sequences that are determined by the initial seeds. According to their applications, PRNGs can be further classified into non-cryptographic PRNGs and cryptographically secure PRNGs (CSPRNGs).

### 2.1.1 Non-cryptographic Pseudorandom Number Generators

Non-cryptographic PRNGs, simply referred to as PRNGs here, are widely used in computer simulations, sampling, recreation, numerical analysis, etc. Many types of PRNGs have been developed, including Linear Congruential Generators (LCGs), Multiple Recursive Generators (MRGs), Lagged Fibonacci Generators (LFGs), Xorshift generators, Mersenne Twister (MT) and its successor Well Equidistributed Long-period Linear generators (WELL) and SIMD-oriented Fast Mersenne Twister (SFMT), and so on. We give a brief overview of these generators in the following subsections.

#### 2.1.1.1 Linear Congruential Generators

A Linear Congruential Generator (LCG) is defined by the simple recurrence relation:

$$x_n = (ax_{n-1} + c) \bmod m \quad (n \geq 1)$$

and is usually expressed as LCG ($m$, $a$, $c$), where $x_0$, $x_1$, $x_2$, ... is the pseudorandom sequence and $m$, $a$, $c$ are integers. The quality of an LCG depends on the choice of $m$, $a$, and $c$. The maximum period of the linear congruential generator is $m$. If $c = 0$, the generator is often called a Multiplicative Linear Congruential Generator (MLCG) and its maximum period is $m - 1$. Most existing LCGs choose $m = 2^k$ ($k$ is a positive integer), in which case the costly modulus operations can be converted into efficient bitwise AND operations or completely omitted if $k = 2^i \times 8$ ($i = 0, 1, 2, ...$). Nevertheless, the low order bits of pseudorandom numbers generated from this type of LCGs have rather short period lengths [12]. This is the reason that some LCGs only output the high order bits. Among the widely used LCGs, none can generate high quality pseudorandom numbers according to the statistical testing results given in [13]. Although much efforts have been made to search for better

LCGs [12, 14–16], few can meet the needs of today's advanced simulations in terms of randomness, period length, equidistribution, and so on.

#### 2.1.1.2 Multiple Recursive Generators

A generalization of multiplicative linear congruential generators are Multiple Recursive Generators (MRGs) whose recurrence relations follow the form:

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k}) \bmod m$$

where $m$ and $k$ are positive integers, and $k$ is called the order of the generator. An MRG achieves its maximal period length $m^k - 1$ if and only if $m$ is prime and the characteristic polynomial $f(z) = z^k - a_1 z^{k-1} - \cdots - a_{k-1} z - a_k$ is a primitive polynomial modulo $m$ [17]. To achieve good quality of lattice structure, maximal period, and easy implementation, extensive computer search was performed in order to find good parameters. Good MRGs and combined MRGs whose parameters are carefully selected have been proposed [18–20]. One of the well-known combined multiple recursive generators is MRG32k3a [18], which is widely used in simulations and statistical software such as Matlab and SAS. MRG32k3a combines two multiple recursive generators of order 3. It has a period length $2^{191}$ and a very good randomness property. It is slower than LCGs, but can be parallelized to achieve a higher performance.

#### 2.1.1.3 Lagged Fibonacci Generators

Lagged Fibonacci Generators (LFGs) use an initial set of numbers $x_1, x_2, \cdots, x_r$ and two "lags" $r$ and $s$ with $r > s$, and use recurrence $x_n = (x_{n-s} \circ x_{n-r}) \bmod m$, $0 < s < r < n$ to generate a sequence of pseudorandom numbers, where operation $\circ$ can be addition (+), subtraction (-), multiplication (*), or bitwise exclusive-or ($\oplus$) [21]. An LFG is usually denoted as $F(m, r, s, \circ)$. When using addition, an LFG is often referred to as an Additive Lagged Fibonacci Generator (ALFG); when using multiplication, an LFG is often referred to as a Multiplicative Lagged Fibonacci Generator (MLFG). An ALFG is in fact a special case of MRG. If the modulus is a power of 2, say $2^p$, an ALFG can reach

a maximal period $(2^r - 1)2^{p-1}$ and an MLFG can reach a maximal period $(2^r - 1)2^{p-3}$. To achieve the maximal period, the characteristic polynomial $f(z) = z^r - z^s - 1$ must be primitive modulo 2 and at least one of the $x_1, x_2, \cdots, x_r$ is odd [22]. Studies suggest that MLFGs with proper choice of parameters can have very good statistical behavior while LFGs using bitwise exclusive-or should be avoided. Several good examples of LFGs are $F(2^{32}, 17, 5, *)$ with period $(2^{17} - 1)2^{29}$ and $F(2^{64}, 1279, 861, *)$ with period around $2^{1339}$ [13, 21]. LFGs are generally slower than LCGs and faster than MRGs. LFGs are good candidates for parallel pseudorandom number generators [23, 24].

#### 2.1.1.4 Xorshift generators

Xorshift generators are a class of high speed PRNGs [1]. They produce output sequences by repeatedly using xorshift operations: XOR (exclusive-or) a computer word with a shifted version of itself. The high speed comes from that they use only three xorshift operations to generate one pseudorandom number. Xorshift generators can not generate high quality pseudorandom numbers. Some more complicated Xorshift generators that use more than three xorshift operations are proposed in [25]. They are much slower and not much better than those given in [1] in terms of randomness. A better solution is given in [26], which combines the output of Xorshift generators with the output of a Weyl generator. The new generators are collectively named as Xorgens generators. They are fast, can generate high quality pseudorandom sequences, and have long periods.

#### 2.1.1.5 Mersenne Twister and its successors

Mersenne Twister is a pseudorandom number generator based on matrix linear recurrence over a finite binary field. MT is well-known for its long period and high-dimensional equidistribution property. It is one of the main PRNGs used in statistical simulations. MT has several variants and the commonly used one is MT19937, whose state contains 624 32-bit integers and whose period is as long as $2^{19937} - 1$. WELL and SFMT are deemed better choices for any applications currently using MT. They have better equidistribution

properties and similar statistical properties compared with MT. They both overcome one major shortcoming of MT, i.e., the generated pseudorandom numbers are heavily 0-biased for a long period of time if MT is initialized or falls into a state with many 0's. MT and WELL are relatively slow compared with LCG generators and Xorshift generators. SFMT is a 128-bit PRNG and is faster than LCG generators and Xorshift generators on platforms that support Streaming SIMD Extensions 2 (SSE2) instruction set. MT, WELL, and SFMT generate good quality pseudorandom sequences except a shortcoming in linear complexity [3].

### 2.1.2 Cryptographically Secure Pseudorandom Number Generators

Pseudorandom numbers are of paramount importance in cryptography. Keystreams, keys, seeds, salts, and challenges used in various cryptosystems are all assumed to be random. A CSPRNG usually falls into one of the following three categories.

#### 2.1.2.1 CSPRNGs based on number theoretic problems

Some CSPRNGs are based on the intractability of the underlying mathematical hard problems. The RSA pseudorandom bit generator is based on the computational difficulty of integer factorization problem. The Blum-Blum-Shub pseudorandom bit generator [27] assumes the intractability of the quadratic residuosity problem. The modular multiplications used in both generators are very inefficient. This has substantially limited their applications. Another number theoretic problem that is widely used in CSPRNGs is the elliptic curve discrete logarithm problem. Several elliptic curves with associated curve points are specified and approved in [28] for the use in elliptic curve pseudorandom bit generators.

#### 2.1.2.2 CSPRNGs based on cryptographic primitives

Some CSPRNGs employ ad-hoc techniques for pseudorandom number generation. They are usually based on cryptographic primitives such as cryptographic hash functions and block cipher algorithms. A necessary and sufficient condition for the existence of

CSPRNGs is the existence of one-way functions [29, 30]. While there is no proof that theoretically perfect one-way function exists, some functions including cryptographic hash functions and block cipher algorithms are considered computationally difficult to invert in practice.

Two well-known standardized CSPRNGs in this category are ANSI X9.17 generator [31], which uses DES two-key triple-encryption algorithm, and FIPS 186 generator [32], which uses SHA1 or DES algorithm as a one-way function. They were deemed good for many cryptographic applications, assuming the underlying cryptographic primitives were secure. After DES was revealed to be insecure and later replaced by AES, the above two standards were withdrawn. Currently, NIST recommends two random number generator mechanisms based on approved hash functions and block cipher algorithms [28]. The hash functions that are approved are SHA1, SHA224, SHA256, SHA384, and SHA512 [33], and the approved block cipher algorithms are three-key Triple Data Encryption Algorithm (TDEA) and AES [5].

A stream cipher generates a keystream that looks random and is XORed with plaintext to produce ciphertext. Essentially, the keystream generator is a pseudorandom number generator. The security of stream ciphers is based on the cryptographic security of keystream generators. The closer the keystream generator's output is to random, the harder it is to break the stream cipher [6].

#### 2.1.2.3 CSPRNGs with entropy inputs

This category of CSPRNGs produce pseudorandom numbers non-deterministically like TRNGs. They achieve this by frequently refreshing their internal states using unpredictable data from one or several external entropy sources. One example is the Linux PRNG [34], which processes events from different entropy sources, including user inputs (such as keyboard inputs and mouse movements), disk timings, and interrupt timings. The security of the generator strongly relies on the cryptographic primitive SHA1 hash function, which is used for output generation and entropy transfers between the input pool and the out-

put pools. Another example is Yarrow-160 [35]. Yarrow algorithm assumes that enough entropy can be accumulated to get the PRNG into an unguessable state. It uses SHA1 to accumulate entropy inputs and uses three-key triple-DES in counter mode to generate outputs. Entropy estimation and reseeding are critical for the security of Yarrow, which is limited to 160 bits due to the size of its entropy accumulation pools.

## 2.2 (Pseudo)random Permutations and (Pseudo)random Mappings

### 2.2.1 Random Permutations and Pseudorandom Permutations

A permutation of a set $X_n$ of *n* elements is defined as a bijection from $X_n$ to itself, i.e., a one-to-one and onto mapping $f : X_n \rightarrow X_n$. There are $n!$ permutations of *n* elements. A random permutation is one in which each of the $n!$ possible permutations occurs equally often.

There are various random permutation algorithms, among which are the most well-known Fisher-Yates Shuffle [36] and its modern versions, Knuth Shuffle [17], and Durstenfeld algorithm [37]. The Knuth Shuffle Algorithm works as follows.

Let $x_1$, $x_2$, ..., $x_n$ be a set of *n* numbers to be shuffled.

1. Set $i = n$.
2. Generate *j* uniformly from the integers 1, 2, . . ., *i*.
3. Exchange $x_i$and $x_j$.
4. Set *i* = *i* -1. If *i* > 1, go to step 2; otherwise, terminate.

Producing uniformly distributed random index *j* in the range 1 through *i* is the most crucial step. In practice, a software-based pseudorandom number generator is used in step 2. As long as the underlying generator for index *j* is unbiased, the generated permutation is unbiased.

Random permutation statistics have been intensely studied in combinatorics and probability. The average period of a random permutation can be easily deduced. Let $N$ denote the number of possible permutations and $P_k$ denote the probability that the period length is $k$. For a random permutation, the probability that the period equals 1 is $P_1 = \frac{1}{N}$. This is because, to get a period of 1, the second permutation should be the same as the first one, which has a probability of $\frac{1}{N}$. The probability that the period equals 2 is $P_2 = \frac{N-1}{N} \times \frac{1}{N-1} = \frac{1}{N}$. It is not difficult to see that for any $k \geq 1$,

$$P_k = \frac{1}{N-k+1} \prod_{i=1}^{k-1} \frac{N-i}{N-i+1} = \frac{1}{N}$$

The average period length thus equals

$$\bar{k} = \sum_{k=1}^{N} P_k k = \frac{1}{N} \sum_{k=1}^{N} k = \frac{N+1}{2} \approx \frac{N}{2}$$

A pseudorandom permutation is a function that "behaves" like a random permutation. In cryptography, pseudorandomness is defined more strictly using computational complexity theory. A pseudorandom permutation refers to a function that is indistinguishable from a random permutation using any polynomial-time distinguisher [38].

Pseudorandom permutations are used in cryptography, such as, pseudorandom number generation, block ciphers, and stream ciphers. Pseudorandom permutations are defined and further constructed from pseudorandom functions using Feistel construction in [39]. A block cipher can be considered as an instance of a pseudorandom permutation. For example, AES has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. For a given key, AES is a permutation of the $2^{128}$ possible values. As a purportedly secure block cipher, AES is expected to be indistinguishable from a random permutation [40].

Stream cipher RC4 uses a 256-byte permutation table to generate pseudorandom sequences. RC4 shuffling algorithm used in the key scheduling phase and the pseudorandom number generation phase has been discovered to be biased in the first few rounds. To find out when the nonuniformity of RC4 shuffling algorithm vanishes completely, an idealized

model of RC4 is proposed and analyzed using the theory of random shuffles in [41]. The empirical evidence shows that after $1.4n\,log_2 n \approx 12 \cdot 256$ (for $n = 256$) discarded shuffles, the idealized model of RC4 is close to a uniform distribution on permutation.

### 2.2.2 Random Mappings and Pseudorandom Mappings

Random mapping is defined in [42] as follows. Assume a single-valued mapping from a finite set of *n* elements into itself is $f : X_n = \{1, 2, ..., n\} \rightarrow X_n$. Let $\sum_n$ denote the set of all single-valued mappings of $X_n$ into itself. If a uniform distribution is defined on the set $\sum_n$, the random mapping $\sigma$ is the identity transformation of the set $\sum_n$ into itself, and $P(\sigma = f) = n^{-n}$ for any $f \in \sum_n$. A general random mapping is not bijective, thus is non-invertible.

Consider a random number generator $G$ with *n*-bit internal state and assume the state transition of $G$ is a random mapping. Denote the state transition function by $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$ and denote by $\sum_m$ the collection of all functions from $\{0, 1\}^n$ into itself, where $m = 2^n$. It is shown that, in a random mapping, cycles occur after about $\sqrt{m} = 2^{n/2}$ iteration steps. For more detailed analysis of random mapping statistics, please see [43]. We now present another approach which reaches the same conclusion.

In a random mapping transition, each state has an equal probability to become the next state. For an *n*-bit internal state, the probability that the period length is equal to or larger than *k* is

$$\begin{aligned} P_{\geq k} &= \left(\frac{2^n - 1}{2^n}\right)\left(\frac{2^n - 2}{2^n}\right)\cdots\left(\frac{2^n - (k-1)}{2^n}\right) \\ &= \frac{2^n!}{2^{nk}(2^n - k)!} \end{aligned}$$

Therefore the probability that the period length equals *k* is

$$P_k = P_{\geq k} - P_{\geq k+1}$$

and the average period length is

$$\begin{aligned}
\bar{k} &= \sum_{k=1}^{2^n} P_k k \\
&= \sum_{k=1}^{2^n} \left(P_{\geq k} - P_{\geq k+1}\right) k \\
&= P_{\geq 1} - P_{\geq 2^n+1} \times 2^n + \sum_{k=2}^{2^n} P_{\geq k} \\
&= \sum_{k=1}^{2^n} P_{\geq k} \\
&= \sum_{k=1}^{2^n} \left(\frac{2^n!}{2^{nk}(2^n-k)!}\right) \\
&\approx 1.25 \times 2^{n/2} \\
&\sim 2^{n/2}
\end{aligned} \tag{2.1}$$

The estimation step is carried out in Mathematica computation system.

In the above random mapping state transition, since each state has an equal probability to present as the next state, it is easy to prove by contradiction that, for each bit of any state, two possible values, 0 and 1, must have an equal probability to occur, i.e., each has a probability of 0.5 to occur. This is also true for a random permutation transition.

We use pseudorandom mapping to refer to a function whose input-output behavior statistically looks like that of a random mapping. Note that random permutation is a special random mapping.

# Chapter 3

# Design Criteria, Assessment Methods, and Notations

This chapter outlines the design criteria, describes the assessment methods, and lists the notations used in the thesis.

## 3.1 Design Criteria

High performance and high quality randomness are two common design criteria for all our PRNGs. Although randomness is a fundamental requirement for the design of any PRNG, it has no research values if the PRNG does not show advantages in performance. There are already some PRNGs that can generate high quality pseudorandom numbers, but few can run at high speed. It is relatively easy to design a PRNG that can generate high quality pseudorandom numbers, either from scratch or based on some existing primitives, if high speed is not a requirement. For this reason, we will not put a high priority on either speed or randomness, rather we will try to achieve both.

Generally speaking, non-cryptographic PRNGs run faster than cryptographically secure ones. Among the PRNGs that are publicly available, the fastest non-cryptographic PRNG we know is the 128-bit SFMT, which can generate two bytes in one clock cycle on an Intel Core i3 processor. Most 64-bit PRNGs are not able to run into one clock cycle per byte. No

CSPRNG or stream cipher can run into 2 clock cycles per byte so far except MaD2 [44]. In terms of randomness, CSPRNGs excel. All the CSPRNGs we tested pass the NIST statistical tests, the new Diehard battery of tests, and the TestU01 batteries of tests. On the other hand, among a dozen of non-cryptographic PRNGs we tested, only two, including our MaD0, can clear all the above tests.

A good PRNG should have a long period so that the output sequence will not repeat for a long time. Non-cryptographic PRNGs are often used in high speed simulations, which often require a large amount of pseudorandom numbers. A long period length is essential for such simulations. For a CSPRNG, a short period length makes the generator vulnerable to various cryptographic attacks. Therefore, a long period is a must-have property.

Besides the above common design criteria, some other requirements that should be taken into consideration when designing non-cryptographic PRNGs are high-dimensional equidistribution, quick recovery from biased states, and ease of use and error-proofing.

In principle, those design requirements set for non-cryptographic PRNGs also apply to CSPRNGs. But due to the different design approaches and the different application focuses of CSPRNGs, some requirements are automatically met and some others are taken for granted. In either case those requirements will not be explicitly listed. For example, CSPRNGs should seldom get into biased states and must be error-proofing.

One of the most important design requirements for CSPRNGs is security. Except in rare case (e.g., the *one-time pad* cipher [45]) where security can be theoretically proved, the security of a cryptosystem is usually measured through the following two conditions: (1) whether a brute force attack is feasible and (2) whether there is any known efficient algorithm, defined as polynomial-time algorithm, that can be used to break it. A brute force attack, being the least efficient but the simplest attack, is the last resort of an attacker. A cryptosystem that cannot resist a brute force attack simply has no value. In terms of brute force attack, the strength of a cryptosystem is usually defined as the key size it supports. The 56-bit Data Encryption Standard (DES) [46] was deemed secure for a long time, but reported broken in late 1990s. A 128-bit key is generally recommended now and

some cryptosystems choose to support even longer keys. The minimum strength of our CSPRNGs is set to 256 bits.

For other attacks, the strength of a cryptosystem may vary and should be analyzed on a case-by-case basis. There are many known attacks. Those attacks can be classified as non-practical attacks and practical attacks. A non-practical attack is an attack that can break a cryptosystem at a cost that is lower than that of a brute force attack but is still too high in practice. For example, an attack that can reduce the cost from the designed 128 bits to 120 bits is a non-practical attack, because 120 bits are still too high and do not pose as a real threat to the cryptosystem [47]. Another example is that an attack can break a reduced version of a cryptosystem, but not the full version of it. Practical attacks are those that can actually break a cryptosystem. They are real threats and can immediately render a cryptosystem useless. Our security design goal is to make our CSPRNGs resist most known cryptanalytic attacks.

A CSPRNG should also meet a special requirement, namely, withstanding state compromise extensions. This requirement distinguishes a CSPRNG from a general cryptographic PRNG. For example, a stream cipher is a general cryptographic PRNG, which is not required to withstand state compromise extensions.

## 3.2 Assessment Methods

Randomness is usually empirically tested rather than theoretically proved. A couple of statistical testing tools have been developed, among which are the widely used NIST statistical test suite [48] and the Diehard battery of tests [49]. A more stringent statistical test suite is TestU01 [13]. We use these three testing suites in the randomness testing of our proposed PRNGs. We have also developed some tools for testing avalanche effect, high-dimensional equidistribution, etc.

### 3.2.1 Statistical Testing

#### 3.2.1.1 The NIST Statistical Test Suite

The NIST statistical test suite [48] consists of 15 tests. Each test is formulated to test a specific null hypothesis. A probability value (*P-value*) is computed for each test. It is the probability that a perfect random number generator would have produced a sequence less random than the sequence that was tested, given the kind of non-randomness assessed by the test. Each *P-value* is compared to the level of significance ($\alpha$) of the test, which is the probability that the test will indicate that the sequence is not random when it really is random. If *P-value* $\geq \alpha$, then the null hypothesis is accepted. If *P-value* $< \alpha$, then the null hypothesis is rejected. Typically, $\alpha$ is chosen in the range [0.001, 0.01]. An $\alpha$ of 0.01 indicates that one would expect one sequence out of 100 sequences to be rejected by the test if the sequence was random. For a *P-value* $\geq 0.01$, a sequence would be considered to be random with a confidence of 99%. For a *P-value* < 0.01, a sequence would be considered to be non-random with a confidence of 99%.

Most tests in NIST suite return one *P-value*. Some of them contain multiple sub-tests and generate multiple *P-value*s. The 15 tests are listed below. Technical details of each test can be found in [48].

1. Frequency Test
2. Block Frequency Test
3. Cumulative Sums Test
4. Runs Test
5. Longest Run of Ones Test
6. Rank Test
7. Discrete Fourier Transform (Spectral) Test
8. Non-overlapping Template Matching Test
9. Overlapping Template Matching Test

10. Universal Statistical Test
11. Approximate Entropy Test
12. Random Excursions Test
13. Random Excursions Variant Test
14. Serial Test
15. Linear Complexity Test

The empirical results can be interpreted in different ways. The two approaches adopted by NIST are examining the proportion of sequences that pass a statistical test and checking the distribution of *P-values* for uniformity.

The range of acceptable proportions is determined using the confidence interval defined as

$$\hat{p} \pm 3\sqrt{\frac{\hat{p}(1-\hat{p})}{m}}$$

where $\hat{p} = 1 - \alpha$ and *m* is the sample size. If the proportion falls outside of this interval, then there is evidence that the data is non-random. The confidence interval was calculated using a normal distribution as an approximation to the binomial distribution, which is reasonably accurate for large sample sizes (e.g., $m \geq 1000$). For $\alpha = 0.01$ and $m = 1000$, the confidence interval is $0.99 \pm 0.0094392 \geq 0.9805607$ (i.e., the minimum number of sequences passing each test is approximately 980 for a sample size 1000).

The uniform distribution of *P-values* can be visually checked using a histogram. Uniformity may also be determined via a chi-square goodness-of-fit test on the *P-values* obtained for a test (i.e., a *P-value* of the *P-values*). This is accomplished by computing

$$\chi^2 = \sum_{i=1}^{10} \frac{(F_i - {}^{m}/_{10})^2}{{}^{m}/_{10}}$$

where $F_i$ is the number of *P-values* in sub-interval *i* and *m* is the sample size. A *P-value* is then calculated as *P-value*$_T = igamc({}^{9}/_{2}, {}^{\chi^2}/_{2})$. If *P-value*$_T \geq 0.0001$, then the sequences

can be considered to be uniformly distributed. At least 55 sequences are needed to provide statistically meaningful results.

#### 3.2.1.2 The Diehard Battery of Tests

The Diehard battery of tests [49] were developed by George Marsaglia at Florida State University and first published in 1995. It consists of 15 goodness-of-fit tests, some including several variations. The new Diehard release contains 17 tests, including three "tough" tests [50]. The new Diehard tests are:

1. Birthday Spacings Test (including new "tough" Birthday Spacings Test)
2. GCD Test (new "tough" test)
3. Gorilla Test (new "tough" test)
4. Overlapping 5-Permutation Test
5. Binary Rank Test for 31x31 and 32x32 Matrices
6. Binary Rank Test for 6x8 Matrices
7. Bitstream Test
8. Monkey Tests OPSO (Overlapping-Pairs-Sparse-Occupancy), OQSO (Overlapping-Quadruples-Sparse-Occupancy), DNA
9. Count-the-1's Test on a Stream of Bytes
10. Count-the-1's Test for Specific Bytes
11. Parking Lot Test
12. Minimum Distance Test
13. 3-D Spheres Test
14. Squeeze Test
15. Overlapping Sums Test
16. Up-Down Runs Test
17. Craps Test.

Unlike NIST test suite, the Diehard test suite does not provide specific criteria for determining the success or failure of a test. The interpretation of the testing results are discussed in [51,52]. Most of the tests in Diehard return a *P-value*, which should be uniform on [0,1) if the input file contains truly independent random bits. Those *P-values* are obtained by $p = F(X)$, where $F$ is the assumed distribution of the sample random variable $X$, often normal. But that assumed $F$ is often just an approximation, for which the fit will likely be worst in the tails. Thus the Diehard documentation advises that occasional *P-values* near 0 or 1 do not mean failure; when a bit stream really FAILS BIG, a *P-value* of 0 or 1 to six or more places may appear. As suggested in [51], marginal rejections are of no interest. Instead, of interest are *P-values* that are zero or unity to all decimal places, which is evidence of catastrophic failure. Some of Diehard tests yield a series of *P-values*, in which case a Kolmogorov–Smirnov (KS) test might be performed on those *P-values* to produce a single *"P-value"*. The Diehard tests yield a total of 269 *P-values*, including 7 KS test *P-values* from test 1, 3, 6, 11, 12, 13 and 15.

The C language implementation of Overlapping Sums Test (test 15) is not a faithful interpretation of the author's original Fortran language implementation and none of the (pseudo)random number generators we tested can pass this test (please refer to Marsaglia's response at `http://www.varioustopics.com/cryptography/782655-diehards-overlapping-sum-test.html` for more details). So we exclude this test from our testing.

We evaluate our testing results in two ways: (1) checking the number of *P-values* that are smaller than 0.00001 or larger than 0.99999; (2) assessing the distribution of *P-values* for uniformity in the same way as in NIST.

#### 3.2.1.3 The TestU01 Batteries of Tests

TestU01 [13] is the most comprehensive and stringent statistical test suite that is publicly available so far. It is a software library implemented in the ANSI C language. The TestU01 suite includes the tests from NIST and DIEHARD. It provides general implemen-

tations of the classical statistical tests for RNGs, as well as several others tests proposed in the literature, and some original ones. TestU01 consists of six pre-defined batteries of tests: SmallCrush, Crush, BigCrush, Rabbit, Alphabit, and BlockAlphabit. The first three are for sequences of real numbers in the interval (0, 1) and the other three are for bit sequences. From the perspective of statistical testing, these two types of sequences are closely related and one can be converted into another. The tests can be applied to instances of the generators predefined in the library, or to user-defined generators, or to streams of random numbers produced by any kind of device or stored in files. Any (pseudo)random number generator that implements both the *double (*GetU01) (void *param, void *state)* and *unsigned long (*GetBits) (void *param, void *state)* interfaces defined in TestU01 (see below *unif01_Gen* struct) can use all the 6 pre-defined batteries of tests.

```
typedef struct {
    void *state;
    void *param;
    char *name;
    double (*GetU01)(void *param, void *state);
    unsigned long (*GetBits)(void *param, void *state);
    void (*Write) (void *state);
} unif01_Gen;
```

In the current version, Crush uses approximately $2^{35}$ random numbers and applies 96 statistical tests (it computes a total of 144 test statistics and *P-values*), whereas BigCrush uses approximately $2^{38}$ random numbers and applies 106 tests (it computes 160 test statistics and *P-values*). When invoking the battery Rabbit, Alphabit, and BlockAlphabit, one must specify the number of bits available for each test. Other parameters of each test are chosen automatically as a function of the number of available bits. Rabbit and Alphabit apply 26 and 9 different statistical tests respectively. BlockAlphabit applies the Alphabit battery of tests repeatedly to a generator or a binary file after reordering the bits by blocks of different sizes (with sizes of 1, 2, 4, 8, 16, 32 bits).

The TestU01 batteries of tests consist of 301 statistical tests and return 478 *P-values*. Each *P-value* is interpreted as follows: a value in the interval $[10^{-3}, 1 - 10^{-3}]$ means the test is successful; a value outside the interval $[10^{-10}, 1 - 10^{-10}]$ (i.e., too close to 0 or 1) indicates a clear failure; and a value in between is a suspect value. For a suspect *P-value,* the test can be replicated "independently" with disjoint output sequences from the same generator until either failure becomes obvious or suspicion disappears.

### 3.2.2 Avalanche Testing

Avalanche effect measures the diffusion efficiency of a system, e.g., a hash function or a block cipher. It shows how the output changes when the input is changed slightly. The strict avalanche criterion (SAC) is introduced in [53]. It is satisfied if each output bit changes with a probability of 0.5 whenever a single input bit is complemented. For PRNGs, avalanche effect can reflect the quality of key scheduling and state initialization. The avalanche test is performed as follows:

1. Randomly select a key/seed of size 64 bytes (worst case for diffusion in our PRNGs), denoted as K1.
2. Get the following variants of K1:
   (a) K2 = 1's complement of K1
   (b) K3 = reverse of K2
   (c) K4 = 1's complement of K3
   (d) If K2 and K3 are same (i.e., input is symmetric), then K3 and K4 are not used.
3. For each of the above K1, K2, K3, and K4, flip one bit of it each time (starting from the most significant bit) and compare the output of the flipped version with the output of its unflipped version. This step is referred to as one experiment.
4. Repeat step 1, 2, 3 until the number of flippings, denoted as *N*, reaches a specified number, for example $10^6$. Because the test does not stop until all the bits of an input or its variant are flipped and the key/seed length is 512 bits, the actual number of

flippings will be a multiple of 512. For an expected number $10^6$, the actual number of flippings is *N* = 1000448.

Several files are generated during each avalanche test and we describe two of them here, the avalanche (.ava) file and the report (.rpt) file. The avalanche (.ava) file contains a sequence of values, each being the XOR result of two output values corresponding to a pair of input values (unflipped and flipped). Each bit of the XOR result indicates whether the flipping of one input bit has caused the corresponding output bit to flip or not. This file can be submitted to a statistical testing tool for testing. The report (.rpt) file contains the comprehensive analysis results of the avalanche test. Table 3.1 displays part of a sample report file, which shows how each output bit is affected by the flippings of input bits. The first column *bit* lists each output bit. The second column *#chg* and the third column *probChg* give the number of flippings and the flipping probability of each output bit.

Ideally, for the flipping of each input bit, the flipping of each output bit should be like the flipping of a coin, which follows a binomial distribution with a probability $p = 0.5$ for each of the two possible outcomes. The flippings of the output bits can be approximated using normal distribution for a not too small sample size *N* with mean $\mu = N \times p$ and standard deviation $\sigma = \sqrt{N \times p \times (1-p)}$. According to the central limit theorem, an outcome has a probability of 68.3% to lie within $\mu \pm \sigma$, a probability of 95.4% within $\mu \pm 2\sigma$, and a probability of 99.7% within $\mu \pm 3\sigma$. For each output bit, we check if the number of flippings (value given in the second column *#chg*) falls in the above three ranges. The results are shown in column $\sigma$, $2\sigma$, and $3\sigma$, using a value $n$ ($n = 1, 2, 3$) to indicate which range it falls in. The right half of the table (from the seventh column *#bit* to the ninth column *probChg*) measures the probability that a certain number of bits flipped. It tells the probabilities that one bit is flipped, two bits are flipped, and so on. Ideally this probability distribution should also be normal and the maximum probability should occur at the middle (i.e., half of the bits are flipped).

The results given in Table 3.1 are from a test with the following testing parameters: 64-bit input, 16-bit output, and 3008 flippings. This is purely for demonstration. The actual

Table 3.1: A Sample Report of Avalanche Testing (for all input bits)

| bit | #chg | probChg | $\sigma$ | $2\sigma$ | $3\sigma$ | #bit | #flippings | probChg |
|---|---|---|---|---|---|---|---|---|
| 0 | 1529 | 0.508311 | 1 | - | - | 0 | 0 | 0.000000 |
| 1 | 1530 | 0.508644 | 1 | - | - | 1 | 3 | 0.000997 |
| 2 | 1523 | 0.506316 | 1 | - | - | 2 | 4 | 0.001330 |
| 3 | 1457 | 0.484375 | - | 2 | - | 3 | 34 | 0.011303 |
| 4 | 1487 | 0.494348 | 1 | - | - | 4 | 85 | 0.028258 |
| 5 | 1468 | 0.488032 | - | 2 | - | 5 | 199 | 0.066157 |
| 6 | 1547 | 0.514295 | - | 2 | - | 6 | 367 | 0.122008 |
| 7 | 1505 | 0.500332 | 1 | - | - | 7 | 535 | 0.177859 |
| 8 | 1487 | 0.494348 | 1 | - | - | 8 | 564 | 0.187500 |
| 9 | 1543 | 0.512965 | - | 2 | - | 9 | 539 | 0.179189 |
| 10 | 1485 | 0.493684 | 1 | - | - | 10 | 365 | 0.121343 |
| 11 | 1485 | 0.493684 | 1 | - | - | 11 | 195 | 0.064827 |
| 12 | 1512 | 0.502660 | 1 | - | - | 12 | 85 | 0.028258 |
| 13 | 1458 | 0.484707 | - | 2 | - | 13 | 29 | 0.009641 |
| 14 | 1506 | 0.500665 | 1 | - | - | 14 | 3 | 0.000997 |
| 15 | 1485 | 0.493684 | 1 | - | - | 15 | 1 | 0.000332 |
| ave | 1500.44 | 0.498816 | 1 | - | - | 16 | 0 | 0.000000 |

Input bit(s): all 64 bits
Number of flippings: 3008
Normal distribution:
mean ($\mu$): 1504.000000
std_dev ($\sigma$): 27.422619
$\mu \pm \sigma$ (68.3%): 1477 – 1531
$\mu \pm 2\sigma$ (95.4%): 1449 – 1559
$\mu \pm 3\sigma$ (99.7%): 1422 – 1586

testing parameters we used are much larger (e.g., 512-bit input, 2048-bit output, and $10^6$ flippings) and the generated report file contains many large tables and is far too large to be directly used. For this reason, we plot a set of figures to visually examine the avalanche effect.

### 3.2.3 High-Dimensional Equidistribution

Equidistribution measures the uniformity of a PRNG. If we construct a sequence $\mathbf{u} = \{\mathbf{u}_i | i = 0, 1, 2, \cdots, 2^n - 1\}$ of $k$-dimensional unit vectors from a full period of pseudorandom sequence (where $2^n$ is the period) and equally divide the $k$-dimensional unit hypercube [0, 1]$^k$ into $2^{kw}$ cubic cells, we can measure the $k$-dimensional equidistribution by computing the number of points (out of the total $2^n$ points in $\mathbf{u}$) in each cell. The pseudorandom sequence is said to be (*k*, *w*)-*equidistributed* if each cell contains the same

number of points. The condition $n \geq kw$ is necessary if each cell is going to contain at least one point.

The equidistribution property of linear generators over $F_2$ can be studied from a theoretical perspective without computing random points explicitly. Non-linear generators, on the other hand, lack such mathematical support and are usually evaluated via empirical tests. It is impossible to actually test equidistribution for long period PRNGs due to huge memory requirement. For example, if we want to test whether a PRNG is (35, 32)-*equidistributed*, we need $2^{35\times32} = 2^{1120}$ counters. Due to this reason, no high-dimensional equidistribution testing tools or methods have been published so far. Our approach is to project the *k*-dimensional space onto a subspace, for example a 3-dimensional space, and then check the equidistribution in this subspace. If the PRNG is (*k*, *w*)-*equidistributed*, it should be equidistributed in all those subspaces. Although being equidistributed in all those subspaces is not a sufficient condition for drawing the conclusion that the PRNG is (*k*, *w*)-*equidistributed*, it can be used for likelihood estimation. The equidistribution in each subspace is evaluated by applying a $\chi^2$ test and then computing a *P-value* corresponding to the goodness-of-fit distributional test on the density values. A larger *P-value* means a better fit, indicating better equidistribution.

## 3.3 Notations

All pseudorandom number generation algorithms are described using pseudo code and the following notations are used:

| Notation | Usage |
|---|---|
| # | starting a comment line |
| ++ | increment ($x$++ is the same as $x = x + 1$) |
| % | modulo |
| << | left logical bitwise shift |
| >> | right logical bitwise shift |
| <<< | left bitwise rotation |
| >>> | right bitwise rotation |
| & | bitwise AND |
| \| | bitwise OR |
| ^ | bitwise XOR |
| [ ] | array subscripting (subscript starts from 0) |

Hexadecimal numbers are prefixed by “0x” and all variables and constants are unsigned integers in little endian.

# Chapter 4

# MARC: A Simple Pseudorandom Number Generator

In this chapter, we propose a simple pseudorandom number generator, called Modified Alleged RC4 (MARC). It is a new variant of RC4 stream cipher. MARC enhances the security of RC4 by modifying its key scheduling algorithm and improves the performance by modifying its pseudorandom generation algorithm. MARC retains the simplicity of RC4 and is much faster than RC4.

The rest of this chapter is organized as follows: Section 4.1 presents the algorithm details of MARC. Section 4.2 gives a security analysis of the algorithm. Section 4.3 and 4.4 provide the statistical testing results and performance testing results respectively. Finally section 4.5 summarizes the chapter.

## 4.1 Algorithm Design

In this section we present the algorithm details of MARC. We first give a brief review of RC4 and then describe the algorithm of MARC and the differences between RC4 and MARC.

Listing 4.1: RC4 Algorithm

```
1  ## addition (+) and increment (++) operations ##
2  ## are performed modulo 256                   ##
3
4  # Key Scheduling Algorithm (KSA)
5  for i from 0 to 255
6      S[i] = i
7  endfor
8  j = 0
9  for i from 0 to 255
10     j = j + S[i] + key[i % keylength]
11     swap(S[i], S[j])
12 endfor
13
14 # Pseudorandom Generation Algorithm (PRGA)
15 i = 0
16 j = 0
17 while GeneratingOutput
18     i++
19     j = j + S[i]
20     swap(S[i], S[j])
21     n = S[i] + S[j]
22     output S[n]
23 endwhile
```

### 4.1.1 RC4

RC4 is a stream cipher designed by Ron Rivest in 1987. It was kept as a trade secret of RSA Security until it leaked out in 1994. Because RSA Security has never officially released the algorithm and the name RC4 is trademarked, what is often referred to in the open literature is Alleged RC4 (ARC4) [6]. Although some weaknesses in its key scheduling algorithm have been reported and new faster and claimed secure stream ciphers have been proposed, RC4 is likely to remain as a big player in cryptographic applications.

RC4 is simple and ideal for software implementation. It maintains an internal state, which consists of a permutation of all $2^8 = 256$ possible bytes and two 8-bit indices for accessing elements in the permutation. The permutation is first initialized to the identity permutation and then shuffled using a secret key according to the key scheduling algorithm (KSA) given on lines 5 to 12 of Listing 4.1. Afterwards, pseudorandom number sequence is generated using the pseudorandom generation algorithm (PRGA) given on lines 15 to 23 of Listing 4.1. The PRGA iterates as many times as needed and during each iteration it

continues to shuffle the permutation by swapping two elements and outputs one byte. To encrypt or decrypt a message, RC4 simply acts the pseudorandom number sequence on the message using bitwise exclusive-or (XOR). The key length of RC4 varies and can be up to 256 bytes, typically between 5 and 32 bytes.

### 4.1.2 MARC

The Key Scheduling Algorithm (KSA) and Pseudorandom Generation Algorithm (PRGA) of MARC are shown in Listing 4.2. MARC KSA accepts a key up to 64 bytes and iterates 256 + 256 + 64 = 576 times to shuffle the identity permutation. Note that we use a new 16-bit unsigned integer *r* instead of the 8-bit index *i* as the loop counter in the second *for* loop. This is because we need to iterate 576 times but index *i* can only have 256 different values. MARC KSA introduces a third index *k*, which is initialized to 0 and then XORed with index *j* during each iteration. MARC replaces RC4's swap operation between $S[i]$ and $S[j]$ with a left rotation operation among $S[i]$, $S[j]$, and $S[k]$ (i.e., $tmp = S[i]$, $S[i] = S[j]$, $S[j] = S[k]$, $S[k] = tmp$) during each iteration.

MARC PRGA keeps the values of indices *j* and *k* from KSA and sets the initial value of index *i* to the sum of indices *j* and *k*. It shuffles the permutation and outputs 4 bytes, $S[m]$, $S[n]$, $S[m \wedge j]$, and $S[n \wedge k]$, during each iteration using indices $j$, $k$, $m$, and $n$, where $m$ is the sum of $S[j]$ and $S[k]$, and $n$ is the sum of $S[i]$ and $S[j]$. The combination of $m$, $n$, $m \wedge j$, and $n \wedge k$ renders good statistical properties (see section 4.3).

## 4.2 Properties

### 4.2.1 Pseudorandom Permutation State Transition

The expected period of MARC is largely determined by the transition of the 256-byte state table. The transition effect of the state table can be evaluated like the avalanche effect.

Listing 4.2: MARC Algorithm

```
## addition (+) and increment (++) operations    ##
## are performed modulo 256; except variable r,  ##
## which is a 16-bit unsigned integer, all other ##
## variables are 8-bit unsigned integers.        ##

# Key Scheduling Algorithm (KSA)
for i from 0 to 255
    S[i] = i
endfor
i = 0
j = 0
k = 0
for r from 0 to 575
    j = j + S[i] + key[i % keylength]
    k = k ^ j
    left_rotate(S[i], S[j], S[k])
    i++
endfor

# Pseudorandom Generation Algorithm (PRGA)
# (j and k are from KSA)
i = j + k
while GeneratingOutput
    i++
    j = j + S[i]
    k = k ^ j
    swap(S[i], S[j])after iteration k
    m = S[j] + S[k]
    n = S[i] + S[j]
    output S[m]
    output S[n]
    output S[m ^ j]
    output S[n ^ k]
endwhile
```

We can compare a state with its next state to examine the change of each bit.

Theoretically if the state transition follows a random permutation, then, out of the total *N* possible states, each state has an equal probability to be the next state and, out of the two possible values (0 or 1) for each bit of a state, each value has an equal probability to occur.

Statistically this can be measured in several ways. One way is to measure the probability that each bit is flipped during a transition. In theory this probability should be 0.5. Another way is to measure the number of bits in the whole state table that are flipped during a transition. Half of the bits in the whole state table are expected to be flipped during a transition. In general, these two approaches are not equivalent, because each bit having a probability of 0.5 to be flipped is the sufficient condition, not the necessary condition, for half of the bits in the whole table to be flipped. However, since MARC uses swap operations for state transition and it iterates through all the elements of the state table during each 256 iterations, no state table element is treated differently from others. Hence, each bit having a probability of 0.5 to be flipped must also be the necessary condition for half of the bits in the whole state table to be flipped. As such, we can take any of the two approaches to examine the random permutation feature of the state table.

We design a state transition test as follows. Let $Q_j = \{S[0]_k, S[1]_k, ..., S[255]_k\}$ ($j, k \in \mathbb{N}$) denote one state after iteration $k$. The next state is chosen to be $Q_{j+1} = \{S[0]_{k+256}, S[1]_{k+256}, ..., S[255]_{k+256}\}$, i.e., $Q_{j+1}$ is 256 iterations away from state $Q_j$. This ensures each byte of the state table has at least one chance to be swapped, which is necessary to complete a permutation. Let $f : Q_j \rightarrow Q_{j+1}$ denote the state transition from state $Q_j$ to its next state $Q_{j+1}$. For convenience, we call one such transition an experiment. We apply the chi-square statistic test on the distribution of the Hamming distance between state $Q_j$ and its next state $Q_{j+1}$. The expected distribution for a random permutation transition should be a binomial distribution with a probability of 0.5. We compare the chi-square statistic with the critical value 2199.06 (for our chosen significance level $\alpha = 0.01$) to determine whether to accept or reject the null hypothesis $\mathcal{H}_0$ that the distribution of the Hamming distance of two consecutive states in MARC is close to that of a random permu-

Table 4.1: [MARC] Chi-Square Statistic Testing Results for State Table Transition

| Run | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\chi^2$ | 223.687 | 484.083 | 290.068 | 264.105 | 264.17 | 301.109 | 360.267 | 255.353 | 285.75 | 209.909 |
| Reject $\mathcal{H}_0$? | No | No | No | No | No | No | No | No | No | No |
| Testing parameters:<br>Number of transitions in each run ($N$): 100352<br>Output size: 2047 bits (exclude one bit from the 256-byte state table)<br>Critical value (for $\alpha = 0.01$ and *d.o.f.* = 2047) for rejecting $\mathcal{H}_0$: 2199.06 | | | | | | | | | | |

tation (for more details of chi-square test, please see section 5.2). The testing results for 10 runs are shown in Table 4.1. Each run contains 100352 experiments. The testing results strongly support the null hypothesis $\mathcal{H}_0$.

### 4.2.2 Period

MARC and RC4 have the same state transition algorithm. Their state transition is carried out by swapping two elements of the state table $S$. Therefore their expected periods can be evaluated in the same way.

Theoretical analysis has established that the expected period of RC4 is overwhelmingly likely to be greater than $10^{100}$ [54]. The expected period of RC4 was also analyzed in [55, 56]. A generalized RC4 using $n$-bit words was analyzed in [55], where $n = 8$ is the most commonly used value. It was shown that for $n = 2$ and $n = 3$ RC4 does resemble a random permutation. For $n = 4$, experimental attempts at determining the cycle length were unsuccessful, indicating that short periods are very rare. Their work suggests that with high probability the expected period of RC4 should be sufficiently large, especially for the recommended word size $n = 8$. Our study suggests that the RC4 state transition follows a pseudorandom permutation. Therefore, it is reasonable to estimate its period in the form of a random permutation.

For the 256-byte state table of RC4, the number of possible permutations is $N = 256! \approx 2^{1684}$. This gives an average period around $N/2 = 2^{1683}$ according to subsection 2.2.1. The index $i$ is used as a loop counter and has a cycle of $2^8$. The transition of index $j$ roughly follows a pseudorandom mapping and has a cycle around $2^{8/2} = 2^4$. In summary,

if the transition of the 256-byte state table follows a pseudorandom permutation, then the expected period of RC4 should be around $2^{1683} \times 2^8 \times 2^4 = 2^{1695} \approx 1.76 \times 10^{510}$. In this sense, the lower bound of period given in [54] is rather conservative.

For MARC, there is an additional index *k*, which contributes another factor $2^4$. Therefore the expected period of MARC is around $2^{1699} \approx 2.82 \times 10^{511}$.

### 4.2.3 Security

In this section we discuss how MARC enhances the security of RC4 by modifying its key scheduling algorithm.

RC4 has some weaknesses in its key scheduling algorithm [57–61]. This has been exploited to break the Wired Equivalent Privacy (WEP) encryption used in 802.11 wireless local area networks. A remedy suggested by RSA is to discard the first 256 bytes of the output sequence [62]. IETF RFC 4345 requires that RC4 used in Secure Shell (SSH) Transport Layer Protocol must discard the first 1536 bytes of keystream [63]. This in effect introduces more shuffling and ensures the internal state is well mixed. The KSA of MARC iterates 576 times and each time it rotates three values. The KSA of RC4 only iterates 256 times and each time it swaps two values. If one rotation is computed as 1.5 swaps, then MARC swaps 864 times. This is more than three times of RC4. Note that this is not exactly the same as dropping 864 - 256 = 608 bytes at the beginning since the shuffling during pseudorandom number generation does not involve the key as key scheduling does.

In RC4, the first *keylength* bytes of *S* are likely to have similar patterns after key scheduling if similar keys that only differ at the end are used. In MARC, the additional 256 + 64 = 320 iterations (the maximum *keylength* in MARC is 64 bytes) are used for breaking such patterns. To break the patterns, it is sufficient to add (256 + *keylength*) iterations instead of 320 iterations. But it is unwise to make the number of iterations depend on the key size, because it leaks information about the key size and may be exploited by a side-channel attack. This is the reason we choose to add 320 iterations regardless of the actual key size.

For similar keys that only differ at the end, the differences do not come into play before

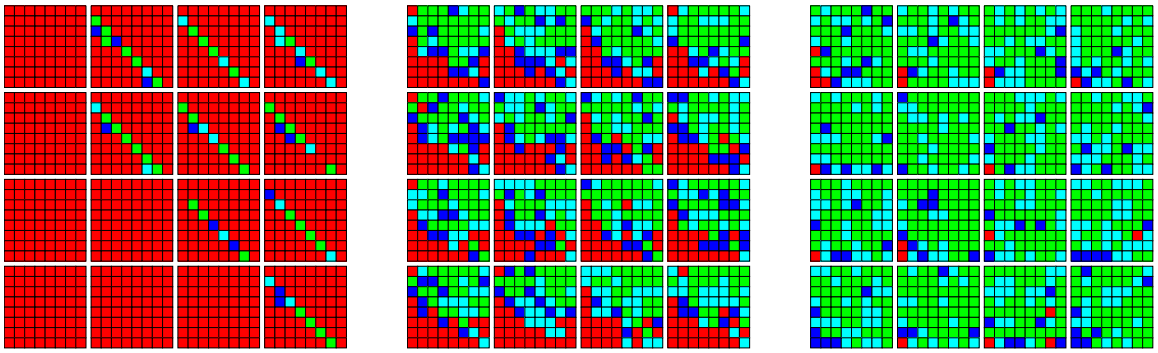
the *keylength*-th (*keylength* $\leq 64$) iteration of key scheduling. This means the first *keylength* bytes are very likely to have a similar distribution for those keys. MARC deals with this by shuffling the first 64 bytes one more time than the other part.

At the end of key scheduling, MARC sets the value of index $i$ to the sum of indices $j$ and $k$ and persists all three indices for future use. Because the output sequence is highly sensitive to the values of the indices $i$, $j$, and $k$, making indices depend on the secret key instead resetting them to 0 at the beginning of the PRGA helps reduce correlations among similar keys.

The internal state of RC4 evolves relatively slowly and what matters more is the change of $n$ if a short sequence (e.g., a few bytes) is to be generated. In this case, it makes no big difference whether one generates 4 bytes using 4 different values of index $n$ or using 4 different indices, as long as the 4 values of $n$ and the 4 different indices are both computed in a proper way. In RC4, $n = S[i] + S[j]$. $S[i]$ iterates through the state table. $S[j]$ is more irregular and unpredictable. In MARC, the new $m = S[j] + S[k]$ (is actually $S[i] + S[k]$ before swap operation) is computed in a similar way as $n$. The computation of the other two values, $m\text{^}j = (S[i] + S[k])\text{^}j$ (using the $S[i]$ value before swapping) and $n\text{^}k = (S[i] + S[j])\text{^}k$, involves all three indices and mixes two different levels of data, i.e., the indices and the value of state elements located through the indices. These two levels of data are bitwise XORed together. XOR is linear in $\mathbb{F}_2$, but cannot be handled using pure linear algebra in the residue class ring $\mathbb{Z}/2^n\mathbb{Z}$.

### 4.2.4 Avalanche Property

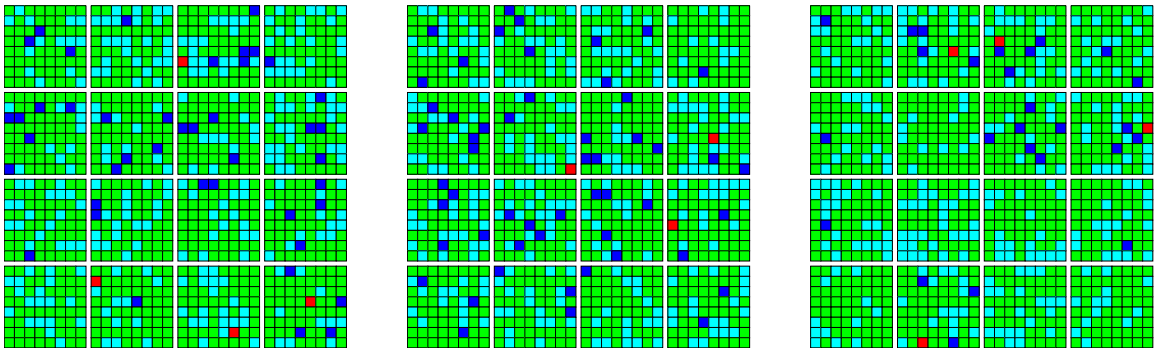
To measure the differences between MARC KSA and RC4 KSA, we test their avalanche effects as described in subsection 3.2.2. For key scheduling, the input is the key and the output is the 256 byte state table after key scheduling. The flippings of output bits are checked against a normal distribution with mean $\mu = N \times p$ and standard deviation $\sigma = \sqrt{N \times p \times (1-p)}$, where $p = 0.5$ is the ideal flipping probability and $N = 1000448$ is the sample size. For each output bit, we check if the number of flippings falls within $\mu \pm n\sigma$.



(a) RC4 (output offset = 0)

(b) RC4 (output offset = 32 bytes)

(c) RC4 (output offset = 64 bytes)



(d) MARC (output offset = 0)

(e) MARC (output offset = 32 bytes)

(f) MARC (output offset = 64 bytes)

Figure 4.1: Avalanche Effect - Comparison of RC4 KSA and MARC KSA

The corresponding output bit is given a value $n$ ($n = 1, 2, 3$) if the number of flippings falls within $n$ standard deviation(s); it is assigned a value 4 otherwise. These four different values, from the best 1 to the worst 4, are depicted using different colors: green (■) for 1, cyan (■) for 2, blue (■) for 3, and red (■) for 4.

The testing results for both RC4 KSA and MARC KSA are illustrated in Figure 4.1. Each small square plotted in one of the four different colors shows how well the flipping of an input bit (row) causes the flipping of an output bit (column). The input bit range is [1, 32]. The three output bit ranges are [1, 32], [257, 288], and [513, 544] respectively. RC4 KSA has a really poor avalanche effect at the beginning of the internal state (Figure 4.1 (a)). The testing revealed that the flipping probability is far below the ideal value 0.5 for those

bits at the beginning. The situation improves as we move the output bit range away from the beginning of the internal state (Figure 4.1 (b)). The avalanche effect is significantly better when the output bit range is shifted out of the first *keylength* (here 64) bytes of the internal state (Figure 4.1 (c)). MARC KSA has a better avalanche effect than RC4 KSA in all three cases.

## 4.3 Statistical Testing

In this section we present the statistical testing results of MARC using the NIST statistical test suite, the new Diehard battery of tests, and the TestU01 batteries of tests.

### 4.3.1 NIST Statistical Test Suite

#### 4.3.1.1 Testing Setup

We tested and compared the pseudorandom number generation of RC4 and MARC. For each of them, we tested 1000 pseudorandom sequences, each containing one million bits (125 KB). The significance level ($\alpha$) is set to 0.01 in all tests. For each sequence, a random key is generated and used to initialize the generator. This random key can be up to 64 bytes and is generated from an enhanced RC4 PRNG (discarding the first 512 bytes of the pseudorandom output) using the computer clock value as its key.

#### 4.3.1.2 Testing Results

The testing results are shown in Table 4.2. The number of *P-values* (NoP) returned by each test is listed in the second column. The *P-value*$_T$ assesses the uniformity of the distribution of *P-values* returned by each test and a value equal to or larger than 0.0001 means the distribution can be considered uniform. The proportion of sequences (PoS) passing a test is given in another column. For 1000 sequences, a minimum value of 980 is required to pass the test. For a few tests, some sequences may be invalid, in which case NIST test suite gives out the number of sequences actually used, e.g., $\frac{587}{592}$ means 592

Table 4.2: [MARC] Statistical Testing Results (NIST)

| Test | NoP | RC4 | | MARC | |
|---|---|---|---|---|---|
| | | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS |
| 1 | 1 | 0.161703 | 991 | 0.534146 | 989 |
| 2 | 1 | 0.146982 | 998 | 0.733899 | 994 |
| 3 | 2 | 0.572847 | 989 | 0.769527 | 986 |
| | | 0.928857 | 989 | 0.632955 | 988 |
| 4 | 1 | 0.229559 | 984 | 0.777265 | 993 |
| 5 | 1 | 0.595549 | 991 | 0.552383 | 987 |
| 6 | 1 | 0.134172 | 992 | 0.566688 | 991 |
| 7 | 1 | 0.274341 | 988 | 0.163513 | 985 |
| 8 | 148 | 0.470224 | 989 | 0.518298 | 989 |
| 9 | 1 | 0.404728 | 992 | 0.532132 | 988 |
| 10 | 1 | 0.029401 | 988 | 0.862883 | 988 |
| 11 | 1 | 0.676615 | 990 | 0.302657 | 987 |
| 12 | 8 | 0.299100 | 587/592 | 0.570090 | 615/621 |
| 13 | 18 | 0.417622 | 585/592 | 0.673667 | 614/621 |
| 14 | 2 | 0.530120 | 991 | 0.599693 | 986 |
| | | 0.576961 | 989 | 0.245490 | 986 |
| 15 | 1 | 0.771469 | 991 | 0.077607 | 986 |

Table 4.3: [MARC] Statistical Testing Results (Diehard)

| Test | NoP | RC4 | | MARC | |
|---|---|---|---|---|---|
| | | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p |
| 1 | 11 | 5.3e-10 | 0 | 5.4e-006 | 0 |
| 2 | 2 | 0.816537 | 0 | 0.304126 | 0 |
| 3 | 33 | 0.170722 | 0 | 0.479073 | 0 |
| 4 | 5 | 0.534146 | 0 | 0.204439 | 0 |
| 5 | 2 | 0.024356 | 0 | 0.699313 | 0 |
| 6 | 26 | 0.711915 | 0 | 0.052326 | 0 |
| 7 | 20 | 0.100709 | 0 | 0.271619 | 0 |
| 8 | 82 | 0.339928 | 1 | 0.198214 | 0 |
| 9 | 1 | 0.289667 | 0 | 0.514124 | 0 |
| 10 | 25 | 0.815812 | 0 | 0.838472 | 1 |
| 11 | 11 | 0.492614 | 0 | 0.297118 | 0 |
| 12 | 11 | 0.415748 | 0 | 0.404407 | 0 |
| 13 | 21 | 0.587668 | 0 | 0.202783 | 0 |
| 14 | 1 | 0.494392 | 0 | 0.350485 | 0 |
| 16 | 3 | 0.334538 | 0 | 0.540878 | 0 |
| 17 | 4 | 0.875539 | 0 | 0.847183 | 0 |

sequences are used and 587 of them pass the test. The testing results show, both MARC and RC4 pass the NIST statistical tests.

### 4.3.2 Diehard Battery of Tests

#### 4.3.2.1 Testing Setup

We divide Diehard tests into two groups based on the minimum random sequence size that is needed by each test. GCD, Gorilla, and Overlapping Permutations (test 2, 3, 4) need a much longer random sequence than other tests and are put in a group. All other tests are put in another group. For GCD, Gorilla, and Overlapping Permutations, we tested 50 random sequences for each random number generator, each containing 2176 million bits (272 MB). For other tests, we tested 100 random sequences for each random number

generator, each containing 96 million bits (12 MB). Using this setup, at least 100 *P-values* are generated for each test.

#### 4.3.2.2 Testing Results

The Diehard testing results are given in Table 4.3. The values given in "x-p" column are numbers of extreme *P-values* (i.e., *P-values* smaller than 0.00001 or larger than 0.99999). Both generators passed all the tests except for the *P-value*$_T$ of the first test, Birthday Spacings Test. To find the problem, we tested another two (pseudo)random number generators: SHA1 (running in counter mode) and QRNG (a quantum random number generator available at `http://qrng.physik.hu-berlin.de/`). Both of them failed this part too, which suggests it could be a problem with the test itself or the interpretation of the testing results. For this reason, we will not draw any conclusion from this part.

### 4.3.3 TestU01 Batteries of Tests

#### 4.3.3.1 Testing Setup

TestU01 requires much more (pseudo)random numbers than the NIST and Diehard suites. Each generator is implemented in C programming language and created as a *unif01_Gen* object. Both *GetU01()* and *GetBits()* interfaces are implemented for each generator so that all 6 batteries of tests can be applied. Built-in parameters are used for SmallCrush, Crush, and BigCrush. For Rabbit, Alphabit, and BlockAlphabit, the size of bit sequence is set to $32 \times 10^9$.

#### 4.3.3.2 Testing Results

The testing results from TestU01 batteries of tests are given in Table 4.4. The 6 batteries of tests return 478 *P-values*. A *P-value* in the interval $[10^{-3}, 1 - 10^{-3}]$ means the test is successful; a value outside the interval $[10^{-10}, 1 - 10^{-10}]$ indicates a failure; and a value in between is a suspect value. For a suspect *P-value*, we repeat the test 5 times. If no failures

Table 4.4: [MARC] Statistical Testing Results (TestU01)

| Battery | Parameters | Tests | NoP | Failures | |
|---|---|---|---|---|---|
| | | | | RC4 | MARC |
| SmallCrush | Built-in | 10 | 15 | 0 | 0 |
| Crush | Built-in | 96 | 144 | 0 | 0 |
| BigCrush | Built-in | 106 | 160 | 0 | 0 |
| Rabbit | $32 \times 10^9$ bits | 26 | 40 | 0 | 0 |
| Alphabit | $32 \times 10^9$ bits | 9 | 17 | 0 | 0 |
| BlockAlphabit | $32 \times 10^9$ bits | $6 \times 9$ | 102 | 0 | 0 |

or suspect *P-values* are observed during the retries, we clear the test; otherwise we mark the test as failed. Both RC4 and MARC passed all the tests.

## 4.4 Performance Testing

The speed testing results for MARC are given in Table 4.5. The testing results for RC4 and the four software-efficient finalists of eStream project are also included in the table for comparison. The testing is performed for a software implementation using C programming language. The C implementation closely follows the pseudo code given in section 4.1. There are no special optimizations done at the source code level except that register variables are used to minimize memory access whenever possible. Most modern compilers are smart enough and know more about code generation than the developer [64]. They can perform various optimizations to generate more compact and/or faster code, including constant folding, dead code elimination, inline expansion or macro expansion, strength reduction, loop optimization, code re-ordering for maximum pipeline throughput and cache effects, and many more. Therefore we leave optimizations largely to the compiler.

Both 32-bit and 64-bit executables compiled using Microsoft Visual C/C++ Optimizing Compiler Version 16 with option /O2 (optimized for maximum speed) are tested. RC4 and eStream finalists are not 64-bit algorithms, but their 64-bit executables run slightly faster on our machine (Intel Core i3 370M, 2.4GHz, 64 KB L1 data cache, 64 KB L1 instruction

Table 4.5: [MARC] Pseudorandom Number Generation Speed (cycle/byte)

| Generator | Sequence size (KB) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 5 | 10 | 100 | 1000 | 10000 |
| RC4 | 9.53 | 7.67 | 7.09 | 6.98 | 7.04 | 7.04 |
| MARC | 17.46 | 6.60 | 5.21 | 3.98 | 3.89 | 3.86 |
| HC-128 | 55.21 | 13.27 | 7.96 | 3.58 | 3.15 | 3.11 |
| Rabbit | 12.20 | 10.06 | 9.63 | 9.51 | 9.52 | 9.49 |
| Salsa20 | 8.94 | 8.95 | 8.95 | 8.89 | 8.90 | 8.88 |
| Sosemanuk | 48.67 | 13.48 | 9.70 | 5.79 | 5.61 | 5.36 |

cache, 512 KB L2 cache) and are chosen for the testing.

For each sequence size, we run each executable 30 times and get the average value of the top 3 speeds. The reason we exclude low speeds in our calculation is that the measured cycles may contain contributions from some system processes that we cannot stop and the small cycles more likely reflect the actual performance. MARC outperforms all generators except HC-128.

## 4.5 Conclusion

In this chapter we have presented a variant of RC4 called MARC. It enhances the key scheduling algorithm of RC4 and resists known attacks that exploit the weakness in RC4's key scheduling. It also improves the pseudorandom generation algorithm of RC4 and has a better performance than RC4 and most eStream finalists. It passed all the NIST statistical tests, the new Diehard battery of tests, and the TestU01 batteries of tests.

# Chapter 5

# MARC-bb: MARC as a Building Block

MARC is a fast standalone PRNG. As a byte-oriented PRNG, however, MARC does not take the advantages of modern 64-bit platforms. For this reason, MARC is mainly used as a building block to construct more advanced PRNGs in this thesis. In this chapter, we introduce an iteration-reduced version of MARC, referred to as MARC-bb (bb stands for building block) hereafter, which will be used in the key scheduling, state initialization, and in some cases reseeding of our other PRNGs. In MARC-bb, the total number of iterations in the key scheduling is reduced from 576 to 320. This is because we either do not need that much shuffling for non-cryptographic PRNGs or we introduce additional shuffling after the key scheduling for cryptographically secure PRNGs.

As a key scheduling and state initialization component, it is essential that MARC-bb diffuse the key bits fast enough. In other words, it should have an excellent avalanche effect. This chapter is solely concerned with the study of the avalanche property of MARC-bb. We first give an analytical analysis in section 5.1. Then in section 5.2 we perform a chi-square goodness-of-fit test. Finally in section 5.3, we further test and analyze the avalanche effect using the tool described in subsection 3.2.2.

## 5.1 Analytical Analysis

For key scheduling, the input is the key and the output is the 256-byte state table, denoted as $K$ and $S$ respectively here. What we want to know is how the $q$-th byte of $K$, denoted as $K[q]$, affects the $r$-th byte of $S$, denoted as $S[r]$, where $0 \leq q < keylength$ and $0 \leq r < 256$. For convenience, we will use $L_k$ instead of $keylength$ to represent the key length in the following analysis. In RC4, the value of $S[r]$ is affected by the swap operation between $S[i]$ and $S[j]$, where $i$ is the loop counter and $j$ is a function of $i$ and $K$ as shown below:

$$j(i,\ K) = \sum_{t=0}^{i} (S[t] + K[t\ \%\ L_k])$$

For each swap operation, the value of $S[r]$ will be replaced by another value if $r$ equals $i$ or $j$ but not both. In the special case that $r$, $i$, and $j$ have the same value, $S[r]$ will be replaced by itself. In either case, we say that the value of $S[r]$ has been affected (but not necessarily changed).

In the case that $r$ equals $i$, the swap is performed between $S[r]$ and $S[j]$, where $j = j(r,\ K) = \sum_{t=0}^{r} (S[t] + K[t\ \%\ L_k])$. If $q$ is larger than $r$, then $K[q]$ is not included in the computation of $j(r,\ K)$ and therefore has no effect on this swap operation. Since there is exactly one time that $r$ will be equal to $i$ during the key scheduling, the number of times or the probability that the value of $S[r]$ will be affected by $K[q]$ under the condition that $r$ equals $i$ can be computed as:

$$\begin{aligned} P_1 &= \sum_{i=0}^{255} Prob(q \rightarrow r | r = i) \\ &= \begin{cases} 0 & (q > r) \\ 1 & (q \leq r) \end{cases} \end{aligned} \tag{5.1}$$

where $q \rightarrow r$ reads that $K[q]$ affects $S[r]$.

In the case that $r$ equals $j$, the swap is performed between $S[i]$ and $S[r]$, where $r = j(i,\, K) = \sum_{t=0}^{i} (S[t] + K[t \,\%\, L_k])$. If $q$ is larger than $i$, then $K[q]$ is not included in the computation of $r = j(i,\, K)$ and therefore has no effect on the swap operation. So the number of times that the value of $S[r]$ will be affected by $K[q]$ under the condition that $r$ equals $j$ is:

$$\begin{aligned} P_2 &= \sum_{i=0}^{255} Prob(q \rightarrow r | r = j) \\ &= \sum_{i=q}^{255} Prob(q \rightarrow r | r = j) \\ &= \sum_{i=q}^{255} \frac{1}{256} \\ &= 1 - \frac{q}{256} \end{aligned} \tag{5.2}$$

The number of times that the value of $S[r]$ will be affected by $K[q]$ under the condition that $r$, $i$, and $j$ are identical is:

$$\begin{aligned} P_{12} &= \sum_{i=0}^{255} Prob(q \rightarrow r | r = i = j) \\ &= \sum_{i=q}^{255} Prob(q \rightarrow r | r = i = j) \\ &= \sum_{i=q}^{255} \left(\frac{1}{256}\right)^2 \\ &= \frac{1}{256} - \frac{q}{256^2} \end{aligned} \tag{5.3}$$

We can estimate the number of times that the value of $S[r]$ will be affected by $K[q]$ by adding $P_1$ and $P_2$ and then subtracting $P_{12}$ (it is duplicated and included in both $P_1$ and $P_2$) from the sum, that is,

$$
\begin{aligned}
N(q,\, r)_{RC4} &= P_1 + P_2 - P_{12} \\
&= \begin{cases} 1 - \frac{q+1}{256} + \frac{q}{256^2} & (q > r) \\ 2 - \frac{q+1}{256} + \frac{q}{256^2} & (q \leq r) \end{cases} \\
&= \begin{cases} \frac{255}{256^2}(256 - q) & (q > r) \\ 1 + \frac{255}{256^2}(256 - q) & (q \leq r) \end{cases}
\end{aligned} \tag{5.4}
$$

For MARC-bb KSA, the value of $S[r]$ is affected via the left rotation operation among $S[i]$ , $S[j]$ and $S[k]$. The formula similar to (5.1) is:

$$
\begin{aligned}
P_1 &= \sum_{i=0}^{255} Prob(q \rightarrow r | r = i) + \sum_{i=0}^{63} Prob(q \rightarrow r | r = i) \\
&= \begin{cases} 2 & (q \leq r \leq 63) \\ 1 & (\text{otherwise}) \end{cases}
\end{aligned} \tag{5.5}
$$

Here the value of $P_1$ can exceed 1 and thus it is more proper to view $P_1$ as the number of times instead of probability that the value of $S[r]$ will be affected by $K[q]$. The second summation in (5.5) comes from the additional 320 - 256 = 64 iterations added to the key scheduling. During these 64 iterations, all the bytes of the key affect each of the first 64 bytes of $S$, hence eliminating the case $P1 = 0$ for $q > r$ in (5.1). In addition, for $q \leq r \leq 63$, $S[r]$ will be affected by $K[q]$ twice, one time in each summation in (5.5). In summary, $P_1$ takes value 1 or 2, as shown in (5.5).

Due to the introduction of the third index $k$ and the use of rotation operations in MARC-bb KSA, the calculation of $P_2$ needs to take into account both the case $r = j$ and $r = k$ as follows:

$$
\begin{aligned}
P_2 &= \sum_{i=0}^{255} (Prob(q \to r | r = j \text{ or } r = k) - Prob(q \to r | r = j = k)) \\
&\quad + \sum_{i=0}^{63} (Prob(q \to r | r = j \text{ or } r = k) - Prob(q \to r | r = j = k)) \\
&= \sum_{i=q}^{255} (Prob(q \to r | r = j \text{ or } r = k) - Prob(q \to r | r = j = k)) \\
&\quad + \sum_{i=0}^{63} (Prob(q \to r | r = j \text{ or } r = k) - Prob(q \to r | r = j = k)) \\
&= \sum_{i=q}^{255} \left( \frac{2}{256} - \frac{1}{256^2} \right) + \sum_{i=0}^{63} \left( \frac{2}{256} - \frac{1}{256^2} \right) \\
&= \frac{511}{256^2}(320 - q) \qquad (5.6)
\end{aligned}
$$

Similarly, $P_{12}$ can be calculated as:

$$
\begin{aligned}
P_{12} &= \sum_{i=0}^{255} (Prob(q \to r | r = i = j \text{ or } r = i = k) - Prob(q \to r | r = i = j = k)) \\
&\quad + \sum_{i=0}^{63} (Prob(q \to r | r = i = j \text{ or } r = i = k) - Prob(q \to r | r = i = j = k)) \\
&= \sum_{i=q}^{255} (Prob(q \to r | r = i = j \text{ or } r = i = k) - Prob(q \to r | r = i = j = k)) \\
&\quad + \sum_{i=0}^{63} (Prob(q \to r | r = i = j \text{ or } r = i = k) - Prob(q \to r | r = i = j = k)) \\
&= \sum_{i=q}^{255} \left( \frac{2}{256^2} - \frac{1}{256^3} \right) + \sum_{i=0}^{63} \left( \frac{2}{256^2} - \frac{1}{256^3} \right) \\
&= \frac{511}{256^3}(320 - q) \qquad (5.7)
\end{aligned}
$$

And finally we have

$$
\begin{aligned}
N(q,\, r)_{MARC-bb} &= P_1 + P_2 - P_{12} \\
&= \begin{cases} 2 + \frac{511\times 255}{256^3}(320 - q) & (q \le r \le 63) \\ 1 + \frac{511\times 255}{256^3}(320 - q) & (\text{otherwise}) \end{cases}
\end{aligned} \tag{5.8}
$$

Both $N(q,\, r)_{RC4}$ and $N(q,\, r)_{MARC-bb}$ are plotted in Figure 5.1, which clearly shows that each byte of the key has a better chance to affect each byte of the state in MARC-bb than in RC4. How many times should a byte of the key affect a byte of the state in terms of avalanche effect? Each time $K[q]$ affects $S[r]$, the value of $S[r]$ will be replaced by a value, which can be any of the 256 possible values including the old value of $S[r]$. If each of the 256 possible values has the same chance to be the new value of $S[r]$, then on average each bit of $S[r]$ has an ideal 50% chance to be flipped during one value replacement. Additional value replacements cannot further improve the avalanche effect under this perfect condition. But neither RC4 nor MARC-bb can guarantee that each of the 256 possible values will have the same chance to be the new value during each value replacement. So additional value replacements do help improve the avalanche effect. This is one reason that MARC-bb has better avalanche effect than RC4. More importantly, $N(q,\, r)_{MARC-bb}$ is always larger than 1 but $N(q,\, r)_{RC4}$ can be less than 1. This difference is significant, since a value less than 1 means that sometimes $K[q]$ might not affect $S[r]$ at all. Since $N(q,\, r)_{RC4} < 1$ occurs for $q > r$, RC4 is expected to have rather poor avalanche effect at the beginning part of the state.

## 5.2 Chi-Square Statistic Test

In this section we conduct a chi-square goodness-of-fit test to assess the avalanche property of MARC-bb KSA. The testing results for RC4 and hash function MD5, SHA1,

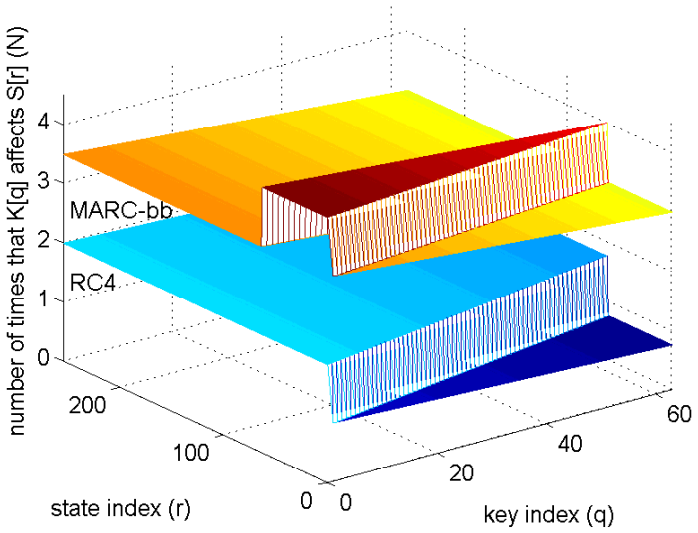



Figure 5.1: [MARC-bb] Impact of Key on Internal State in Key Scheduling

and SHA256 are also provided for comparison. For MARC-bb and RC4, the input is the key/seed and the output is the internal state after key scheduling. For hash function MD5, SHA1, and SHA256, the input is the input message and the output is the corresponding hash value. Our goal is to accept or reject the null hypothesis $\mathcal{H}_0$ that, for the flippings of input bits, the flippings of output bits follow a binomial distribution.

Let $K$ represent the 64-byte input and $S = (s_0, s_1, ..., s_{L-1})$ represent the output, where $L$ is the bit length of the output. For the flipping of each input bit, we calculate the Hamming distance $H$ between the flipped version $S'$ and the unflipped version $S$ as:

$$H = \sum_{i=0}^{L-1} (s_i^{'} \wedge s_i)$$

Let $H_j$ denote the Hamming distance corresponding to the $j$-th experiment (i.e., the $j$-th flipping of input bits). The number of times that exactly $m$ output bits are flipped is

$$Count_m = \sum_{j=1}^{N} H_j \times \delta_{mH_j} \quad m = 0, 1, 2, ..., L$$

Table 5.1: [MARC-bb] Chi-Square Statistic Testing Results for Key Scheduling

| Algorithm | Input size (bytes) | Output size (bytes) | *d.o.f.* | *C.V.* ($\alpha = 0.01$) | $\chi^2$ | Reject $\mathcal{H}_0$? |
|---|---|---|---|---|---|---|
| MD5 | 64 | 16 | 128 | 168.233 | 49.527 | No |
| SHA1 | | 20 | 160 | 204.633 | 66.401 | No |
| SHA256 | | 32 | 256 | 311.674 | 77.629 | No |
| MARC-bb | | 32 | 256 | 311.674 | 79.46 | No |
| | | 256* | 2047 | 2199.06 | 238.36 | No |
| RC4 | | 256* | 2047 | 2199.06 | $4.56 \times 10^{55}$ | Yes |
| RC4 (+64 iterations) | | 256* | 2047 | 2199.06 | $1.87 \times 10^{16}$ | Yes |
| RC4 (+256 iterations) | | 256* | 2047 | 2199.06 | 244.29 | No |
| * We use 2047 bits out of the total 2048 output bits, because for swap operations the number of flippings is always an even number if the whole state table is used. | | | | | | |

where *N* is the number of experiments and $\delta_{mH_j}$ is the Kronecker delta [65]. Then we compute the chi-square value

$$\chi^2 = \sum_{m=0}^{L} \frac{(Count_m - NP_m)^2}{NP_m} \tag{5.9}$$

where $P_m$ is the theoretical probability that *m* output bits are flipped for a binomial distribution, calculated as

$$P_m = \begin{pmatrix} L \\ m \end{pmatrix} \times \frac{1}{2^L} = \frac{L!}{m!(L-m)!} \times \frac{1}{2^L}$$

The critical value of the chi-square distribution with the degrees of freedom (*d.o.f.*), *v,* can be computed as (see [17]):

$$C.V. = v + \sqrt{2v}x_p + \frac{2}{3}x_p^2 - \frac{2}{3} + O(\frac{1}{\sqrt{v}}) \tag{5.10}$$

where $x_p = 2.33$ for our chosen significance level $\alpha = 0.01$. Finally we compare the chi-square statistic value computed from (5.9) with the critical value given in (5.10) to determine whether to accept or reject the null hypothesis $\mathcal{H}_0$.

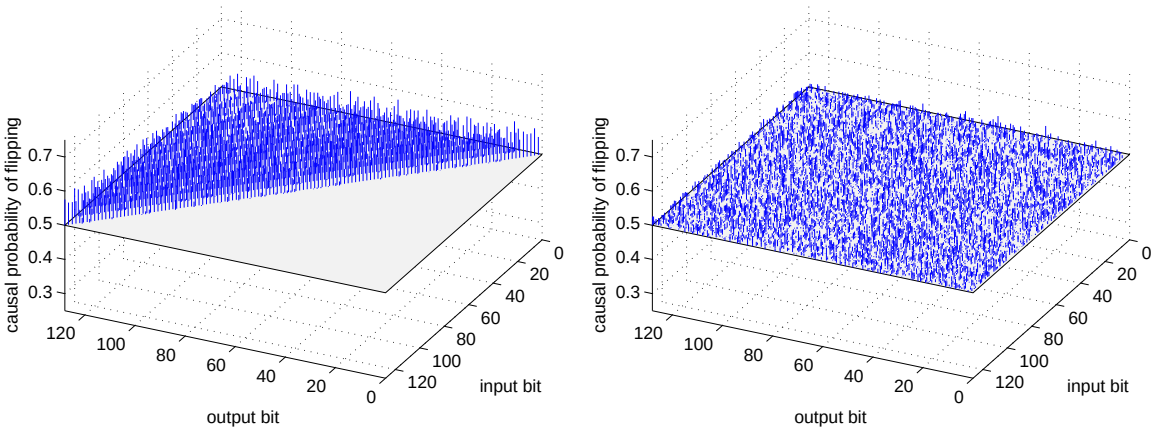
The chi-square testing results are shown in Table 5.1. Each $\chi^2$ value is the average re-

sult of 10 runs, with each run containing $N = 100352$ experiments. The results show that MARC-bb KSA has a similar avalanche effect as hash function MD5, SHA1, and SHA256. It satisfies the strict avalanche criterion. Each $\chi^2$ value is far less than the corresponding critical value at the 0.01 significance level, suggesting that the null hypothesis cannot be rejected and the observed distribution has a close match with the expected binomial distribution. The distributions of the original RC4 and revised RC4 with additional 64 iterations do not agree with the expected binomial distribution. But more shuffling, e.g., with additional 256 iterations, helps to improve the avalanche effect of RC4 KSA.

## 5.3 Statistical Analysis

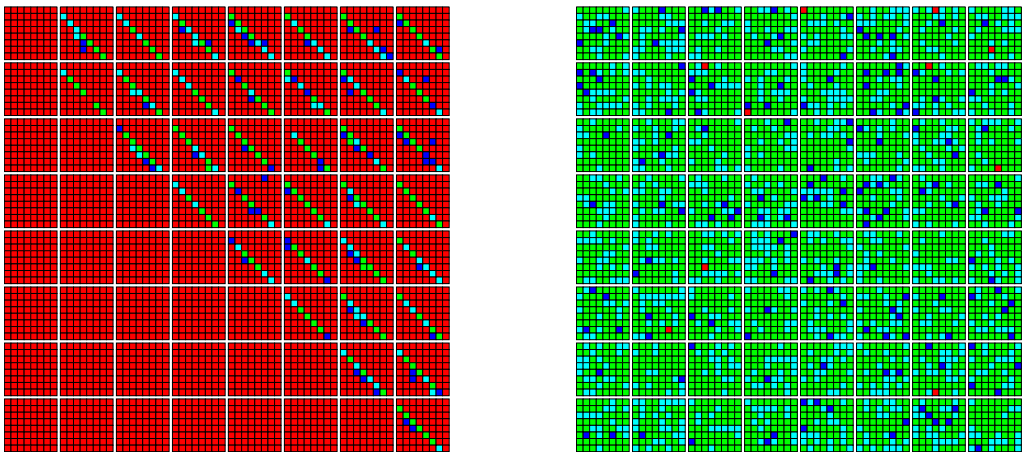
In this section we present and analyze some statistical testing results, obtained through the steps described in subsection 3.2.2. Figures 5.2 (a) and (b) show the probabilities that the flipping of each input bit causes the flipping of each output bit for RC4 and MARC-bb respectively. Figure 5.2 (a) reveals that the *i*-th byte of the key does not have enough impact on the *j*-th ($j < i$) byte of the internal state. This is also the main cause of the problems of RC4 key scheduling reflected in other figures. Because of this, the overall causal probability of flipping is much smaller than the ideal value 0.5 in RC4. Note that the causal probabilities corresponding to the empty triangle area in Figure 5.2 (a) are not 0. They are less than 0.5 (below the 0.5 probability plane) and not shown in the figure. Figures 5.2 (c) and (d) show the causal probabilities in a different format, which we have explained in section 4.2.

Figures 5.3 (a), (b), (c), and (d) illustrate the flipping probabilities of each output bit. Different from Figures 5.2 (a), (b), (c), and (d), which are three-dimensional (input bit/output bit/causal probability), they are two-dimensional (output bit/probability of flipping). The first two figures depict the average probability of flipping of each output bit of RC4 and MARC-bb respectively. To show the deviation from the expected probability 0.5, a line is also drawn from 0.5 to the probability value plotted using x-mark. The last two fig-




(a) RC4

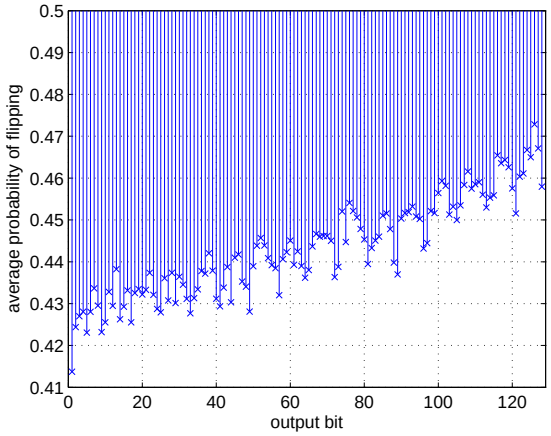
(b) MARC-bb



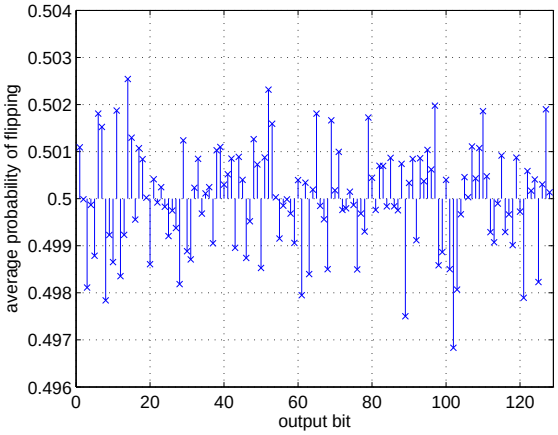
(c) RC4

(d) MARC-bb

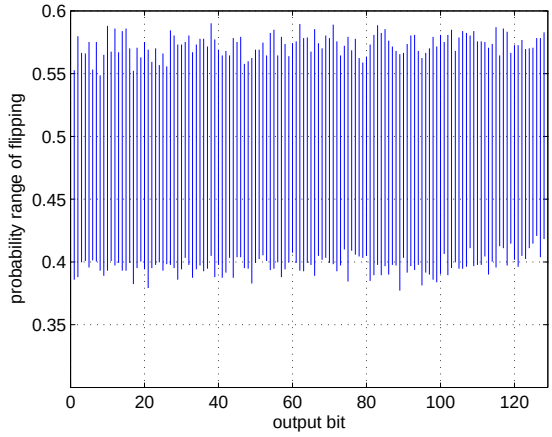
Figure 5.2: [MARC-bb] Avalanche Effect – Causal Probability of Flipping (output offset = 0 bytes)



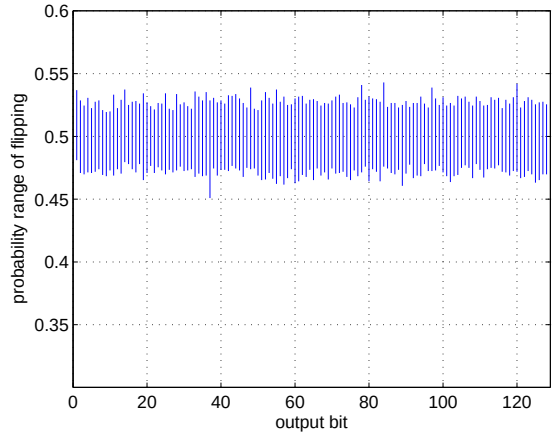

(a) RC4 (average probability)




(b) MARC-bb (average probability)

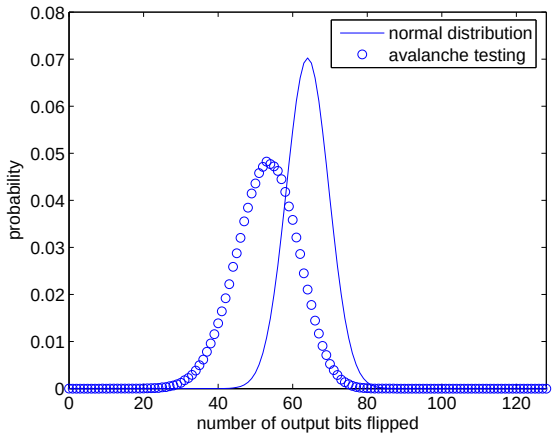



(c) RC4 (probability range)

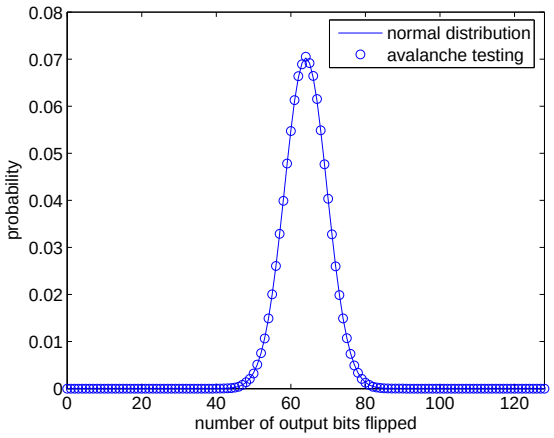



(d) MARC-bb (probability range)

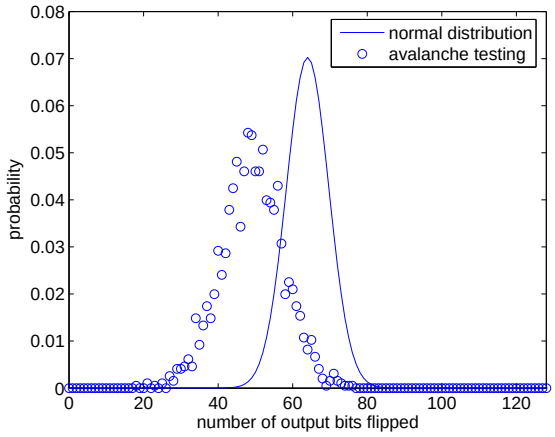
Figure 5.3: [MARC-bb] Avalanche Effect – Probability of Flipping (output offset = 0 bytes)



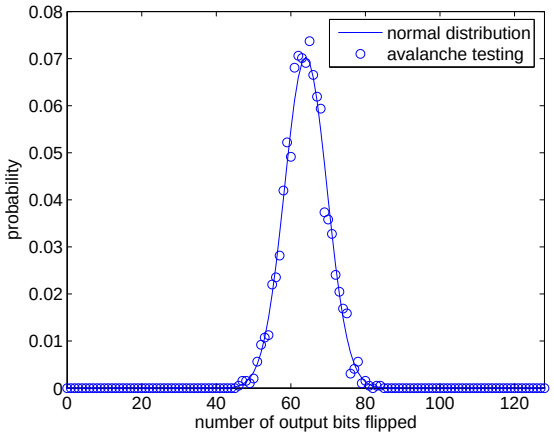

(a) RC4 (for all input bits)




(b) MARC-bb (for all input bits)




(c) RC4 (for last input bit)




(d) MARC-bb (for last input bit)

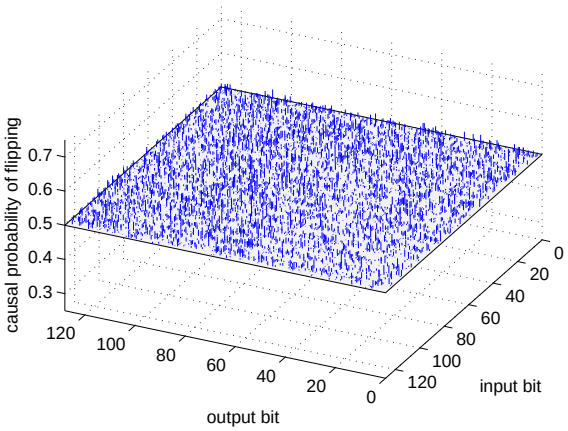
Figure 5.4: [MARC-bb] Avalanche Effect – Probability Distribution (output offset = 0 bytes)

ures show the range of flipping probabilities of each output bit. MARC-bb has much better probability distributions than RC4 in both cases.

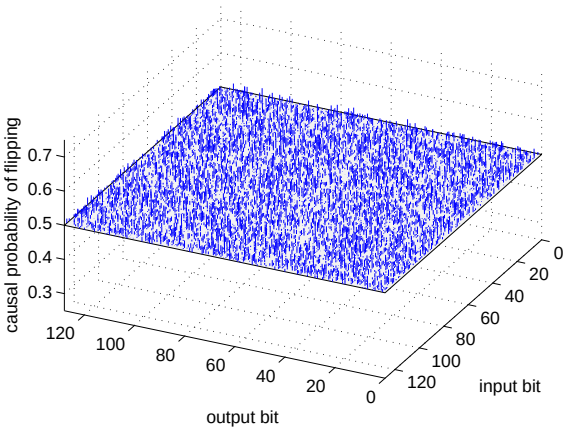
So far we have examined the flipping probabilities for individual output bits. We also want to know the probability that a certain number of output bits are flipped (like in the chi-square statistic test), e.g., the probability of the extreme case that all output bits (or none of them) are flipped. Figures 5.4 (a) and (b) compare the actual probability distribution with the expected normal distribution for RC4 and MARC-bb respectively. The probability distribution from RC4 is far from the expected distribution, while the one from MARC-bb has an almost perfect match with the expected distribution. Figures 5.4 (c) and (d) are for last input bit rather than for all input bits. In this case the sample size is much smaller and as a result both distributions are not as good as their corresponding ones for all input bits, but MARC-bb still performs much better than RC4.

As we mentioned above, in RC4 the first *keylength* bytes of $S$ are likely to have similar patterns after key scheduling if similar keys that only differ at the end are used. This is clearly reflected in our avalanche testing. After the first *keylength* bytes, the situation begins to improve, since any differences between two keys will come into play and affect the shuffling. Figures 5.5, 5.6, and 5.7 show the avalanche testing results for the part of the internal state starting from the 65th byte. RC4 has an obvious improvement in this part, although is still worse than that of MARC-bb.

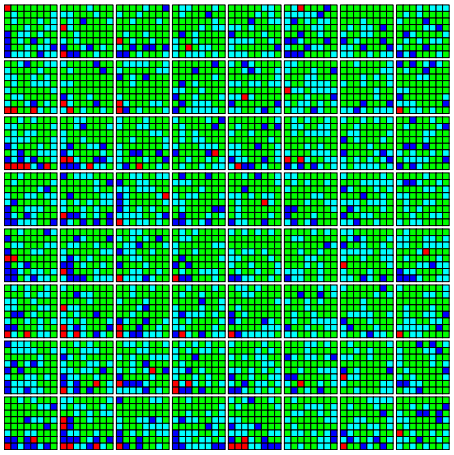
We have also visually compared the avalanche effect of MARC-bb KSA with those of the three popular hash functions: MD5, SHA1, and SHA256. The results are given in Figure 5.8. In general all four functions perform well.



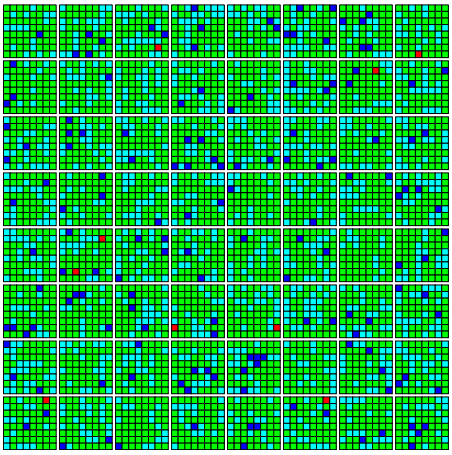

(a) RC4




(b) MARC-bb

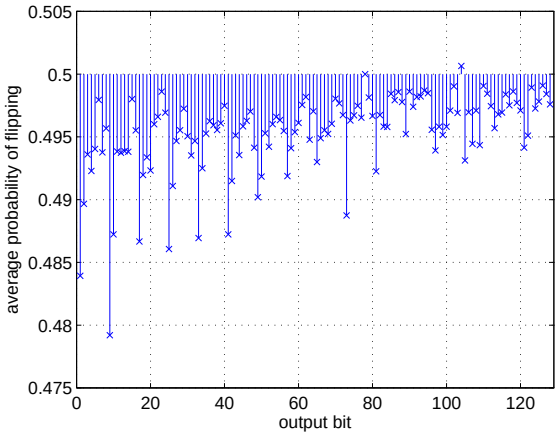


(c) RC4

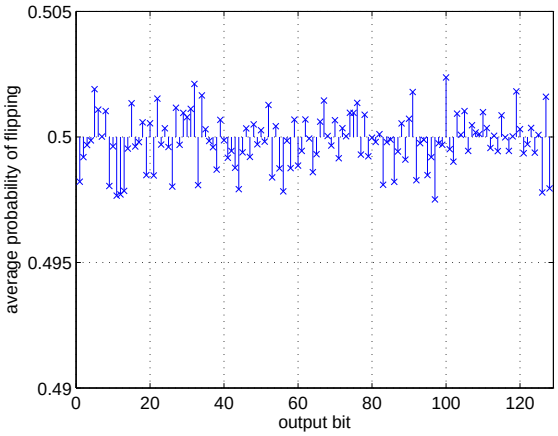


(d) MARC-bb

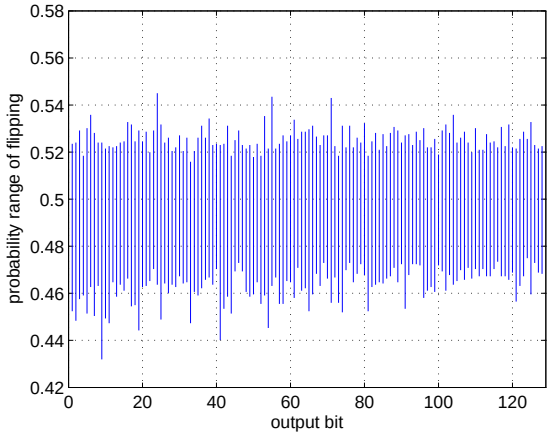
Figure 5.5: [MARC-bb] Avalanche Effect – Causal Probability of Flipping (output offset = 64 bytes)



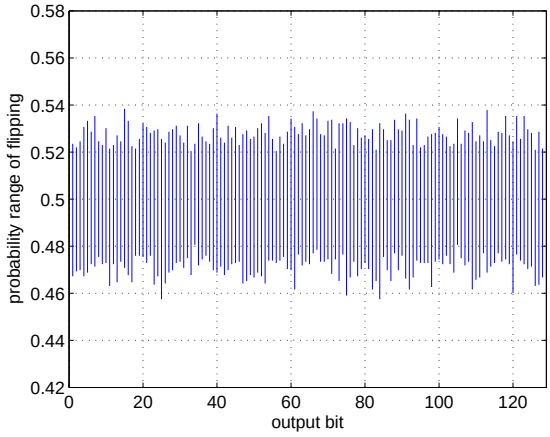

(a) RC4 (average probability)




(b) MARC-bb (average probability)

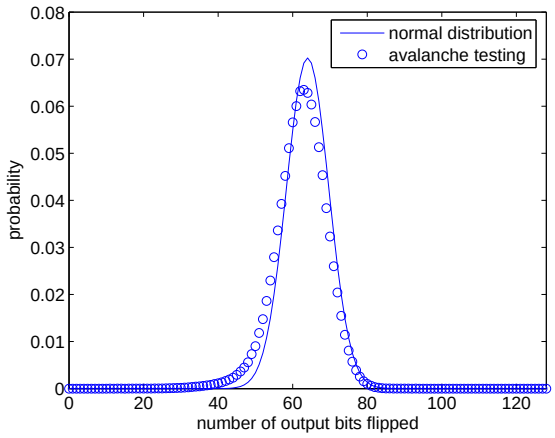



(c) RC4 (probability range)

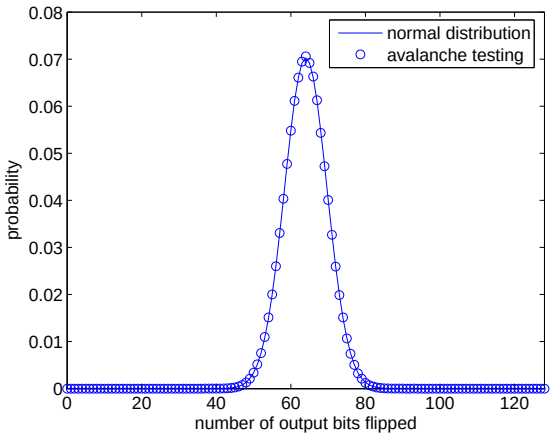



(d) MARC-bb (probability range)

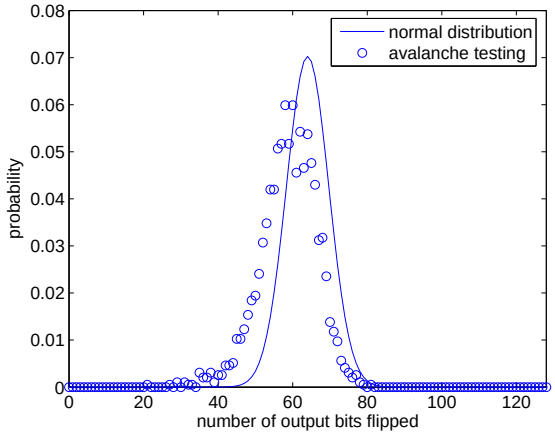
Figure 5.6: [MARC-bb] Avalanche Effect – Probability of Flipping (output offset = 64 bytes)



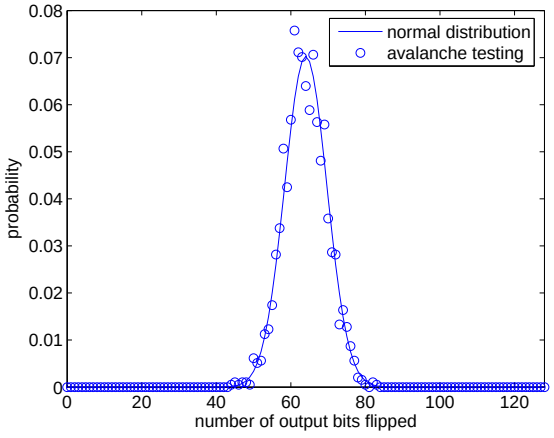

(a) RC4 (for all input bits)




(b) MARC-bb (for all input bits)

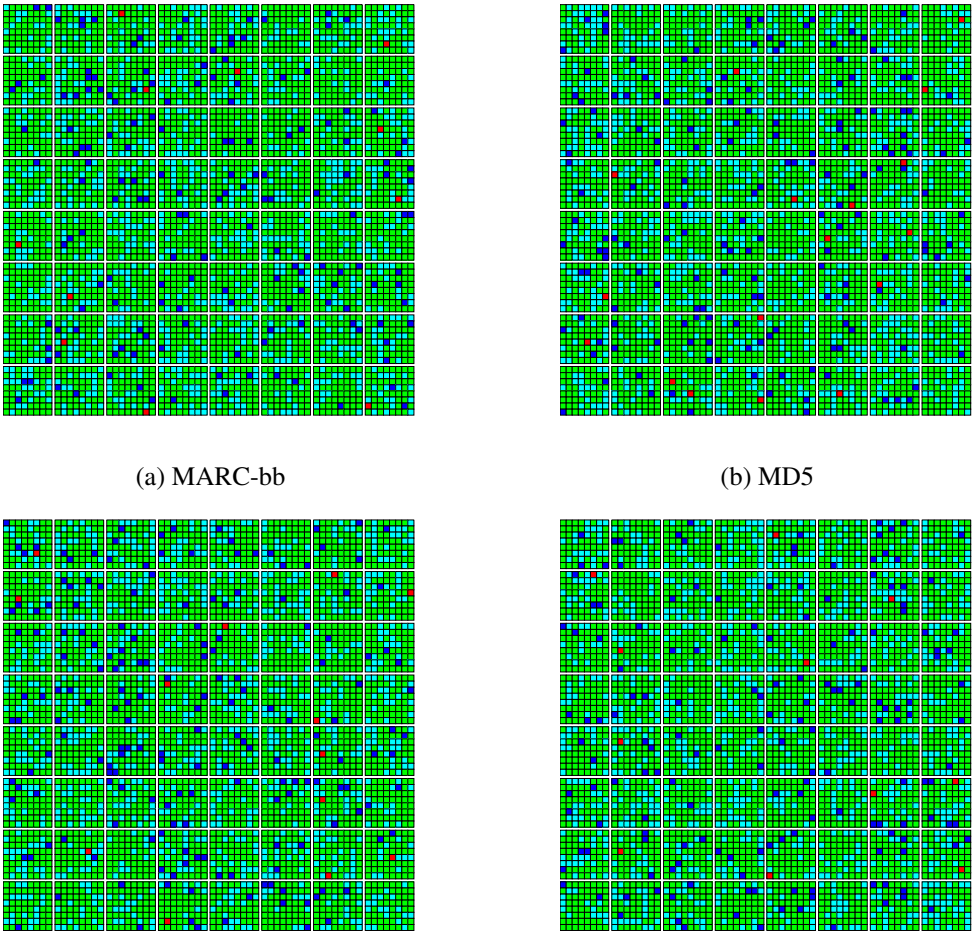



(c) RC4 (for last input bit)




(d) MARC-bb (for last input bit)

Figure 5.7: [MARC-bb] Avalanche Effect – Probability Distribution (output offset = 64 bytes)



(a) MARC-bb

(b) MD5

(c) SHA1

(d) SHA256

Figure 5.8: [MARC-bb] Avalanche Effect - Comparison of MARC-bb, MD5, SHA1, and SHA256

# Chapter 6

# MaD0: An Ultrafast High Quality Non-Cryptographic Pseudorandom Number Generator

In this chapter we present a high speed non-cryptographic PRNG called MaD0. It includes MARC-bb for key scheduling and state initialization, and utilizes fast pseudorandom mappings for state transition and pseudorandom number generation. MaD0 can generate pseudorandom numbers at a speed of half clock cycle per byte on an Intel Core i3 2.4GHz personal computer. It clears all the NIST statistical tests, the new Diehard battery of tests, and the TestU01 batteries of tests. MaD0 also demonstrates other good properties such as long period, high dimensional equidistribution, quick recovery from biased states, and ease of use and error-proofing.

The remainder of this chapter is organized as follows. The algorithm details of MaD0 are presented in section 6.1 and its properties are discussed in section 6.2. The statistical testing results and performance testing results are given in section 6.3 and 6.4 respectively. The chapter concludes in section 6.5.

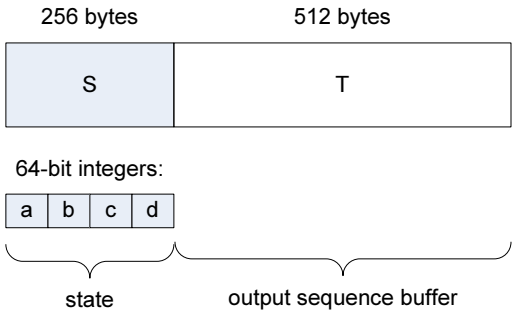



Figure 6.1: [MaD0] Data Structure

## 6.1 Algorithm Design

In this section we present the algorithm details of MaD0, including data structure, functional model, key scheduling and state initialization, and pseudorandom number generation.

### 6.1.1 Data Structure

MaD0 maintains a data structure shown in Figure 6.1. It comprises a 256-byte state table (denoted as $S$), four 64-bit integers (denoted as $a$, $b$, $c$, and $d$), and a 512-byte output sequence buffer (denoted as $T$). State table $S$ and integers $a$, $b$, $c$, $d$ construct the internal state of MaD0. $T$ is used for buffering generated pseudorandom numbers.

### 6.1.2 Functional Model

The functional model of MaD0 is given in Figure 6.2. MaD0 uses MARC-bb KSA to initialize state table $S$ and uses MARC-bb PRGA to initialize integers $a$, $b$, $c$, and $d$. The pseudorandom generation algorithm is logically divided into two functions: the *update function* for updating the internal state and the *generate function* for generating pseudorandom numbers. The details of this model are discussed in the following subsections.

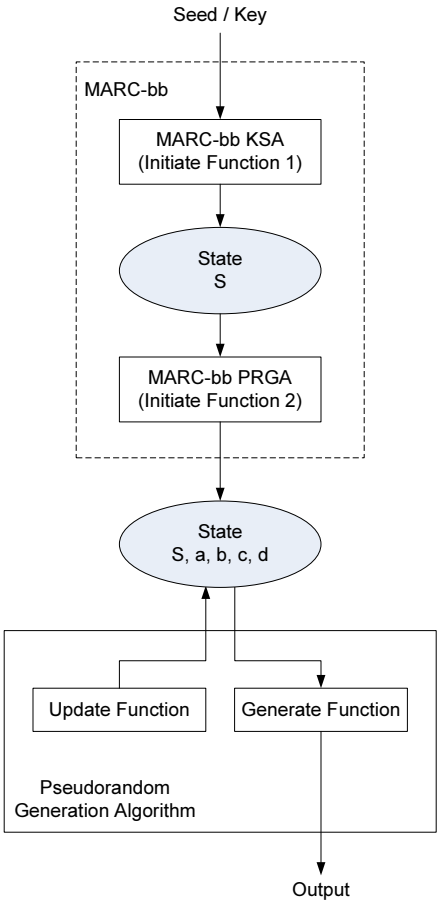



Figure 6.2: [MaD0] Functional Model

### 6.1.3 Key Scheduling and State Initialization

MaD0 accepts a seed/key that is no larger than 64 bytes (512 bits). MaD0 uses MARC-bb to initialize state table $S$ and four integers $a$, $b$, $c$, and $d$. The initialization includes two steps:

1. First, state table $S$ and indices $i$, $j$, and $k$ are initialized after running MARC-bb KSA once a seed/key is provided.

2. 32 pseudorandom bytes are generated by running MARC-bb PRGA. These 32 bytes are converted into four 64-bit integers using little endian and assigned to integers $a$, $b$, $c$, and $d$.

Listing 6.1: [MaD0] One Round of Pseudorandom Number Generation

```
# additions are performed modulo
# 0x10000000000000000;
ta = a = a + c
tb = b = b + d
for i = 0 to 31
    T[2i] = c = c ^ (S[i] + a)
    c = c + (ta ^ tb)
    d = d ^ (c + b)
    ta = ta <<< 3
    T[2i+1] = d = d + (ta ^ tb)
    S[i] = d
    tb = tb >>> 5
endfor
```

After initialization the internal state contains one permutation of {0, 1, ..., 255} and four initialized 64-bit integers.

### 6.1.4 Pseudorandom Number Generation

MaD0 takes advantages of modern 64-bit platforms and uses 64-bit operations to generate pseudorandom numbers. State table $S$ and output buffer $T$ are cast into and used as 64-bit integer arrays during pseudorandom number generation. The output sequence buffer $T$ is marked as empty after state initialization. When a pseudorandom number generation request is received, the buffer is checked. If it is not empty, the data stored in it are used to serve the need. After all the data in the buffer are consumed, MaD0 generates pseudorandom numbers and refreshes the buffer according to the pseudorandom generation algorithm shown in Listing 6.1. Each generation round consists of 32 iterations. In each iteration, two 64-bit integers are generated and an element of state table $S$ is updated. So total 512 bytes are generated and the whole state table $S$ is updated in each generation round.

## 6.2 Properties

We describe some properties of MaD0 in this section, including recurrence relation, pseudorandom mapping state transition, period length, equidistribution, recovery from bi-

ased states, and ease of use and error-proofing. The statistical property and performance of MaD0 will be addressed separately in the next two sections.

To see where MaD0 stands, we will also compare it with other PRNGs where appropriate. We pick up PRNGs for comparison from the following three categories: LCG generators, Xorshift-based generators, and MT-flavored generators, specifically:

- LCG ($m$, $a$, $c$)
  - LCG ($2^{31}$, 1103515245, 12345) – the LCG used in ANSI C, now ISO C, and will be denoted as LCG / ISO C in this thesis. It outputs high order bits 30..16 of the integer computed during each recurrence.
  - LCG ($2^{48}$, 25214903917, 11) – the LCG used in Java's package java.util.Random and will be denoted as LCG / Java hereafter. It outputs high order bits 47..16 of the integer computed during each recurrence.
  - LCG ($2^{64}$, 6364136223846793005, 1) – the LCG used in Newlib, a C standard library implementation intended for use on embedded systems, and will be denoted as LCG / Newlib. It outputs high order bits 63..32 of the integer computed during each recurrence.
  - LCG ($2^{64}$, 3935559000370003845, odd) – one of the 64-bit LCGs that have good lattice structure according to L'Ecuyer [12]. It will be referred to as LCG / L1. The value of $c$ can be any odd number and we simply choose 1 for it. We output high order bits 63..32 of the integer computed during each recurrence as in LCG / Newlib.
  - LCG ($2^{64}$, 2685821657736338717, 0) – another pick from L'Ecuyer's table [12]. It will be referred to as LCG / L2. We output high order bits 63..32 of the integer computed during each recurrence as in LCG / Newlib. Since $c = 0$, it is a multiplicative LCG (MLCG) and runs faster than LCG / L1.
- Xorshift

  - Xorshift – We choose one of Marsaglia's "favorite" 64-bit Xorshift generators, the Xorshift with (a, b, c) = (13, 7, 17) [1].
  - Xorshift7 – the Xorshift generator given in [25]. It uses 7 instead of 3 xorshift operations.
  - Xorgens – We use the 64-bit generator included in the package Xorgens305, which is available at `http://wwwmaths.anu.edu.au/~brent/random.html`.

- MT

  - MT19937ar – This is MT19937 with the addition of an initialization function that admits an array of arbitrary length as a seed. We choose the optimized implementation from Cokus and Bellew, which is available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`. It will be referred to as MT for simplicity.
  - WELL512 – the 512-bit WELL. We take the optimized implementation from Lomont [66], which is about 40% faster than the one released by the authors of WELL [3]. It will be referred to as WELL.
  - SFMT19937 – It is chosen to match MT19937ar. The implementation version 1.4 from the authors (available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html`) is used. It will be referred to as SFMT.

### 6.2.1 Recurrence Relation

MaD0 generates pseudorandom number sequence through some recurrence relation. Each term of the sequence is defined as a function of the preceding term(s).

Assume the initial values at generation round $j$ ($j \geq 1$) are as follows:

$$a = A_j$$
$$b = B_j$$
$$c = C_j$$
$$d = D_j$$

then $ta = a = a + c$ and $tb = b = b + d$ give

$$ta = a = A_j + C_j$$
$$tb = b = B_j + D_j$$

The recurrence relation for generation round *j* is

$$T[0]_j = c \wedge (S[0]_{j-1} + a)$$
$$S[0]_j = T[1]_j = d \wedge (T[0]_j + l(0)_j \wedge r(0)_j + b) + l(1)_j \wedge r(0)_j$$
$$T[2i]_j = (T[2i-2]_j + l(i-1)_j \wedge r(i-1)_j) \wedge (S[i]_{j-1} + a)$$
$$S[i]_j = T[2i+1]_j = T[2i-1]_j \wedge (T[2i]_j + l(i)_j \wedge r(i)_j + b) + l(i+1)_j \wedge r(i)_j$$

where

$i = 1, 2, ..., 31$, all additions are performed modulo $2^{64}$;

$l(i)_j = ta$ <<< $3i$, i.e., left bitwise rotate $ta$ by $3 \times i$ bits;

$r(i)_j = tb$ >>> $5i$, i.e., right bitwise rotate $tb$ by $5 \times i$ bits.

At the end of generation round *j*, *c* and *d* become

$$c = T[62]_j + l(31) \wedge r(31) \ = C_{j+1}$$
$$d = S[31]_j = T[63]_j = D_{j+1}$$

At the beginning of generation round *j+1*, *a*, *b*, *ta*, and *tb* are first updated through $ta = a = a + c$ and $tb = b = b + d$ again. Thus

$$ta = a = A_j + C_j + C_{j+1}$$
$$tb = b = B_j + D_j + D_{j+1}$$

Then it repeats the same process as generation round *j*.

The data flow of MaD0 in one iteration is shown in Figure 6.3, where $xor(l, r)_1 = l(i-1) \wedge r(i-1)$, $xor(l, r)_2 = l(i) \wedge r(i)$, and $xor(l, r)_3 = l(i+1) \wedge r(i)$.

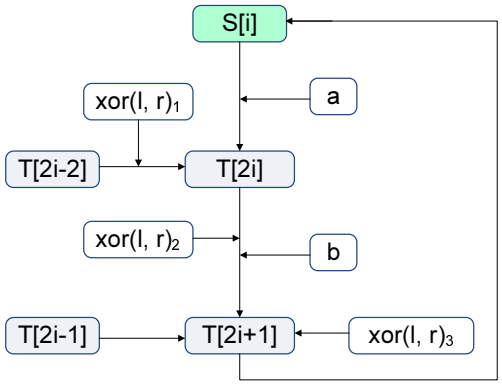



Figure 6.3: [MaD0] Data Flow in One Iteration of PRGA

### 6.2.2 Pseudorandom Mapping State Transition

The internal state of MaD0 comprises state table $S$ and four integers $a$, $b$, $c$, and $d$. Let $Q_j = \{S[0]_j, S[1]_j, ..., S[31]_j, a_j, b_j, c_j, d_j\}$ denote the state after generation round $j$ ($j = 1, 2, 3, ...$) and $Q_j \rightarrow Q_{j+1}$ denote the state transition from generation round $j$ to generation round $j$+1. As shown in subsection 6.2.1, MaD0 does not have a simple recurrence relation that lasts for the whole sequence as in many known PRNGs. The parameters are changed in each generation round. At the beginning of each generation round, integers $a$ and $b$ are updated through $c$ and $d$ respectively and then further assigned to two temporary variables *ta* and *tb*. During each generation round, integers $c$ and $d$ are updated through *ta*, *tb*, and $S[i]$ in each iteration; $S[i]$ itself is also updated in the $i$-th iteration. The combination of left bitwise rotation and right bitwise rotation in each iteration introduces different random values into the state unless *ta* and *tb* are all 1's or all 0's.

We can examine state transition after each iteration of a generation round or after a whole generation round. As we will explain, these two approaches are equivalent, but it is more convenient to examine state transition after a whole generation round since all elements in the state are updated after each generation round. The importance of state transition is that it determines the period of the generator. To evaluate the period of a generator, we need to know when a state repeats itself. If we examine the state transition after each iteration, we conclude that a state repeats itself when we see two states, after the

same iteration of two different generation rounds, are the same. Note that, if two states are the same, but in different iterations of two generation rounds, then the state is not repeated. If we examine state transition after a whole generation round, we actually examine the state transition after the last iteration of each generation round. When a state repeats itself after any iteration, it must also repeat itself after the last iteration. Therefore the two approaches are equivalent and should give us the same result when used for period evaluation. For a clear view of the state transition, we will take the second approach.

We are interested in knowing whether the state transition $Q_j \rightarrow Q_{j+1}$ follows a random mapping. We cannot theoretically prove our state transition follows random mappings. But our design and empirical testing suggest so.

The internal state shows a very good randomness feature. We separately output state table *S*, individual integers *a*, *b*, *c*, or *d* to generate five different sequences and apply randomness tests on each sequence. Each of them passes all the NIST, Diehard, and TestU01 tests, with or without filtering [1].

Random mapping can also be evaluated using the state transition test, like the one we performed for random permutation in subsection 4.2.1. In the state $Q_j = \{S[0]_j, S[1]_j, ..., S[31]_j, a_j, b_j, c_j, d_j\}$ after generation round *j*, integers *a*, *b*, *c*, and *d* are updated 32 times more frequently than state table *S*. Therefore we ignore these four integers and only concentrate on the state table *S*. The state is chosen after each generation round since all elements in the table are updated after one round. We choose the entire state table as output without excluding any bit. Table 6.1 shows the testing results for state table transition. Clearly, the testing results indicate the transition of the state table follows a pseudorandom mapping.

### 6.2.3 Period

For an *n*-bit internal state, the maximum possible period length is $2^n$. Depending on the state transition function, the actual period length may be much shorter. For a random

[1] Various filterings are performed, including only taking the least significant *n* bytes (*n* = 1, 2, 4), most significant *n* bytes, and *m* (*m* = 2, 4) middle bytes of each 64-bit integer.

Table 6.1: [MaD0] Chi-Square Statistic Testing Results for State Table Transition

| Run | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\chi^2$ | 222.092 | 202.259 | 245.927 | 203.364 | 221.283 | 248.937 | 164.793 | 188.304 | 271.104 | 276.716 |
| Reject $\mathcal{H}_0$? | No | No | No | No | No | No | No | No | No | No |
| Testing parameters:<br>Number of transitions in each run (*N*): 100352<br>Output size: 2048 bits (the 256-byte state table)<br>Critical value (for $\alpha = 0.01$ and *d.o.f.* = 2048) for rejecting $\mathcal{H}_0$: 2200.09 | | | | | | | | | | |

mapping, the average period is around $2^{n/2}$ (please see subsection 2.2.2).

MaD0 has a 2240-bit internal state ($S$ plus $a$, $b$, and $c$)[2]. Assume its state transition function follows a random mapping, then the expected period is about $2^{1120} \approx 1.42 \times 10^{337}$.

A period of $2^{1120}$ is more than enough for any application we can envision at the moment, although longer periods are provided in some other PRNGs. The period of MaD0 can be easily extended if such a need arises. For example, we can double the size of state table $S$ to extend the period to $2^{2144}$. The change has virtually no impact on the performance or the quality of the pseudorandom numbers according to our testing.

Aside from the average period length, another metric of practical interest is the lower bound of the period length. Some stream ciphers and pseudorandom number generators provide a hard lower bound of period length through the use of a counter [8,67,68]. MaD0 does not use a counter and theoretically any period length is possible, but it is unlikely to hit a short period length in practice due to the huge number of internal states. For an $n$-bit state, the probability that the period length is equal to or smaller than $k$ can be computed as

[2] $d$ has the same value as $S[31]$ at the end of a generation round and is thus excluded in the calculation.

$$\begin{aligned}
P_{\leq k} &= 1 - P_{>k} \\
&= 1 - \left(\frac{2^n - 1}{2^n}\right)\left(\frac{2^n - 2}{2^n}\right)\cdots\left(\frac{2^n - k}{2^n}\right) \\
&< 1 - \left[\left(\frac{2^n - k}{2^n}\right)\right]^k \\
&= 1 - \left(1 + \frac{-k}{2^n}\right)^k \\
&= 1 - \sum_{j=0}^{k} \left( \begin{array}{c} k \\ j \end{array} \right) \left(\frac{-k}{2^n}\right)^j
\end{aligned}$$

where

$$\left( \begin{array}{c} k \\ j \end{array} \right) = \frac{k!}{j!(k-j)!}$$

is the binomial coefficient. For $1 \ll k \ll 2^{n/2}$, it follows that

$$\begin{aligned}
P_{\leq k} &< 1 - \sum_{j=0}^{k} \left( \begin{array}{c} k \\ j \end{array} \right) \left(\frac{-k}{2^n}\right)^j \\
&= 1 - \left[1 + \left( \begin{array}{c} k \\ 1 \end{array} \right) \left(\frac{-k}{2^n}\right) + \left( \begin{array}{c} k \\ 2 \end{array} \right) \left(\frac{-k}{2^n}\right)^2 + \cdots\right] \\
&< \left( \begin{array}{c} k \\ 1 \end{array} \right) \left(\frac{k}{2^n}\right) \\
&= \frac{k^2}{2^n} \qquad (6.1)
\end{aligned}$$

Table 6.2 gives some "small" period lengths of MaD0 and their associated probabilities computed using the above formula.

Table 6.2: [MaD0] Period Lengths and Associated Probabilities

| Period length | $\leq 2^{64}$ | $\leq 2^{128}$ | $\leq 2^{256}$ |
|---|---|---|---|
| Probability | $< 2^{-2112}$<br>$\approx 1.68 \times 10^{-636}$ | $< 2^{-1984}$<br>$\approx 5.71 \times 10^{-598}$ | $< 2^{-1728}$<br>$\approx 6.61 \times 10^{-521}$ |

### 6.2.4 Equidistribution

Equidistribution measures the uniformity of a PRNG. It converts the pseudorandom sequence generated from a PRNG into "random points" in a high-dimensional space (usually a unit space) and checks whether these points are randomly distributed in the space or have some distribution patterns that deviate from randomness. For example, the infamous RANDU [69], a widespread LCG PRNG in 1960s and early 1970s, has a "nice" lattice structure in the 3-dimensional space. In fact, any LCG PRNG has a lattice structure if it is used for the full period [70].

Now we give out the formal definition of equidistribution. Consider a pseudorandom sequence $\mathbf{x} = \{x_i\}$ of $w$-bit unsigned integers with period $2^n$. We can construct a sequence $\mathbf{v} = \{\mathbf{v}_i\}$ of $k$-dimensional vectors from $\mathbf{x}$ as follows:

$$\mathbf{v}_i = (x_i,\, x_{i+1},\, \cdots,\, x_{i+k-1}) \qquad i = 0,\, 1,\, 2,\, \cdots,\, 2^n - 1$$

By normalizing $\mathbf{x} = \{x_i\}$ into a real number sequence $\mathbf{y} = \{y_i\}$ in the interval [0, 1] through $y_i = x_i/2^w$, we get a new sequence $\mathbf{u} = \{\mathbf{u}_i\}$ from $\mathbf{v}_i$, where

$$\mathbf{u}_i = (y_i,\, y_{i+1},\, \cdots,\, y_{i+k-1}) \qquad i = 0,\, 1,\, 2,\, \cdots,\, 2^n - 1$$

If we equally divide the $k$-dimensional unit hypercube $[0,\, 1]^k$ into $2^{kw}$ cubic cells and map the vectors $\mathbf{u}$ to points in the k-dimensional space, we can measure the *k*-dimensional equidistribution to *w*-bit accuracy by computing the number of points (out of the total $2^n$ points in $\mathbf{u}$) in each cell. The pseudorandom sequence is said to be (*k*, *w*)-*equidistributed* if each cell contains the same number of points. The condition $n \geq kw$ is necessary if each

cell is going to contain at least one point.

The above definition of equidistribution has more value for short period PRNGs than for long period ones. For PRNGs such as MT, WELL, SFMT, and MaD0, no applications will use the full pseudorandom sequence of length $2^n$. A pseudorandom sequence having a perfect equidistribution within a full period does not mean it has a good equidistribution within a short segment. For example, MT is a 623-dimensionally equidistributed PRNG within its period $2^{19937} - 1$, but a short sequence generated from it can be seriously biased even in one dimension. Let us consider the number of 0's in the state of MT. According to central limit theorem, the number of 0's has a probability of 99.73% to fall within $\mu \pm 3\sigma$, where $\mu$ is the average number of 0's and $\sigma$ is the standard deviation. Because a 0-biased state is a more serious problem than a 1-biased state, we only consider the case $\mu + 3\sigma$. If we define a state with more than $\mu + 3\sigma$ 0's as a 0-biased state, then on average there is one 0-biased state out of each $1/(0.5 - 0.4987) \approx 770$ states. Since MT is slow to recover from a state with many 0's [3, 4], local equidistribution cannot be always maintained in MT.

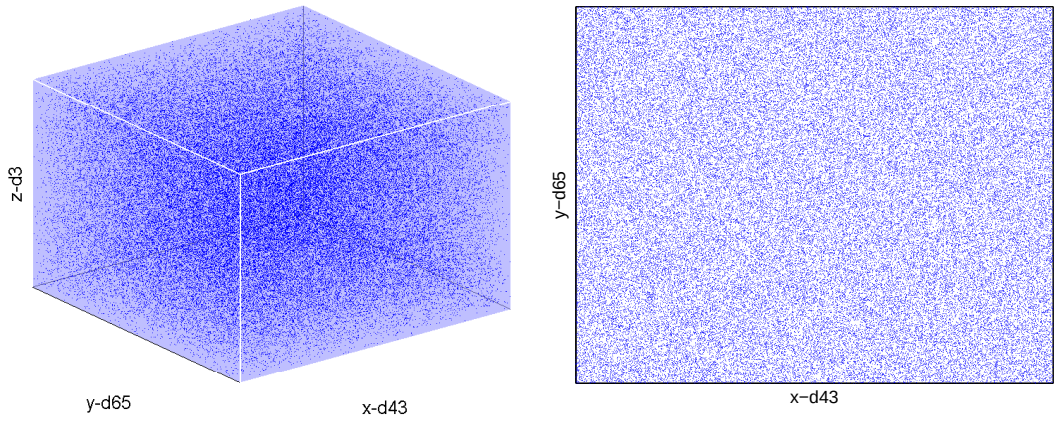
MaD0 has an expected period of $2^{1120}$. If we target a 32-bit accuracy, MaD0 can be at most (35, 32)-*equidistributed*. It is impossible to actually test if MaD0 is (35, 32)-*equidistributed* due to huge memory requirement and computational complexity. If we project the 35-dimensional space onto a subspace, for example a 3-dimensional space, we can check the equidistribution in this subspace. If MaD0 is (35, 32)-*equidistributed*, it should be equidistributed in all those subspaces. Although being equidistributed in all those subspaces is not a sufficient condition for drawing the conclusion that MaD0 is (35, 32)-*equidistributed*, it can be used for likelihood estimation. When testing in a subspace, we can use short pseudorandom sequences. As aforementioned, being equidistributed within the full period is not necessarily a desirable property of a PRNG with a long period. Using short sequences does not reduce the value of the testing from the perspective of applications.

There are $6545$ 3-dimensional subspaces in a 35-dimensional space. We perform an equidistribution test in all these subspaces. To speed up the testing and reduce the memory consumption, we divide each subspace into $(4w)^3$ cubic cells and use a pseudorandom

Table 6.3: [MaD0] Equidistribution Testing Results

| *P-value* interval | Number of *P-values* | | | | | |
|---|---|---|---|---|---|---|
| | RANDU | Xorgens | MT | WELL | SFMT | MaD0 |
| $[0.001, 0.01)$ | 11.1 | 54.2 | 49.1 | 74.4 | 69.2 | 65.3 |
| $(-\infty, 0.001)$ | 451.7 | 5.5 | 5.3 | 4.0 | 5.7 | 4.2 |

sequence of about 67 MB for testing, which corresponds to an average density of 8 points per cubic cell. The equidistribution is evaluated by applying a $\chi^2$ test and then computing a *P-value* corresponding to the goodness-of-fit distributional test on the density values. A larger *P-value* means a better fit, indicating better equidistribution in our case. We test six generators: RANDU, Xorgens, MT, WELL, SFMT and MaD0. Equidistribution is one of the main features of MT, WELL, and SFMT. This is the reason we choose them for comparison. Among all the PRNGs we tested, Xorgens and MaD0 are the only two PRNGs that pass all the statistical tests (see section 6.3). For this reason we decide to also include Xorgens in the equidistribution testing. RANDU is chosen as a bad example. For each generator, a random key is used to generate a pseudorandom sequence of about 67 MB. The test is repeated 10 times. *P-values* in the interval $(-\infty, 0.001)$ and $[0.001, 0.01)$ are counted for each test and averaged. Table 6.3 gives the average numbers of *P-values* that are in the two intervals. The testing results show that all generators except RANDU have similar performance. MaD0, MT, WELL, SFMT, and Xorgens each have less than 0.1% *P-values* less than 0.001. RANDU has significantly more (about 7%) *P-values* less than 0.001. Figure 6.4 (a) illustrates the distribution of random points in one 3-dimensional (dimension 43, 65, and 3) subspace for MaD0. The projection of this 3-dimensional subspace onto the xy-plane is shown in Figure 6.4 (b). Note that, for better viewing, only one out of every 200 points is depicted in each figure.




(a) (b)

Figure 6.4: [MaD0] Distribution of Random Points Generated by MaD0

### 6.2.5 Recovery from Biased States

A biased state can be either 0-biased or 1-biased. Since most operations cannot produce non-zero values from zero values, 0-biased states are more of a concern than 1-biased states. We measure the recovery speed from 0-biased states using the following steps:

1. Create an initial state that only has the first bit set to 1.
2. Scan the generated pseudorandom sequence using a window of 2048 bits at a step of one byte; count the number of 1's within the window and compare it with the ideal value 1024 to determine if the output is still 0-biased. We use a relatively small window compared with the 32000-bit window used in [4]. This helps us to better pinpoint the recovery point.

In the ideal case, the number of 1's within the 2048-bit window should be $\mu = 1024$. We measure the standard score by computing $z = (m - \mu)/\sigma$, where $m$ is the actual number of 1's within the 2048-bit window and $\sigma \approx 22.63$ is the standard deviation. A state is deemed unbiased if $|z| \leq 3$. We scan a pseudorandom sequence of about one million bits for each of the following five PRNGs: Xorgens, MT, WELL, SFMT, and MaD0. The results are presented in Figure 6.5. Xorgens can recover immediately. It is because the output of

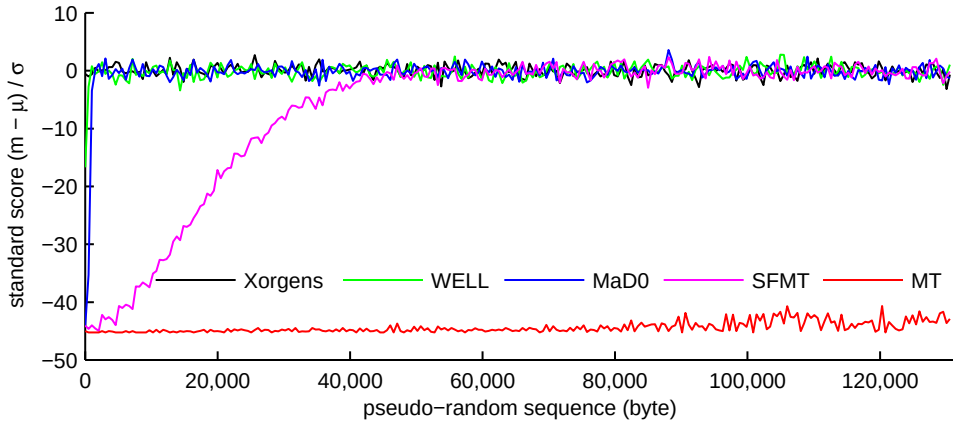



Figure 6.5: [MaD0] Recovery Ability from Biased States

Xorgens combines the output from an Xorshift generator and a Weyl generator, and the Weyl generator is initialized with a constant and virtually not impacted by the initial 0-biased state. However, the output from the Xorshift generator is 0-biased and the recovery point is around 28 KB. Before the recovery point, the quality of the output of Xorgens is mainly determined by the Weyl generator, which by itself cannot generate high quality pseudorandom numbers per our testing. WELL can recover within 230 bytes. MaD0 can recover within 1 KB. The recovery point for SFMT is around 42 KB. MT is the slowest one, taking about 3.1 MB to recover.

### 6.2.6 Ease of Use and Error-Proofing

To use MaD0, all a user needs to do is to provide a seed for state initialization. No matter what seed is provided, the initialized state will be well balanced since the 256-byte state table $S$ is initialized to a permutation of all the 256 possible values. By contrast, it is the user's responsibility to provide a *good* initial state in many other PRNGs.

Due to the excellent avalanche effect of state initialization, even a single bit change in the seed will change about half of the bits in the initialized state of MaD0. This means distinct sequences will be generated as long as different seeds are used, no matter by how many bits they are different. As such, it is easy to turn MaD0 into a parallel pseudorandom number generator by running multiple instances simultaneously or using multi-thread technology,

provided each instance or thread is given a different seed. This is not the case with most existing PRNGs. For example, two initialized states of MT may not have enough Hamming distance if caution is not exercised, in which case the two output sequences will be close to each other.

Hitting 0-biased states during a period is inevitable for most generators. In the case that MaD0 does fall into such a state, it can get out quickly due to its fast state update.

All the above mentioned features show that MaD0 is a care-free and error-proofing PRNG, well suited for serious applications.

## 6.3 Statistical Testing

### 6.3.1 NIST Statistical Test Suite

We tested and compared 12 PRNGs: LCG / ISO C, LCG / Java, LCG / Newlib, LCG / L1, LCG / L2, Xorshift, Xorshift7, Xorgens, MT, WELL, SFMT, and MaD0. The testing setup is the same as that for MARC (see subsection 4.3.1.1). The testing results are shown in Table 6.4. For easy reading, we have marked each failed test in red and with a star (*). The testing results show, besides LCG / ISO C, which failed test 7, all other generators passed the NIST statistical tests.

### 6.3.2 Diehard Battery of Tests

The testing results are given in Table 6.5. Both LCG / ISO C and LCG / Java significantly failed a couple of Diehard tests. Xorshift failed test 10. All other generators passed all the tests except for uniformity test of test 1, which we ignore for the reason given in subsection 4.3.2.2.

Table 6.4: [MaD0] Statistical Testing Results (NIST)

| Test | NoP | LCG / ISO C | | LCG / Java | | LCG / Newlib | | LCG / L1 | | LCG / L2 | | Xorshift | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS |
| 1 | 1 | 0.339271 | 989 | 0.406499 | 994 | 0.120909 | 992 | 0.937919 | 992 | 0.591409 | 988 | 0.628790 | 986 |
| 2 | 1 | 0.392456 | 993 | 0.861264 | 992 | 0.361938 | 985 | 0.892036 | 981 | 0.880145 | 991 | 0.433590 | 987 |
| 3 | 2 | 0.000432 | 986 | 0.574903 | 987 | 0.469232 | 990 | 0.576961 | 994 | 0.370262 | 988 | 0.188601 | 984 |
| | | 0.932333 | 986 | 0.911413 | 991 | 0.060112 | 992 | 0.664168 | 994 | 0.225998 | 986 | 0.146982 | 987 |
| 4 | 1 | 0.574903 | 992 | 0.065230 | 990 | 0.995777 | 995 | 0.761719 | 983 | 0.907419 | 984 | 0.025193 | 992 |
| 5 | 1 | 0.620465 | 985 | 0.955835 | 990 | 0.169981 | 985 | 0.727851 | 990 | 0.259616 | 991 | 0.653773 | 986 |
| 6 | 1 | 0.777265 | 992 | 0.420827 | 987 | 0.830808 | 990 | 0.848027 | 986 | 0.914025 | 989 | 0.544254 | 988 |
| 7 | 1 | 0* | 975* | 0.841226 | 990 | 0.839507 | 990 | 0.268917 | 990 | 0.363593 | 988 | 0.912724 | 993 |
| 8 | 148 | 0.475538 | 989 | 0.491992 | 990 | 0.502442 | 989 | 0.475143 | 989 | 0.498879 | 989 | 0.510839 | 990 |
| 9 | 1 | 0.599693 | 989 | 0.010988 | 988 | 0.032061 | 989 | 0.131122 | 986 | 0.630872 | 986 | 0.630872 | 989 |
| 10 | 1 | 0.310049 | 982 | 0.114040 | 985 | 0.560545 | 984 | 0.956729 | 990 | 0.916599 | 990 | 0.779188 | 989 |
| 11 | 1 | 0.229559 | 986 | 0.013856 | 995 | 0.893482 | 993 | 0.670396 | 988 | 0.496351 | 986 | 0.538182 | 991 |
| 12 | 8 | 0.484197 | $\frac{609}{617}$ | 0.399710 | $\frac{631}{638}$ | 0.458480 | $\frac{610}{616}$ | 0.558224 | $\frac{616}{622}$ | 0.535354 | $\frac{609}{617}$ | 0.539436 | $\frac{598}{605}$ |
| 13 | 18 | 0.619553 | $\frac{610}{617}$ | 0.545954 | $\frac{632}{638}$ | 0.495192 | $\frac{611}{616}$ | 0.443339 | $\frac{617}{622}$ | 0.494944 | $\frac{609}{617}$ | 0.444066 | $\frac{601}{605}$ |
| 14 | 2 | 0.348869 | 989 | 0.001641 | 987 | 0.839507 | 988 | 0.721777 | 992 | 0.081013 | 993 | 0.934599 | 984 |
| | | 0.383827 | 991 | 0.195864 | 995 | 0.933472 | 993 | 0.125927 | 988 | 0.941144 | 990 | 0.971006 | 990 |
| 15 | 1 | 0.884671 | 989 | 0.420827 | 993 | 0.906069 | 988 | 0.070299 | 988 | 0.620465 | 989 | 0.731886 | 983 |

| Test | NoP | Xorshift7 | | Xorgens | | MT | | WELL | | SFMT | | MaD0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS | $P\text{-}value_T$ | PoS |
| 1 | 1 | 0.915317 | 987 | 0.645448 | 989 | 0.820143 | 987 | 0.601766 | 991 | 0.029205 | 989 | 0.775337 | 991 |
| 2 | 1 | 0.326749 | 995 | 0.234373 | 984 | 0.207730 | 994 | 0.589341 | 990 | 0.366918 | 993 | 0.373625 | 990 |
| 3 | 2 | 0.325206 | 987 | 0.803720 | 992 | 0.854708 | 986 | 0.184549 | 989 | 0.068999 | 988 | 0.211064 | 993 |
| | | 0.205531 | 991 | 0.146982 | 989 | 0.664168 | 987 | 0.480771 | 990 | 0.001866 | 988 | 0.599693 | 994 |
| 4 | 1 | 0.050954 | 987 | 0.358641 | 990 | 0.851383 | 989 | 0.459717 | 992 | 0.053286 | 995 | 0.174728 | 985 |
| 5 | 1 | 0.465415 | 991 | 0.585209 | 988 | 0.534146 | 989 | 0.757790 | 988 | 0.461612 | 993 | 0.562591 | 986 |
| 6 | 1 | 0.342451 | 991 | 0.697257 | 990 | 0.245490 | 989 | 0.208837 | 993 | 0.216713 | 987 | 0.965860 | 989 |
| 7 | 1 | 0.044797 | 987 | 0.068571 | 985 | 0.291091 | 985 | 0.100709 | 986 | 0.120207 | 990 | 0.649612 | 986 |
| 8 | 148 | 0.483932 | 989 | 0.532581 | 990 | 0.517117 | 989 | 0.483836 | 990 | 0.507335 | 989 | 0.469644 | 990 |
| 9 | 1 | 0.233162 | 986 | 0.773405 | 994 | 0.334538 | 985 | 0.765632 | 986 | 0.848027 | 988 | 0.805569 | 986 |
| 10 | 1 | 0.342451 | 986 | 0.657933 | 988 | 0.152902 | 987 | 0.461612 | 987 | 0.365253 | 986 | 0.260930 | 986 |
| 11 | 1 | 0.161703 | 992 | 0.104993 | 992 | 0.248014 | 990 | 0.655854 | 990 | 0.316052 | 990 | 0.686955 | 992 |
| 12 | 8 | 0.392344 | $\frac{612}{618}$ | 0.601740 | $\frac{614}{622}$ | 0.403337 | $\frac{615}{623}$ | 0.506625 | $\frac{609}{615}$ | 0.486221 | $\frac{581}{589}$ | 0.538401 | $\frac{641}{646}$ |
| 13 | 18 | 0.545684 | $\frac{612}{618}$ | 0.452089 | $\frac{614}{622}$ | 0.470937 | $\frac{619}{623}$ | 0.516616 | $\frac{609}{615}$ | 0.564611 | $\frac{582}{589}$ | 0.530272 | $\frac{642}{646}$ |
| 14 | 2 | 0.851383 | 992 | 0.790621 | 993 | 0.647530 | 990 | 0.316052 | 987 | 0.120207 | 989 | 0.859637 | 995 |
| | | 0.209948 | 994 | 0.055010 | 991 | 0.291091 | 989 | 0.887645 | 988 | 0.419021 | 990 | 0.033807 | 994 |
| 15 | 1 | 0.450297 | 988 | 0.188601 | 995 | 0.022605 | 988 | 0.703417 | 991 | 0.844641 | 995 | 0.747898 | 989 |

Table 6.5: [MaD0] Statistical Testing Results (Diehard)

| Test | NoP | LCG / ISO C | | LCG / Java | | LCG / Newlib | | LCG / L1 | | LCG / L2 | | Xorshift | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p |
| 1 | 11 | 3.5e-008* | 0 | 1.7e-009* | 0 | 3.9e-010* | 0 | 2.9e-009* | 0 | 1.7e-007* | 0 | 2.9e-008* | 0 |
| 2 | 2 | 0.455937 | 0 | 0.249284 | 0 | 0.275709 | 0 | 0.334538 | 0 | 0.129620 | 0 | 0.419021 | 0 |
| 3 | 33 | 0* | 72* | 0* | 748* | 0.181257 | 0 | 0.104955 | 0 | 0.720362 | 0 | 0.343660 | 0 |
| 4 | 5 | 3.9e-010* | 0 | 0.889118 | 0 | 0.883171 | 0 | 0.641284 | 0 | 0.566688 | 0 | 0.195864 | 0 |
| 5 | 2 | 0.924076 | 0 | 0.419021 | 0 | 0.851383 | 0 | 0.975012 | 0 | 0.554420 | 0 | 0.807412 | 0 |
| 6 | 26 | 0.387795 | 1 | 0.010840 | 0 | 0.301307 | 1 | 0.973768 | 1 | 0.021261 | 0 | 0.365637 | 0 |
| 7 | 20 | 0.247382 | 0 | 0.939544 | 0 | 0.748891 | 0 | 0.647530 | 0 | 0.009267 | 0 | 0.756805 | 0 |
| 8 | 82 | 0* | 2093* | 0* | 2362* | 0.816978 | 0 | 0.934353 | 0 | 0.508558 | 0 | 0.784090 | 0 |
| 9 | 1 | 0.935716 | 0 | 0.514124 | 0 | 0.514124 | 0 | 0.249284 | 0 | 0.366918 | 0 | 0.350485 | 0 |
| 10 | 25 | 0.522900 | 1 | 0.982454 | 1 | 0.344048 | 0 | 0.007037 | 0 | 0.842595 | 0 | 3.2e-268* | 165* |
| 11 | 11 | 0.087499 | 0 | 0.455937 | 1 | 0.786139 | 0 | 0.680567 | 0 | 0.867692 | 0 | 0.857405 | 0 |
| 12 | 11 | 0.285811 | 1 | 0.066126 | 0 | 0.139438 | 0 | 0.268305 | 0 | 0.031561 | 0 | 0.103082 | 0 |
| 13 | 21 | 0.689510 | 0 | 0.327117 | 0 | 0.883887 | 0 | 0.835138 | 0 | 0.777357 | 0 | 0.553450 | 0 |
| 14 | 1 | 0.616305 | 0 | 0.085587 | 0 | 0.759756 | 0 | 0.494392 | 0 | 0.145326 | 0 | 0.085587 | 0 |
| 16 | 3 | 0.077290 | 0 | 0.064149 | 0 | 0.779188 | 1 | 0.851383 | 0 | 0.324180 | 0 | 0.090936 | 0 |
| 17 | 4 | 0.194289 | 0 | 1.0e-048* | 67* | 0.428095 | 0 | 0.591466 | 0 | 0.038678 | 0 | 0.038678 | 0 |

| Test | NoP | Xorshift7 | | Xorgens | | MT | | WELL | | SFMT | | MaD0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p | $P\text{-}value_T$ | x-p |
| 1 | 11 | 1.1e-009* | 0 | 2.7e-008* | 0 | 2.8e-009* | 0 | 1.7e-007* | 0 | 8.0e-008* | 0 | 2.2e-007* | 0 |
| 2 | 2 | 0.494392 | 0 | 0.171867 | 0 | 0.334538 | 0 | 0.181557 | 0 | 0.616305 | 0 | 0.657933 | 0 |
| 3 | 33 | 0.921474 | 0 | 0.016064 | 0 | 0.963081 | 0 | 0.050550 | 0 | 0.102341 | 0 | 0.37557 | 0 |
| 4 | 5 | 0.377007 | 0 | 0.591409 | 0 | 0.699313 | 0 | 0.699313 | 0 | 0.830808 | 0 | 0.377007 | 0 |
| 5 | 2 | 0.342451 | 0 | 0.729870 | 0 | 0.729870 | 0 | 0.904708 | 0 | 0.769527 | 0 | 0.678686 | 0 |
| 6 | 26 | 0.225273 | 0 | 0.135182 | 0 | 0.918841 | 0 | 0.671513 | 0 | 0.276765 | 0 | 0.759756 | 0 |
| 7 | 20 | 0.919131 | 0 | 0.792508 | 0 | 0.917236 | 0 | 0.849708 | 0 | 0.533139 | 1 | 0.666245 | 0 |
| 8 | 82 | 0.710306 | 0 | 0.884999 | 0 | 0.439754 | 0 | 0.702317 | 0 | 0.391609 | 0 | 0.558153 | 1 |
| 9 | 1 | 0.637119 | 0 | 0.289667 | 0 | 0.867692 | 0 | 0.319084 | 0 | 0.798139 | 0 | 0.474986 | 0 |
| 10 | 25 | 0.941778 | 0 | 0.240253 | 0 | 0.223882 | 0 | 0.802608 | 0 | 0.428460 | 0 | 0.183547 | 0 |
| 11 | 11 | 0.719747 | 0 | 0.253941 | 0 | 0.262249 | 0 | 0.935716 | 0 | 0.149879 | 0 | 0.146077 | 0 |
| 12 | 11 | 0.757969 | 0 | 0.152199 | 0 | 0.770410 | 0 | 0.827923 | 0 | 0.748981 | 0 | 0.964295 | 0 |
| 13 | 21 | 0.523624 | 0 | 0.206419 | 0 | 0.374107 | 0 | 0.289667 | 0 | 0.481416 | 1 | 0.85847 | 0 |
| 14 | 1 | 0.028817 | 0 | 0.964295 | 0 | 0.015598 | 0 | 0.924076 | 0 | 0.816537 | 0 | 0.12962 | 0 |
| 16 | 3 | 0.804337 | 0 | 0.685579 | 0 | 0.419021 | 0 | 0.115387 | 0 | 0.096578 | 0 | 0.616305 | 0 |
| 17 | 4 | 0.038678 | 0 | 0.588620 | 0 | 0.726812 | 0 | 0.726812 | 0 | 0.287847 | 0 | 0.975012 | 0 |

Table 6.6: [MaD0] Statistical Testing Results (TestU01)

| Battery | Parameters | Tests | NoP | Failures | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | LCG | | | | | Xorshift | | | MT | | | MaD0 |
| | | | | C | J | NL | L1 | L2 | XS | XS7 | XG | MT | WELL | SFMT | |
| SmallCrush | Built-in | 10 | 15 | 2 | 5 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| Crush | Built-in | 96 | 144 | 39 | 22 | 3 | 4 | 6 | 24 | 6 | 0 | 2 | 4 | 0 | 0 |
| BigCrush | Built-in | 106 | 160 | 84 | 49 | 8 | 8 | 8 | 17 | 6 | 0 | 2 | 6 | 2 | 0 |
| Rabbit | $32 \times 10^9$ bits | 26 | 40 | 15 | 1 | 0 | 0 | 0 | 11 | 3 | 0 | 0 | 2 | 0 | 0 |
| Alphabit | $32 \times 10^9$ bits | 9 | 17 | 12 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| BlockAlphabit | $32 \times 10^9$ bits | $6 \times 9$ | 102 | 70 | 38 | 0 | 0 | 0 | 38 | 0 | 0 | 0 | 0 | 0 | 0 |
| Abbr.: C = ISO C J = Java NL = Newlib XS = Xorshift XS7 = Xorshift7 XG = Xorgens | | | | | | | | | | | | | | | |

### 6.3.3 TestU01 Batteries of Tests

The testing results are given in Table 6.6. Xorgens and MaD0 are the only two generators that passed the TestU01 batteries of tests. MT and SFMT failed some Linear Complexity tests. WELL failed some Matrix Rank tests and Linear Complexity tests. The reason that each of MT, WELL, and SFMT failed some Linear Complexity tests is that they are all based on linear recurrences modulo 2 and their generated sequences lack the linear complexity of a truly random sequence. All other generators failed more tests, among which Xorshift, LCG / Java, and LCG / ISO C are the worst – failed 97, 115, and 222 tests respectively.

## 6.4 Performance Testing

The performance testing is carried out in the same way as in section 4.4. The results are given in Table 6.7. All generators are implemented using C programming language.

The performance testing is performed for the purpose of relative comparison, not for benchmarking, which would require more comprehensive testing on different platforms. For the same reason, we refrain from using the commercial Intel C/C++ compiler, which has the potential to generate faster executables than Microsoft Visual C/C++ compiler on Intel platforms.

Table 6.7: [MaD0] Pseudorandom Number Generation Speed (cycle/byte)

| Generator | Sequence size (KB) | | | | | |
|---|---|---|---|---|---|---|
| | 5 | 10 | 100 | 1000 | 10000 | 100000 |
| LCG / ISO C | 3.42 | 3.47 | 3.57 | 3.72 | 3.74 | 3.74 |
| LCG / Java | 1.20 | 1.20 | 1.28 | 1.29 | 1.30 | 1.30 |
| LCG / Newlib | 1.03 | 1.03 | 1.09 | 1.10 | 1.11 | 1.11 |
| LCG / L1 | 1.03 | 1.03 | 1.09 | 1.10 | 1.11 | 1.11 |
| LCG / L2 | 0.62 | 0.74 | 0.80 | 0.81 | 0.81 | 0.81 |
| Xorshift | 1.00 | 1.03 | 1.12 | 1.13 | 1.13 | 1.13 |
| Xorshift7 | 4.90 | 4.99 | 5.01 | 5.04 | 5.04 | 5.04 |
| Xorgens | 1.92 | 1.71 | 1.43 | 1.41 | 1.40 | 1.40 |
| MT | 10.15 | 7.22 | 4.59 | 4.37 | 4.33 | 4.31 |
| WELL | 3.41 | 3.62 | 3.67 | 3.68 | 3.69 | 3.69 |
| SFMT | 14.11 | 8.14 | 3.12 | 2.59 | 2.56 | 2.54 |
| SFMT (128-bit) | 11.09 | 5.87 | 1.02 | 0.55 | 0.51 | 0.50 |
| MaD0 | 2.71 | 1.59 | 0.57 | 0.47 | 0.46 | 0.46 |

MaD0 is faster than the 128-bit SFMT and is significantly faster than any other PRNGs when generating sequences longer than 100KB. Most LCG generators and Xorshift are relatively fast. LCG / L2 is the only 64-bit PRNG besides MaD0 that runs into one clock cycle per byte. Xorgens also has a decent speed. The 64-bit SFMT is not as fast as the above mentioned PRNGs but is faster than WELL and MT. Xorshift7 comes last in this race.

## 6.5 Conclusion

In this chapter we have presented a new pseudorandom number generator call MaD0, which can be efficiently implemented in software. It is much faster than any existing 64-bit PRNGs we know and is even faster than the 128-bit SFMT generator. It has a large internal state and a long period. It demonstrates an excellent randomness property. MaD0 also shows good high-dimensional equidistribution. It can recover from a biased state quickly

although in normal operations it is unlikely to fall into such a state. It includes a state initialization function that has superb avalanche effect and is of cryptographic quality in its design for ease of use and error-proofing. All these make MaD0 a unique and attractive candidate for parallel and distributed simulations and many other applications.

# Chapter 7

# MaD3: An Ultrafast Cryptographically Secure Pseudorandom Number Generator

In this chapter, we present a new CSPRNG called MaD3, which targets high performance cryptographic applications such as pervasive data encryption. MaD3 can generate high quality pseudorandom numbers with a speed that is unmatched by any CSPRNGs we know. It is designed to resist various known cryptographic attacks and withstand state compromise extension attacks as well. Its internal state comprises two parts: a byte-oriented part and an integer-oriented part, also referred to as byte-oriented state (BOS) and integer-oriented state (IOS) respectively. The byte-oriented part is initialized from a secret key and then further used to initialize the integer-oriented part. After initialization, both parts evolve, with the byte-oriented part serving as a source of entropy to the integer-oriented part. The transition of the byte-oriented part follows a pseudorandom permutation and the transition of the integer-oriented part follows a pseudorandom mapping.

The rest of this chapter is structured as follows. We describe the algorithm details in section 7.1 and the properties of the generator in section 7.2. Next, in section 7.3, we present the security analysis. In section 7.4 and 7.5, we give the statistical testing results and per-

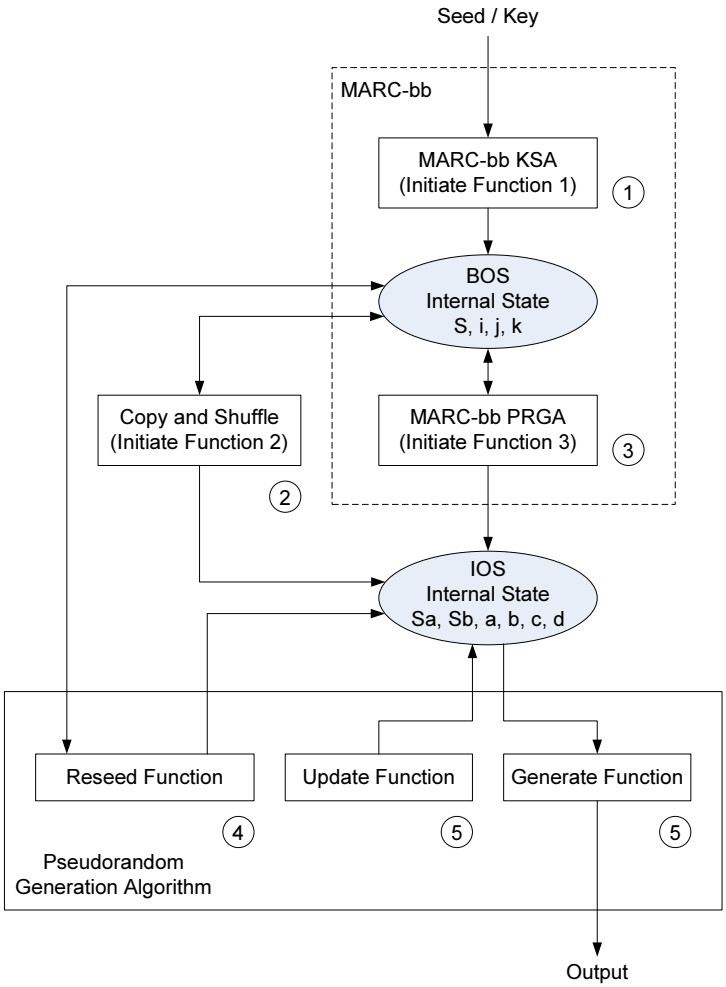



Figure 7.1: [MaD3] Functional Model

formance testing results respectively. Finally, in section 7.6, we conclude the chapter.

## 7.1 Algorithm Design

In this section we present the algorithm details of MaD3, including data structure, key scheduling, state initialization, and pseudorandom number generation. Figure 7.1 provides a functional model of MaD3. MaD3 uses MARC-bb for key scheduling and state initialization. The pseudorandom generation algorithm includes a *generate function* to generates pseudorandom numbers, an *update function* to update internal state, and a *reseed function* to provide new entropy. The details of this model are discussed in the following subsections.

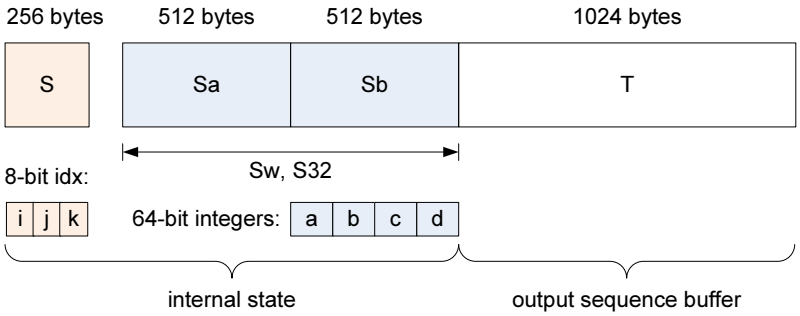



Figure 7.2: [MaD3] Data Structure

### 7.1.1 Data Structure

MaD3 maintains a data structure shown in Figure 7.2, which comprises one 256-byte state table (denoted as $S$), two 512-byte state tables (denoted as $Sa$ and $Sb$), three 8-bit indices (denoted as $i$, $j$, and $k$), four 64-bit integers (denoted as $a$, $b$, $c$, and $d$), and one 1024-byte output sequence buffer (denoted as $T$). State tables $S$, $Sa$, $Sb$, indices $i$, $j$, $k$, and integers $a$, $b$, $c$, $d$ construct the internal state of MaD3. $T$ is used for buffering pseudorandom numbers generated from the internal state. In some functions, the concatenation of $Sa$ and $Sb$ is used as a large 1024-byte state table, referred to as $Sw$, or cast into a 32-bit integer array of size 256, referred to as $S32$. In pseudorandom generation algorithm, state tables ($Sa$, $Sb$, $T$, and $Sw$) are cast into and used as 64-bit integer arrays.

### 7.1.2 Key Scheduling

MaD3 uses MARC-bb KSA for key scheduling. It accepts a key that is no larger than 64 bytes (512 bits). Out of the 64 bytes, we only claim security for 56 bytes. MaD3 does not specify the use of an initialization vector (IV) or a nonce, but an application can certainly use an IV as part of the key in the key scheduling.

### 7.1.3 State Initialization

The initialization of the internal state of MaD3 consists of three steps.

1. First, state table $S$ and indices $i$, $j$, and $k$ are initialized using MARC-bb KSA with the

Listing 7.1: [MaD3] Initialization Shuffling Algorithm (ISA)

```
# addition (+) and increment (++) operations
# are performed modulo 256
for r from 0 to 255
    i++
    j = j + S[i]
    k = k ^ j
    left_rotate(S[i], S[j], S[k])
endfor
```

key provided.

2. Next, state table *S* and indices *i*, *j*, and *k* are used to initialize state tables *Sa* and *Sb,* by repeating the following *copy-and-shuffle* process.

   (a) Copy *S*[0], *S*[1], ..., *S*[255] to *Sa*[0], *Sa*[1], ..., *Sa*[255]

   (b) Shuffle *S* according to the Initialization Shuffling Algorithm (ISA) shown in Listing 7.1. ISA shuffles the 256-byte state table *S* to generate a new permutation.

   (c) Copy *S*[0], *S*[1], ..., *S*[255] to *Sa*[256], *Sa*[257], ..., *Sa*[511];

   (d) Repeat ISA;

   (e) Copy *S*[0], *S*[1], ..., *S*[255] to *Sb*[0], *Sb*[1], ..., *Sb*[255];

   (f) Repeat ISA;

   (g) Copy *S*[0], *S*[1], ..., *S*[255] to *Sb*[256], *Sb*[257], ..., *Sb*[511];

   (h) Repeat ISA.

3. Last, 32 pseudorandom bytes are generated using MARC-bb PRGA. These 32 bytes are converted into four 64-bit integers using little endian and assigned to integers *a*, *b*, *c*, and *d*.

Now the internal state is initialized and contains five permutations of {0, 1, ..., 255}, three initialized 8-bit indices, and four initialized 64-bit integers.

### 7.1.4 Pseudorandom Number Generation

The pseudorandom generation algorithm is shown in Listing 7.2. MaD3 uses 64-bit operations for pseudorandom number generation. Initially, the output sequence buffer $T$ is marked as empty. During each round of pseudorandom number generation, 128 64-bit integers or 1024 bytes are generated and stored in the output sequence buffer. When a pseudorandom number generation request is received, the output sequence buffer is checked. If it is not empty, the data stored in it are used to serve the need. After all the data in the buffer are consumed, MaD3 refreshes the buffer by generating new pseudorandom numbers. During each generation round, the algorithm iterates 64 times[1] and generates two 64-bit integers in each iteration.

The internal state of MaD3 is functionally divided into two parts. State table $S$ and indices $i$, $j$, and $k$ form the byte-oriented state since they are used as bytes in all the operations. State tables $Sa$, $Sb$ and integers $a$, $b$, $c$, $d$ construct the integer-oriented state since they are used in 64-bit integer format in most operations. The byte-oriented state serves as a source of entropy to the integer-oriented state. Before each generation round, 32 pseudorandom bytes are obtained from the byte-oriented state using the reseed function given in Listing 7.3. These 32 bytes are converted into four 64-bit intermediate variables $e$, $f$, $g$, and $h$ using little endian. The reseed function is similar to MARC-bb PRGA except that the 32-bit integer array $S32$ is shuffled, i.e., four elements in $S32$ are left rotated during each of the eight iterations. The four integers $e$, $f$, $g$, and $h$ are then used to update $a$, $b$, $c$, and $d$.

Variable $x$ is a byte array of size 64, used as indices to access state tables for indirection operation. It is computed from the updated $a$, $b$, $c$, $d$, and two constants $M$ and $N$. Each byte of $x$ has a value falling in the range [0, 127] and any two bytes with a distance less than 4 have distinct values. The combination of $Sw[x[i]]$, $Sw[x[i]\text{\^{}}0x7c]$, $Sa[i]$, and $Sb[i]$ introduces pseudorandom indirect access and at the same time guarantees all state table integers get involved during each generation round. The way we choose these four state table

[1] Note that the iteration counter $i$ is a local variable, not the index $i$ used in byte-oriented state.

Listing 7.2: [MaD3] One Round of Pseudorandom Number Generation

```
## additions are performed modulo ##
## 0x10000000000000000            ##

# declare a byte array of size 64
byte x[64]

# cast the byte array into 64-bit integer array
x[64] => x64[8]

# reseeding
(e, f, g, h) = reseed(S32)

# update a, b, c, and d
a = a + e
b = b + f
c = c + g
d = d + h

# populate array x (through x64)
M = 0x7c7c7c7c7c7c7c7c
N = 0x0203000102030001
x64[0] = (a & M) | N
x64[1] = (b & M) | N
x64[2] = (c & M) | N
x64[3] = (d & M) | N
x64[4] = ((a >> 1) & M) | N
x64[5] = ((b >> 1) & M) | N
x64[6] = ((c >> 1) & M) | N
x64[7] = ((d >> 1) & M) | N

# generate pseudorandom numbers and update internal state
for i from 0 to 63
    a = a << 1
    b = b >> 1
    a = a + (e ^ Sw[x[i]])
    b = b + (f ^ Sw[x[i]^0x7c])
    c = c + (g ^ Sa[i])
    d = d + (h ^ Sb[i])
    T[2i] = c ^ (a + d)
    T[2i+1] = d ^ (b + c)
    Sw[x[i]] = a + b
endfor
```

Listing 7.3: [MaD3] Reseed Function

```
# function reseed(S32)
for r from 0 to 7
    i++
    j = j + S[i]
    k = k ^ j
    swap(S[i], S[j])
    m = S[j] + S[k]
    n = S[i] + S[j]
    left_rotate(S32[i], S32[j], S32[k], S32[n])
    output S[m]
    output S[n]
    output S[m ^ j]
    output S[n ^ k]
endfor
```

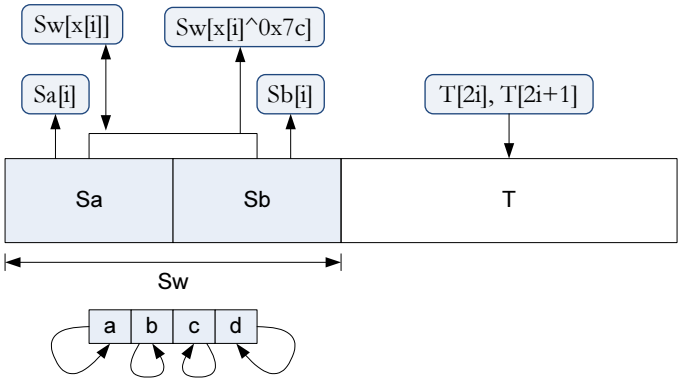



Figure 7.3: [MaD3] Data Flow in One Iteration of PRGA

integers during each iteration deserves some explanations. First note that both $Sw[x[i]]$ and $Sw[x[i]$^0x7c] can access either state table $Sa$ or state table $Sb$ but they can never access the same state table, which also means each state table has the same chance to be accessed by them. The use of constant $M$ and $N$ results in a special feature – the four state table integers $Sw[x[i]]$, $Sw[x[i]$^0x7c], $Sa[i]$, and $Sb[i]$ are distinct and they are also different from any of the four state table integers used in the previous or next iteration. By distinct and different, we mean they point to different state table integers, which do not necessarily but with a high probability have different values. One can verify this feature by observing the following facts: $Sw[x[i]]$ and $Sw[x[i]$^0x7c] are from different tables; so are $Sa[i]$ and $Sb[i]$; the lower two bits of $x[i]$ and $x[i]$^0x7c come from $N$ and cycle through the values 2, 3, 0, 1, while those of $i$ cycle through the values 0, 1, 2, 3. This is demonstrated in Table 7.1.

Figure 7.3 summarizes the data flow during one iteration in pseudorandom number generation, with detailed interactions among different elements omitted.

For efficiency and simplicity, only a few types of operations are used. They are bitwise AND, bitwise OR, bitwise XOR, addition, left logical bitwise shift, and right logical bitwise shift, each taking only one clock cycle for most processors when operands are immediate constants or register variables [71]. All four integers $a$, $b$, $c$, and $d$ are updated during each iteration. Note that, as $e$ and $f$ are fixed during each generation round, the least significant bit (LSB) of $a$ and the most significant bit (MSB) of $b$ have a high correlation with $Sw[x[i]]$ and $Sw[x[i]\text{^}0x7c]$ respectively. This LSB and MSB high correlation can be eliminated by replacing the shift operations with rotation operations. One reason we use shift operations instead of rotation operations is that, although most processors have a bitwise rotation instruction, C programming language does not directly support it. Another reason is that neither $a$ nor $b$ is directly exposed to the outside in pseudorandom number generation.

Besides $a$, $b$, $c$, and $d$, one element from state table $Sa$ or $Sb$ is also updated via $Sw[x[i]]$ during each iteration. In other words, nearly half of $Sa$ and $Sb$ is updated during each generation round or on average each state table element has a 50% chance to get updated. Is it fast enough to update half of $Sa$ and $Sb$ during each generation round? The answer is "yes". Due to the shift operations, the value of $a$ is determined by the most recent 64 values of $Sw[x[i]]$ (and the fixed $e$) and the value of $b$ is largely determined by the most recent 64 values of $Sw[x[i]\text{^}0x7c]$ (and the fixed $f$). On the other hand, $c$ and $d$ are permanently affected by any state table element that has been involved in the computation of their values. This means the update of a single state table element can completely change the evolution path of $c$ and $d$. From the way $a$, $b$, $c$, and $d$ are computed, it suffices to only update some of the elements in $Sa$ and $Sb$ during one generation round.

Table 7.1: [MaD3] State Table Access during Pseudorandom Number Generation

| State table integer | Subscript | State table accessed | Subscript values (last 2 bits) |
|---|---|---|---|
| *Sw*[*x*[*i*]] | *x*[*i*] | *Sy* (*y* = *a* or *b*) | 2, 3, 0, 1, ... |
| *Sw*[*x*[*i*]^0x7c] | *x*[*i*]^0x7c | *Sz* (*z* = *a* or *b*, $z \neq y$) | 2, 3, 0, 1, ... |
| *Sa*[*i*] | *i* | *Sa* | 0, 1, 2, 3, ... |
| *Sb*[*i*] | *i* | *Sb* | 0, 1, 2, 3, ... |

## 7.2 Properties

### 7.2.1 Pseudorandom Mapping State Transition

The internal state of MaD3 consists of two parts: the byte-oriented state including state table *S* plus indices *i*, *j*, and *k*; and the integer-oriented state including state tables *Sa* and *Sb* plus integers *a*, *b*, *c*, and *d*. The byte-oriented state has 2072 bits and the integer-oriented state has 8448 bits. The expected period of the byte-oriented state is about $2^{1699}$ according to subsection 4.2.1.

To determine the expected period of the integer-oriented state, we need to know the features of its state transition. Specifically we want to know if the transition of the integer-oriented state follows a pseudorandom mapping. To find out the answer, we perform a state transition test like the one we did for MaD0 in subsection 6.2.2. Because only part of the integer-oriented state is updated during each generation round, we need to select the next state after several generation rounds rather than one generation round. The principle of next state selection is to ensure a mapping from one state to its next state is complete. In MaD0, a state after each generation round is chosen since the mapping is complete during one generation round. In MaD3, a state that is around 11 generation rounds away from the current state is chosen as the next state so that the mapping can be complete. The analysis is given below.

During each generation round of MaD3, 64 integers of state table $S_w$ are updated through $S_w[x[i]]$. If all updated integers are distinct, then exactly half of state table $S_w$

Table 7.2: [MaD3] Chi-Square Statistic Testing Results for State Table Transition

| State distance (generation round) | $\chi^2$ | *C.V.* ($\alpha = 0.01$) | $\chi^2$ < *C.V.*? |
|---|---|---|---|
| 8 | $3.42 \times 10^8$ | 2200.09 | No |
| 9 | $4.38 \times 10^4$ | | No |
| 10 | $3.76 \times 10^3$ | | No |
| 11 | 966.26 | | Yes |
| 12 | 474.40 | | Yes |
| Testing parameters:<br>Number of transitions in each run (*N*): 100352<br>Output size: 2048 bits (the first 256 bytes of *Sw*)<br>Critical value (for $\alpha = 0.01$ and *d.o.f.* = 2048) for rejecting $\mathcal{H}_0$: 2200.09 | | | |

is updated during one generation round. The updated ratio will increase to $\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$ after the second generation round, $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} = \frac{7}{8}$ after the third generation round, and so on. After 8 generation rounds, each integer of state table $S_w$ has a probability of $\frac{255}{256}$ to be updated. However, the assumption that the 64 updated integers are distinct is obviously not true. Some integers may be updated more than once because the access of $S_w$ is controlled by the "random" variable $x[i]$, not the counter $i$. For this reason, we need to pick a new state after more than 8 generation rounds to ensure the whole state table is updated and the mapping is complete.

It is worth noting that, whether the state is fully updated during one big step or during multiple small steps should not make much difference to the expected period. This is probably more obvious in RC4, whose state table is completely updated during 256 iterations instead of one iteration. Although we can choose a faster state update algorithm (for example, update more than 64 integers during each generation round) and/or a more uniform state update algorithm (for example, update through *Sw*[*i*] instead of *Sw*[*x*[*i*]]), we opt not to do so for the potential performance and/or security penalties. While the number of generation rounds or iterations between two chosen states does not need to be one, this number should not be too large either. Otherwise it may hide the actual state transition feature.

Table 7.2 gives the testing results for the first 256 bytes of *Sw* (no differences are observed among the different parts of *Sw* in our test). Each $\chi^2$ value given in the table is the

Table 7.3: [MaD3] Period Lengths and Associated Probabilities

| Period length | $\leq 2^{64}$ | $\leq 2^{128}$ | $\leq 2^{256}$ |
|---|---|---|---|
| Probability | $< 2^{-10028}$<br>$\approx 1.87 \times 10^{-3019}$ | $< 2^{-9900}$<br>$\approx 6.35 \times 10^{-2981}$ | $< 2^{-9644}$<br>$\approx 7.36 \times 10^{-2904}$ |

average result of 10 runs. The testing results show that the state transition between two states that are 11 or more generation rounds apart is close to a random mapping.

Based on the above test, the transition of the integer-oriented state seems following a pseudorandom mapping.

### 7.2.2 Period

The internal state of MaD3 consists of a byte-oriented state whose transition follows a pseudorandom permutation and an integer-oriented state whose transition follows a pseudorandom mapping. The expected period of the byte-oriented state is about $2^{1699}$. From equation (2.1), the expected period of the integer-oriented state is about $2^{8448/2} = 2^{4224}$. The overall expected period of MaD3 is about $2^{1699+4224} = 2^{5923} \approx 1.00 \times 10^{1783}$.

Table 7.3 gives some "small" period lengths of MaD3 and their associated probabilities computed using equation (6.1). Note that, MaD3 has a state of 10520 bits, but the 2048-bit state table *S* is a permutation of {0, 1, ..., 255}, which gives a total $256! \approx 2^{1684}$ states instead of $2^{2048}$ states. Therefore we need to substitute 10156 instead of 10520 for *n* in equation (6.1) when computing the probabilities.

## 7.3 Security Analysis

### 7.3.1 Resistance against Known Attacks

In this subsection, we give a security analysis of MaD3 in the context of known attacks, including special attacks mounted against RC4 and other generic attacks.

#### 7.3.1.1 Attacks against RC4

RC4 key scheduling algorithm and keystream generation algorithm are based on simple random permutations. This approach makes RC4 one of the most efficient and popular stream ciphers so far. Nonetheless the simplicity of the cipher also leads to some security issues.

In [58], Roos described a class of weak keys in RC4. The author revealed if the first byte and the second byte of the key satisfy the condition $key[0] + key[1] \equiv 0 \, mod \, 256$, then the first output byte has a high probability to be $key[2] + 3$. This class of weak keys affect one key out of every 256 keys.

In [59], Golić derived a linear model of RC4 using the linear sequential circuit approximation method. It is proven that the second binary derivative of the least significant bit output sequence is correlated to 1 with the correlation coefficient close to $15 \times 2^{-3n}$ for large $2^n$ where *n* is the variable word size of RC4 (usually 8). According to this model, it requires about $64^n/225$ keystream words to detect the linear statistical weakness of RC4. This is significantly smaller than $2^p$,where $p = n2^n + 2n$ is the bit size of RC4 internal state.

In 2001, Fluhrer et al. described two significant weaknesses of RC4 key scheduling algorithm [57]. The first one is the existence of large classes of weak keys, whose length is divisible by some non-trivial power of two, i.e., $\ell = 2^q m$ for some $q > 0$. As a result, the initial keystream output is disproportionally affected by a small number of key bits in those keys. The second weakness is a related key vulnerability. The authors observed that when the same secret part of the key is used with numerous different exposed values, it takes relatively little work to rederive the secret part by analyzing the initial word of the keystreams. The significance of this finding is that many applications, including the Wired Equivalent Privacy (WEP) protocol, construct RC4 keys by concatenating a long term secret key with a varying but publicly known IV, thus vulnerable to this related key attack.

A strong correlation between the observable *i*, *S*[*n*] and the internal *j*, *S*[*i*], *S*[*j*] was reported by Klein in [60]. The author showed that, when the internal states of RC4 are approximated using uniform distribution, the following probability equations hold:

$$P(S[j] + S[m] \equiv i \, mod \, N) = \frac{2}{N}$$

$$P(S[j] + S[m] \equiv c \, mod \, N) = \frac{N-2}{N(N-1)}$$

where $N = 2^8$ and $c \neq i$. This strong correlation improves the attack described in [57] and enabled Tews et al. to break 104-bit WEP in less than one minute [61].

There are other attacks reported in the open literature. The success of those attacks has revealed several design problems of RC4:

1. The key scheduling algorithm is too simple – 256 swaps are not enough to break the correlation between the input key and the initialized internal state.
2. The internal state evolves relatively simply and slowly. This helps transfer the initial correlation into later states and the keystream.
3. Permutations are invertible and easier to analyze than non-invertible mappings.

MaD3 uses rotation operations instead of swap operations in its key scheduling and state initialization. This is more efficient and has a better mixing effect. In addition, limiting the maximum key size of MaD3 to 64 bytes helps prevent related key attacks, since any differences between two input keys will come into play within the first 64 iterations of key scheduling, giving more time for diffusion compared with a longer key that might be used with RC4. Persisting the values of all three indices *i*, *j*, and *k* after key scheduling, as opposed to resetting indices *i* and *j* to 0 in RC4, further helps prevent related key attacks, since those indices are sensitive to the change in the input key and affect the output keystream more directly than the state table *S*. Diffusion is most effective when different keys result in different index *j* and/or *k*, which has a high probability to happen in MaD3. More importantly, the state initialization repeats a *copy-and-shuffle* process 4 times to ini-

tialize the entire internal state, which provides more chances for diffusion. This overall design makes it very difficult to find simple correlation relationships between the input key and the initialized internal state as in [57, 58, 60]; nor is it possible to derive a linear model to approximate MaD3 as done for RC4 in [59].

The correlation between the input key and the internal state does not pose a real threat unless it is transferred into the output pseudorandom sequence and thus becomes exploitable. Unfortunately, RC4's simple keystream generation algorithm (swap two elements and then pick one element from the state table for output during each iteration) is too simple to efficiently break or hide the correlation between the input key and the internal state. By contrast, MaD3 uses a more sophisticated pseudorandom generation algorithm. Each output integer is computed from three integers ($a$, $c$, and $d$, or $b$, $c$, and $d$), all in turn computed from many state table integers. This can effectively prevent the correlation, if any, from being transferred into the output pseudorandom sequence.

Random permutations give an average period length $2^{n-1}$ for an $n$-bit internal state. Random mappings, on the other hand, render a much shorter average period length, which according to equation (2.1) is only about $2^{n/2}$ for an $n$-bit internal state. This difference, however, has no practical effects when $n$ is sufficiently large. For example, while the average period length $2^{5923}$ of MaD3 is much shorter than the average period length $2^{10519}$ of random permutations for $n = 10520$, it is still more than enough for any applications. Using random mappings instead of random permutations makes MaD3 more resistant to algebraic cryptanalysis. Since random mappings are non-invertible, MaD3 also possesses another feature that RC4 lacks, that is, knowing a state does not enable one to go back to its previous state (see subsection 7.3.2 for more details).

#### 7.3.1.2 Time-Memory Tradeoff Attacks

Time-memory tradeoff attacks rely on precomputation to reduce the effort needed for recovering the internal state and/or secret key [72]. This type of attacks proceed as follows: assume that the PRNG is in a certain state and calculate a number of output bits and put the

pair (output, state) in a sorted list; after enough pairs are calculated and stored, try to match a received output sequence with the saved output sequences; if the match is successful, then with some likelihood the internal state or partial of it may be determined, which may further lead to the recovery of the secret key. The parameters in a time-memory tradeoff attack are time (*T*), memory (*M*), and amount of output data (*D*). In general $T \times M^2 \times D^2 = S^2$, where *S* is the state space of the PRNG and $D^2 \leq T$. The precomputation time *P* is computed as $P = S/D$. The design strength of MaD3 is 448 bits. For the brute-force equivalent attack with $T = 2^{448}$ and $D \leq \sqrt{T} = 2^{224}$, $M = S/D/\sqrt{T} \geq 2^{10156}/2^{224}/2^{224} = 2^{9708}$. The lower bound on memory for the attack is $2^{9708}$ bits, which is simply impractical.

#### 7.3.1.3 Guessing Attacks

The strategy for this type of attacks is to guess a small part of the internal state and then deduce the remaining part. This is particularly powerful when applied to a word-based PRNG because a word-based PRNG has a relatively small number of internal words and any word guessed has a good chance to participate in the computation of next iteration if the algorithm is not designed with caution. The consequence is that more and more words get revealed and the PRNG is eventually broken. MaD3 is designed to resist this type of attacks.

To be successful, an attacker must be able to do two things, namely, be able to efficiently verify his guessing (guess and verify) and be able to determine more unknowns based on his guessing (guess and determine). In MaD3, by knowing the value of *T*[2*i*] at a certain moment, an attacker can guess two of the three integers (*a*, *c*, and *d*) and then compute the third integer. Since he also knows *T*[2*i*+1], he can further compute the value of *b*. The attacker needs to guess 128 bits to figure out the values of all four integers *a*, *b*, *c*, and *d*. Once the attacker knows *a* and *b*, he can compute *Sw*[*x*[*i*]] = *a* + *b*. To know *x*[*i*] and therefore identify which integer is to be updated, he needs to guess another 5 bits (out of the 7 bits, the lower 2 bits are known apriori). So the attacker needs to guess 133 bits (128 bits if he chooses not to know *x*[*i*]) in total during the first iteration of guessing.

During the second iteration, the attacker needs to guess 128 bits like in the first iteration to figure out the new values of *a*, *b*, *c*, and *d*, and then another 128 bits to figure out the values of two of *e*, *f*, *g*, and *h* (and also two of the four state table integers *Sw*[*x*[*i*]], *Sw*[*x*[*i*]^0x7c], *Sa*[*i*], and *Sb*[*i*]). Note that since the attacker only needs to find out two values (one must be *c* or *d*) so as to know all the values of *a*, *b*, *c*, and *d* during each iteration, it is not necessary for him to find out all the values of *e*, *f*, *g*, and *h*. Also notice that the second 128 bits guessing is based on the design that the four state table integers *Sw*[*x*[*i*]], *Sw*[*x*[*i*]^0x7c], *Sa*[*i*], and *Sb*[*i*] are distinct and they are also different from any of the four state table integers used in the previous iteration. If two integers, for example *Sw*[*x*[*i*]] and *Sa*[*i*], are identical, then the attacker only needs to guess 64 bits instead of 128 bits. If a state table integer used in the previous iteration, for example the already known *Sw*[*x*[*i*]], can appear in the next iteration, then the 128 bits guessing is also reduced to 64 bits. Here we have ignored the relatively small cost that is needed to make two integers point to the same state table integer (a probability of $\frac{1}{128} = 2^{-7}$ or a cost of 7 bits) or make a state table integer used in the previous iteration appear in the next iteration (a probability of $2 \times \frac{1}{128} = 2^{-6}$ or a cost of 6 bits; the coefficient 2 comes from that each of the two state table integers whose values need to be determined can take the known value).

During the third iteration, the attacker still needs to find out the new values of two of the four integers *a*, *b*, *c*, and *d*. To achieve this, he needs to know the values of two of the four state table integers *Sw*[*x*[*i*]], *Sw*[*x*[*i*]^0x7c], *Sa*[*i*], and *Sb*[*i*]. He does not need to guess 128 bits, however, because he already knows the values of three distinct state table integers, one during the first iteration and two during the second iteration. The one whose value is found during the first iteration has a probability of $2^{-5}$ to appear in the third iteration, thus reducing the workload from 128 bits to $64 + 5 = 69$ bits.

The first three iterations require more than 448 bits of work, which is our design strength. But let us go a little further to see what the cost the attacker needs to pay if he continues. During the fourth iteration, all the three state table integers whose values are found during the first two iterations have a chance to reappear. But since one has already

reappeared in the third iteration (otherwise the third iteration requires a 128 instead of 69 bits guessing), only the two state table integers whose values are found during the second iteration may reappear in the fourth iteration[2]. The probability that they both reappear is $\frac{1}{32} \times \frac{1}{32} = 2^{-10}$. This shows that the guessing cost is only 10 bits during the fourth iteration.

The above attack is not unique and different attack strategies can be taken, but none is likely to be more efficient than the above one. The above analysis only covers one generation round. Between each two generation rounds, the attacker has to deal with the 32 pseudorandom bytes obtained from state table *S*, otherwise he will not be able to track the values he has already guessed and determined. So the cost to completely break MaD3 through guessing attacks is far more than that of a brute force attack. To conclude this subsection, we also want to point out that it is infeasible to break the 64-bit integers into smaller units so as to reduce the attack cost. If the smaller units, say bytes, can be computed independently, then the attack cost will be significantly reduced. This is because each 64 bits can be reduced to, for example, eight 8 bits, which is equivalent to 11 bits only ($8 \times 2^8 = 2^{11}$).

#### 7.3.1.4 Algebraic Attacks

Algebraic attacks exploit various types of algebraic relations between the input states and the output states of a cryptographic algorithm. They can be applied to a variety of ciphers, ranging from public key cryptosystems like HFE [73] to block ciphers like AES and Serpent [74], and stream ciphers like Toyocrypt [75], Bluetooth [76], and SNOW [77].

So far algebraic attacks on PRNGs are mainly applied to those whose internal state is updated in a linear way. Most LFSR-based PRNGs fall into this category. A typical LFSR-based PRNG consists of an internal state *S*, a linear state update function *L*, and a nonlinear output function *f*. Let $S_0$ denote the initial internal state at time $t = 0$, then at time *t* the internal state is $L^t(S_0)$ and the pseudorandom output is $z_t = f(L^t(S_0))$. The

[2]For this to happen, the attacker must not have chosen to work on *Sa*[*i*] and *Sb*[*i*], since the value of *i* cannot repeat during one generation round.

goal of an algebraic attack is to recover the initial state or the secret key by setting up and solving a system of equations based on the algebraic relations between the internal state bits (or secret key bits) and the observed output bits. A less ambitious goal is to use it as a distinguisher, in which case the attacker tries to see whether he can construct a set of equations that hold with a probability non-trivially higher for the output sequence of a PRNG than for a truly random sequence.

If the system is over-defined, linearization technique [78, 79] may be used to convert the nonlinear equations into linear ones. The basic idea is to introduce a new variable for each monomial. For a nonlinear boolean function $f$ with $l$ variables and of degree $d$, the maximum number of monomials is $\binom{l}{0} + \binom{l}{1} \cdots + \binom{l}{d} \approx \binom{l}{d}$. The data complexity for solving a linear system with $\binom{l}{d}$ variables is $\binom{l}{d}$ output bits and the time complexity is $\binom{l}{d}^{\omega}$, where $\omega$ is between 2.3727 and 3 [80–82]. There exist other methods for solving nonlinear equations, notably those for computing Gröbner bases [83]. To date the only method whose complexity is well studied is linearization, but in general all those methods rely on the existence of low degree equations to work efficiently. As such, one key step of algebraic attacks is to find low-degree equations.

Using annihilators is one of the techniques to produce low degree algebraic equations. An annihilator of $f$ is a non-zero function $g$ such that $f \times g = 0$. The resistance of a boolean function $f$ against algebraic attacks is measured using algebraic immunity *AI(f)*. It is shown in [79] that the algebraic immunity *AI(f)* of a boolean function $f$ is the minimum value of $d$ such that $f$ or $f + 1$ admits an annihilating function of degree $d$. This is so since $f = 1$ leads to $g = 0$ if $f \times g = 0$ and $f = 0$ leads to $g = 0$ if $(f + 1) \times g = 0$, that is, a high degree equation $f = 1$ or $f = 0$ can be reduced to a low degree equation $g = 0$. To further reduce the degree of the equations, fast algebraic attacks were introduced by Courtois [84]. The strategy of fast algebraic attacks is to eliminate all high degree monomials independent of the keystream bits in a precomputation step.

MaD3 is word-based and both its state update function and output function are nonlinear. The analysis of this type of PRNGs against algebraic attacks is more difficult than

Table 7.4: Results of Algebraic Analysis for RC4

| Operation | $v$ | $d_1$ | $d_2$ |
|---|---|---|---|
| Pointer addition | $n$ | 3 | $n$ |
| State permutation | $n2^n$ | 3 | $n+1$ |
| Keystream generation | $2n$ | $n$ | $n$ |

those for LFSR-based ones. The only related work we can find is by Wong et al., who analyzed the RC4 family of stream ciphers against algebraic attacks in [85]. For a word size of $n$ bits, the algebraic analysis results for the main operations used in RC4 (pointer addition, state permutation, and keystream generation) are given in Table 7.4. $v$ is the number of variables introduced to the system; $d_1$ and $d_2$ are the maximum degrees of the equations with and without introducing low degree multiples (i.e., annihilators) respectively. Remarkably, these three operations constitute a strong system of equations, with word addition making the equation system inseparable, state permutation being a primary contribution to the number of equations, and keystream generation yielding a system of high degree ($n$ is the highest possible algebraic immunity for a boolean function with $2n$ variables according to Courtois and Meier [79]). The analysis and experiment results from the authors suggest that RC4 is most likely immune from algebraic attacks at present.

MaD3 shares some common features with RC4: The operations of MaD3 include word addition; its key scheduling algorithm is based on permutation; its pseudorandom number generation involves indirect access. Aside from those common features, MaD3 is different from RC4. First, MaD3 has a huge internal state that is substantially larger than that of RC4, meaning during an algebraic attack more equations (and likely of higher degrees) need to be handled and more output bits need to be collected. Second, MaD3 switches from random permutations to random mappings once the state initialization is complete. The simple algebraic structure of random permutations has helped the authors to construct the equations and reduce their degrees in [85]. Building and solving equations for MaD3 will be more difficult and costly. Finally, the pseudorandom number generation of MaD3 is more complex and involves more operations than that of RC4, which will also make

algebraic attacks more difficult for MaD3 than for RC4.

Let us take a look at the state update function and the output function of MaD3. The byte-oriented state is updated in the same way as in RC4 and we will not discuss it here. As shown in Listing 7.2, the update of the integer-oriented state is done through $Sw[x[i]] = a + b$ and the output is done through $T[2i] = c \wedge (a + d)$ and $T[2i + 1] = d \wedge (b + c)$. For output part, we will only examine $T[2i]$, since $T[2i + 1]$ is generated in a similar way. To distinguish old values (i.e., values from previous iteration) from current values when the expressions are expanded, we add iteration counter *i* to all variables like this $Sw[x[i]]_i = a_i + b_i$. Expanding $a_i$, $c_i$, and $d_i$ gives us

$$\begin{aligned} Sw[x[i]]_i &= (a_{i-1} \ll 1) + e \wedge Sw[x[i]]_{i-1} + (b_{i-1} \gg 1) \\ &\quad +(f \wedge Sw[x[i] \wedge 0x7c]_{i-1}) \qquad (7.1) \\ T[2i]_i &= (c_{i-1} + g \wedge Sa[i]_{i-1}) \\ &\quad \wedge ((a_{i-1} \ll 1) + e \wedge Sw[x[i]]_{i-1} + d_{i-1} + (h \wedge Sb[i]_{i-1})) \qquad (7.2) \end{aligned}$$

where *e*, *f*, *g*, and *h* are obtained from state table *S*. The above results cannot be directly used to construct a system of multivariate polynomial equations that are needed for algebraic attacks. This is because index $x[i]$ and integers *e*, *f*, *g*, and *h* are themselves unknown variables. The authors of [85] solved the index problem by utilizing the canonical isomorphism between the residue class ring $\mathbb{Z}/2^n\mathbb{Z}$ representing integers modulo $2^n$ and the product ring $\mathbb{F}_2^n$ representing the bit strings of those integers. As a result, all equations describing RC4 are generated as polynomials with coefficients in $\mathbb{F}_2$. In the following analysis, we adopt the same notations and techniques used in [85] and we will temporarily ignore the interactions between the byte-oriented state and the integer-oriented state of the state, that is, treat *e*, *f*, *g*, and *h* as if they are constants and ignore the left rotation among *S32*[*i*], *S32*[*j*], *S32*[*k*], and *S32*[*n*]. Let $u_{(b)} \in \mathbb{F}_2$ $(0 \leq b \leq n - 1)$ denote the *b*-th least significant bit of $u \in \mathbb{Z}/2^n\mathbb{Z}$ (for MaD3, $n = 64$) and let $Sw_{k,(b)}$ $(0 \leq k \leq 127)$ denote the

*b*-th least significant bit of the *k*-th state table variable in *Sw*, then we have

$$Sw_{k,(b)} = \sum_{u=0}^{127} \left( Sw_{u,(b)} \prod_{b=0}^{6} (k_{(b)} + u_{(b)} + 1) \right)$$

The above expression has a degree of 7. Now we need to find the equivalent operations in $\mathbb{F}_2$ for other operations (word addition, bitwise XOR, left logical bitwise shift, and right logical bitwise shift). The equivalent addition over the binary digits in $\mathbb{F}_2^n$ for $u + v = w$ ($u$, $v$, $w \in \mathbb{Z}/2^n\mathbb{Z}$) is defined as

$$\begin{aligned} \mathbf{u} + \mathbf{v} &= (u_{(0)}, u_{(1)}, \cdots, u_{(n-1)}) + (v_{(0)}, v_{(1)}, \cdots, v_{(n-1)}) \\ &= (w_{(0)}, w_{(1)}, \cdots, w_{(n-1)}) \\ &= \mathbf{w} \end{aligned}$$

where $w_{(b)}$ satisfies

$$w_{(b)} = \sum_{k=0}^{b-1} \left( u_{(k)} v_{(k)} \prod_{l=k+1}^{b-1} (u_{(l)} + v_{(l)}) \right) + u_{(b)} + v_{(b)}$$

and has a maximum degree of *n* when $b = n - 1$. For bitwise XOR, we have

$$w_{(b)} = u_{(b)} + v_{(b)}$$

The equivalent operation in $\mathbb{F}_2$ for left logical bitwise shift $w = u \ll 1$ ($u$, $w \in \mathbb{Z}/2^n\mathbb{Z}$) is

$$\begin{aligned} w_{(b)} &= \begin{cases} 0 & \text{for } b = 0 \\ u_{(b-1)} & \text{for } n > b > 0 \end{cases} \\ &= \sum_{k=0}^{b-1} \left( u_{(k)} \prod_{l=k+1}^{b-1} 0 \right) \end{aligned}$$

The above result can also be derived from fact that $w = u \ll 1 = u + u$, that is,

$$\begin{aligned} w_{(b)} &= \sum_{k=0}^{b-1} \left( u_{(k)} u_{(k)} \prod_{l=k+1}^{b-1} (u_{(l)} + u_{(l)}) \right) + u_{(b)} + u_{(b)} \\ &= \sum_{k=0}^{b-1} \left( u_{(k)} \prod_{l=k+1}^{b-1} 0 \right) \end{aligned}$$

Similarly for right logical bitwise shift, we get

$$\begin{aligned} w_{(b)} &= \begin{cases} 0 & \text{for } b = n - 1 \\ u_{(b+1)} & \text{for } 0 \leq b < n - 1 \end{cases} \\ &= \sum_{k=b+1}^{n-1} \left( u_{(k)} \prod_{l=b+1}^{k-1} 0 \right) \end{aligned}$$

Let

$$\begin{aligned} F_u = {} & (a_{i-1} \ll 1) + (e \wedge Sw[x[i]]_{i-1}) + (b_{i-1} \gg 1) + (f \wedge Sw[x[i] \wedge 0x7c]_{i-1}) \\ & - Sw[x[i]]_i \\ F_o = {} & (c_{i-1} + (g \wedge Sa[i]_{i-1})) \wedge ((a_{i-1} \ll 1) + (e \wedge Sw[x[i]]_{i-1}) + d_{i-1} + (h \wedge Sb[i]_{i-1})) \\ & - Sc[i]_i \end{aligned}$$

then we can rewrite equations (7.1) and (7.2) as $F_u = 0$ and $F_o = 0$ respectively.

The subtractions can be converted into additions using 2's complement, for example, $-Sw[x[i]]_i = Sw[x[i]_i \wedge 0x\text{FFFFFFFFFFFFFFFF} + 1$. Noticing that bitwise XOR and logical bitwise shift are linear in $\mathbb{F}_2$ and that each addition operation contributes a degree of *n* in $\mathbb{F}_2$ if all variables involved are distinct, the degrees of $F_u$ and $F_o$ can be easily computed and they are both equal to $4 \times n = 256$. For 8448 boolean variables, the maximum number of monomials $T \approx \binom{8448}{256} = \frac{8448!}{256!(8448-256)!} > 2^{1649}$, which gives a data complexity of $T$ and a time complexity between $T^{2.3727}$ and $T^3$. The attacker can also try to build equations for key bits instead of state bits, but then he has to deal with the state initialization.

The above analysis suggests MaD3 is resistant to algebraic attacks, since the degree of the equations must be less than 19 to make algebraic attacks more efficient than a 448-bit brute force search. In addition, the construction and resolution of equations will become much more difficult, if possible at all, when taking into account the interactions between the byte-oriented state and the integer-oriented state. The interactions include two nonlinear operations: the generation of *e*, *f*, *g*, and *h*; the left rotation among *S32*[*i*], *S32*[*j*], *S32*[*k*], and *S32*[*n*]. The first operation necessitates some processing similar to that done for RC4 in [85] and the second operation makes the actual update function more complicated than the one used in the above analysis.

While the above analysis suggests MaD3 is resistant to algebraic attacks, we also want to emphasize that it does not serve as a proof for several reasons. First, the degree of equations may be reduced if suitable annihilators are found (although, on the other hand, the degree may also increase if more iterations are considered in the analysis). Second, it is not necessary to carry out the attack in $\mathbb{F}_2$. In the residue class ring $\mathbb{Z}/2^n\mathbb{Z}$, the analysis will be very different and may change our view pertaining to MaD3's resistance to algebraic attacks. In $\mathbb{Z}/2^n\mathbb{Z}$, the addition operation becomes linear, but the combination of addition and other operations like bitwise XOR cannot be handled using pure linear algebra. Furthermore, we need to find a way to handle the indirect access, which remains a barrier so far. Finally, there also exists the potential to reduce the attack complexity by combining algebraic attacks with other attacks like Guess-and-Determine attacks. One such attempt

was made by E. Mehrabi et al. [86], but no significant conclusion was drawn.

#### 7.3.1.5 Distinguishing Attacks

As its name suggests, a distinguishing attack tries to distinguish the output sequence of a PRNG from a truly random sequence. All statistical testing tools are designed for detecting this problem in the design of various pseudorandom number generators and ciphers. Passing statistical tests, however, does not necessarily mean a pseudorandom number generator or cipher is immune to distinguishing attacks. A sequence not distinguishable from a truly random sequence by statistical testing tools may still be revealed as not random by distinguishing attacks. This is particularly true when considering that distinguishing attacks often explore specific design details that are not considered by general purpose testing tools.

One way to launch a distinguishing attack against a PRNG is to explore the algebraic structure of the PRNG and try to demonstrate that it can be (partially) described by some algebraic equations, thereby proving it is not random. Although similar techniques are used, a distinguishing attack does not try to completely solve the algebraic equations as an algebraic attack does. It suffices to show that those equations are satisfied with a non-trivially high probability that cannot happen with a truly random sequence. There are only two values in $\mathbb{F}_2$ and therefore each equation is satisfied with a probability of 0.5 for a truly random sequence. For a successful algebraic attack, each equation is satisfied with a probability of 1. For a successful distinguishing attack, each equation should be satisfied with a probability that is non-trivially higher than 0.5 or, equivalently, the number of equations that are satisfied should be non-trivially larger than half of the total number of equations. As shown in subsection 7.3.1.4, the degree of equations derived for one iteration is 256 and the degree for a feasible algebraic attack must be less than 19. This big difference indicates MaD3 has a strong resistance to both algebraic attacks and distinguishing attacks based on similar techniques.

Another distinguishing attack technique that targets stream ciphers using linear masking

was proposed by Coppersmith et al. and applied to SNOW 1.0 [87]. Stream ciphers (also PRNGs) usually include some nonlinear process in their design. The nonlinear process resembles a block cipher and its states are deemed uncorrelated if they are far away in time. Linear masking tries to mask the correlation among states close in time. It masks those states using independent parts of a linear process. The basic idea of the attack is to find some linear combination of the linear process that vanishes. When this same combination is applied to the output stream, the linear process would vanish. This way the attacker is left with the nonlinear process only, for which he can further look for a characteristic that can be distinguished from randomness. Distinguishing attacks have also been mounted for other stream ciphers that use linear masking, including SNOW 2.0 and Sosemanuk [88–90]. MaD3 does not use linear masking, thus rendering this type of distinguishing attacks irrelevant.

#### 7.3.1.6 Differential Attacks

Differential cryptanalysis tries to track the relationship between differences in input and differences in the corresponding output of a cryptosystem. A special differential cryptanalysis called impossible differential cryptanalysis exploits differences that are impossible (i.e., having a probability of 0) instead of differences that have a probability higher than what can be expected from a random transformation. Differential attacks are of particular concern and have long since been the subject of intensive research for block ciphers and other cryptosystems whose process depends on the input plaintext.

Differential cryptanalysis can be used to track how differences of keys and/or IVs propagate and affect the internal state and the output sequence of a PRNG. This has been demonstrated in several attacks on RC4 [91–93]. All these attacks have exploited the relatively simple key scheduling algorithm and pseudorandom generation algorithm of RC4. As shown in chapter 5, MaD3 significantly improves the avalanche effect by using a more complex key scheduling algorithm than RC4. This makes it more difficult to track the relationship between the input key and the internal state. Compared with RC4, MaD3 also

performs some additional state initialization and uses a more complex pseudorandom generation algorithm, which further enhance its resistance against differential attacks.

#### 7.3.1.7 Side Channel Attacks

This type of attacks take advantage of side-channel information to recover the secret parameters of a cryptosystem such as the key and internal state. Side-channel information can be any information related to a specific implementation. Examples are timing information, power consumption, cache, electromagnetic emissions, faults, visual and acoustic information, and so on. Side channel attacks are practical concerns in the design of almost all cryptosystems (one exception is hash functions, which have no secret parameters). They are very efficient and have been successfully applied to RSA [94–96] and AES [97–101]. In [99], the authors took only 65 milliseconds to recover the full AES key. Side channel attacks are of concern also because many techniques and countermeasures that are used to thwart classical attacks either have no effect on side channel attacks or, even worse, turn out to be helpful to them. For example, longer keys, larger S-boxes, and more complex operations can all make side channel attacks easier.

It is primarily the task of implementers to defend against side channel attacks that exploit electromagnetic emissions, faults, visual and acoustic information, etc. Attention from developers or from both developers and implementers is needed to defend against some other side-channel attacks, including timing attacks, power consumption attacks, and cache timing attacks.

MaD3 only uses a few simple operations, each having a constant execution time on most platforms. There are no branches used, which can result in variations in timing and power consumption and have been exploited to attack DSA [102] and ECDSA [103]. No information about the size or the value of the key is used in any way to control what instructions to be executed or how many times they will be executed. This is manifested in the design of our key scheduling algorithm, where a fixed number of iterations are used although making the number of iterations depend on the size of the key is slightly more

efficient.

Cryptosystems that use fixed look-up tables such as S-boxes are vulnerable to cache timing attacks. This has been demonstrated in several attacks against AES [97, 99]. Cryptosystems that heavily rely on slowly evolving permutation tables can also suffer from this type of attacks. For example, in [104] the authors presented a cache timing attack against RC4. They were able to recover the whole internal state of RC4 within a minute by querying the RC4 encryption process byte by byte. Two design factors have helped in the attack. First, the state table evolves slowly and thus caching can take place. Second, the state table is a permutation. This fact enables the authors to use an assignment algorithm such as the Hungarian algorithm [105] from time to time to improve the data complexity and the success probability. MaD3 does not use fixed look-up tables. Although a permutation table is also used in the pseudorandom number generation of MaD3, it is used for reseeding, not directly used to output pseudorandom numbers as in RC4 and thus cannot be easily linked to the observable output sequence. In addition, the permutation table is only a small part of the entire internal state. The other part of the internal state evolves quickly because of pseudorandom mappings and reseeding. In conclusion, MaD3 is not vulnerable to the above mentioned cache timing attacks.

### 7.3.2 Next-Bit Test and State Compromise Extensions

Besides those requirements set for an ordinary PRNG, a CSPRNG also needs to meet some additional requirements, that is, satisfying the "next-bit test" and withstanding "state compromise extensions". In this subsection we address these two requirements.

#### 7.3.2.1 Next-Bit Test

Given the first $k$ bits of a random sequence, if there is no polynomial-time algorithm that can predict the ($k$+1)-th bit with a probability of success significantly greater than 0.5, then the random sequence is said to pass the next-bit test [106]. A more general test is Yao's test [29], which tests whether a random sequence is distinguishable from a

truly random sequence by any polynomial-time algorithm. Obviously next-bit test is only a special case of Yao's test and passing next-bit test is a necessary condition for passing Yao's test. However, Yao proved that passing next-bit test is also a sufficient condition for passing Yao's test. The significance of Yao's work is that it reduces randomness test to a single test, i.e., next-bit test.

It is important to notice that the next-bit test is a theoretical test due to the fact that it needs to be conducted against every polynomial-time algorithm, which is impossible in practice. Although some next-bit tests have been developed [107–110], they are only based on a few specific prediction algorithms and passing them is far from enough to conclude a PRNG passes the next-bit test. Those tests have more values for research than for practical use.

Since it is impossible to implement a perfect next-bit test, no PRNG can prove it passes the next-bit test in the strict sense given in [29]. A common practice is to relax the requirement by replacing *all polynomial-time algorithms* with *(most) known polynomial-time algorithms*. This less strict requirement can be addressed from several respects:

1. Several standard statistical test suites have been developed. They include many tests designed for distinguishing a pseudorandom sequence from a truly random sequence. We tested MaD3 using those statistical test suites and the results will be given in section 7.4.
2. Standard statistical test suites are developed with no knowledge of a specific PRNG, which means they do not address the next-bit test issue with respect to the PRNG algorithm itself. An attacker can use the PRNG algorithm itself to predict or even compute the next bit if he knows part or all of the internal state. How to prevent an attacker from using the PRNG algorithm itself to predict the next bit is addressed by the requirement that a CSPRNG should be able to withstand state compromise extensions. We will discuss MaD3's this capability in next subsection.
3. A third group of polynomial-time algorithms are those used in various known attacks. Being resistant to known attacks is a necessary condition for passing the next-bit test.

For MaD3, this has largely been addressed in section 7.3.

4. There are always unknown attacks and new attacks can keep on emerging. As such, a CSPRNG should be routinely revisited and, if necessary, revised.

#### 7.3.2.2 State Compromise Extensions

A state compromise extension attack attempts to recover unknown outputs and/or internal states of a PRNG by using the knowledge of the internal state of the PRNG at some time. A formal requirement defined in terms of backtracking resistance and prediction resistance is given in NIST special publication 800-90A [28]:

- *Backtracking Resistance* – Backtracking resistance is provided relative to time $T$ if there is assurance that an adversary who has knowledge of the internal state of a PRNG at some time subsequent to time $T$ would be unable to distinguish between observations of ideal random bitstrings and (previously unseen) bitstrings that were output by the PRNG prior to time $T$.

- *Prediction Resistance* – Prediction resistance is provided relative to time $T$ if there is assurance that an adversary who has knowledge of the internal state of the PRNG at some time prior to $T$ would be unable to distinguish between observations of ideal random bitstrings and bitstrings output by the PRNG at or subsequent to time $T$.

The above definition means that a CSPRNG should withstand both distinguishing attacks and state recovery attacks in both backward and forward directions in the case the internal state is compromised at some time. Since distinguishing attacks are easier than state recovery attacks, being resistant to this type of attacks means a higher requirement for the design of the generator.

While MaD3 is not based on computationally hard problems, it is practically impossible to go backwards, either from the observed sequence of pseudorandom numbers to the internal state or from a compromised state to its previous states or the sequence of pseudorandom numbers generated before the state is compromised. MaD3 uses both pseudoran-

dom mappings and pseudorandom permutations for state transition. Since pseudorandom mappings are non-invertible, knowing a state does not enable one to go back to its previous state. By contrast, RC4 uses pseudorandom permutations for state transition. Pseudorandom permutations are invertible and it requires no efforts to go from a state to its previous state. For example, RC4's state transition is done through the following code:

```
i++
j = j + S[i]
swap(S[i], S[j])
```

If the internal state (*j* plus *S*) completely leaks out or is compromised, one can easily go back by reversing the algorithm, that is,

```
swap(S[i], S[j])
j = j - S[i]
i--
```

In Mad3, during each of the 64 iterations of the generation round, all four state integers *a*, *b*, *c*, and *d* are updated. Another state table integer *Sw*[*x*[*i*]] is also updated. The update of those integers involves the old values of those integers, the byte array *x*, and the intermediate variables *e*, *f*, *g*, and *h*. Therefore it is impossible to directly compute the old values from the current known values. Knowing the output pseudorandom sequence additionally does not help much either, since each output integer is computed from many internal integers and the computation involves indirect access (via *Sw*[*x*[*i*]] and *Sw*[*x*[*i*]^0x7c]), which is nonlinear. It is also worth noting that, to go back to the previous state, one must find out the byte array *x* and the four intermediate variables *e*, *f*, *g*, and *h*, which are computed at the beginning of the generation round, that is, they are computed from some state that is far before the previous state one is trying to recover. In principle, this issue can be solved by constructing and resolving a system of equations. But neither the construction nor the resolution of such a system of equations would be easy, if not impossible, due to the special design of the pseudorandom generation algorithm (please refer to subsection 7.3.1.4 for details).

Prediction resistance can only be achieved when at least some part of the internal state or some entropy input is kept unknown or unpredictable from attackers. Knowing the whole internal state does enable one to compute the future states. This is likely to happen when an attacker gains physical access to the internal state. It is often assumed that one or more unpredictable sources of entropies are available to a CSPRNG. In other words, it is a reasonable assumption that at least some information (e.g., a seed, part of the internal state, etc.) is not leaked out. In our case, we assume at least one part of the internal state is not compromised. The internal state of MaD3 is functionally divided into two parts. A specific implementation should take this into consideration and maintain the two parts of the internal state in such a way that they are unlikely to be compromised (e.g., through physical access) at the same time.

## 7.4 Statistical Testing

### 7.4.1 NIST Statistical Test Suite

We tested and compared four pseudorandom number generators: ISO C function rand()[3], RC4, SHA1 (running in counter mode), and MaD3, and a true random number generator, QRNG (a quantum random number generator available at `http://qrng.physik.hu-berlin.de/`). For ISO C function rand(), RC4, and MaD3, a random key is generated for each sequence and used to initialize the generator. This random key can be up to 64 bytes. SHA1 runs in counter mode with an initial 512-byte random counter. For QRNG, we directly connect to `http://qrng.physik.hu-berlin.de/` and download as many random numbers as we need.

The testing results are shown in Table 7.5. The results show, besides ISO C function rand() that is not designed for security applications, all other generators passed the NIST statistical tests.

[3]While it is listed here as a "bad" PRNG example, ISO C function rand() is not a cryptographic pseudorandom number generator and is not required to pass those tests designed for security testing.

Table 7.5: [MaD3] Statistical Testing Results (NIST)

| Test | NoP | rand() | | RC4 | | SHA1 | | QRNG | | MaD3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *P-value*$_T$ | PoS | *P-value*$_T$ | PoS | *P-value*$_T$ | PoS | *P-value*$_T$ | PoS | *P-value*$_T$ | PoS |
| 1 | 1 | 0.339271 | 989 | 0.161703 | 991 | 0.811080 | 991 | 0.018540 | 992 | 0.326749 | 991 |
| 2 | 1 | 0.392456 | 993 | 0.146982 | 998 | 0.925287 | 992 | 0.715679 | 994 | 0.723804 | 994 |
| 3 | 2 | 0.000432 | 986 | 0.572847 | 989 | 0.883171 | 987 | 0.417219 | 987 | 0.743915 | 993 |
| | | 0.932333 | 986 | 0.928857 | 989 | 0.419021 | 991 | 0.568739 | 990 | 0.278461 | 994 |
| 4 | 1 | 0.574903 | 992 | 0.229559 | 984 | 0.041981 | 991 | 0.697257 | 991 | 0.140453 | 989 |
| 5 | 1 | 0.620465 | 985 | 0.595549 | 991 | 0.066882 | 989 | 0.862883 | 986 | 0.285427 | 985 |
| 6 | 1 | 0.777265 | 992 | 0.134172 | 992 | 0.655854 | 990 | 0.946308 | 983 | 0.461612 | 992 |
| 7 | 1 | 0* | 975* | 0.274341 | 988 | 0.213309 | 989 | 0.753844 | 980 | 0.548314 | 988 |
| 8 | 148 | 0.475538 | 989 | 0.470224 | 989 | 0.511719 | 989 | 0.493607 | 990 | 0.508195 | 990 |
| 9 | 1 | 0.599693 | 989 | 0.404728 | 992 | 0.743915 | 988 | 0.295391 | 985 | 0.800005 | 989 |
| 10 | 1 | 0.310049 | 982 | 0.029401 | 988 | 0.788728 | 991 | 0.249284 | 989 | 0.266235 | 989 |
| 11 | 1 | 0.229559 | 986 | 0.676615 | 990 | 0.459717 | 990 | 0.763677 | 995 | 0.952152 | 991 |
| 12 | 8 | 0.484197 | $\frac{609}{617}$ | 0.299100 | $\frac{587}{592}$ | 0.362296 | $\frac{603}{610}$ | 0.542881 | $\frac{599}{606}$ | 0.602597 | $\frac{613}{620}$ |
| 13 | 18 | 0.619553 | $\frac{610}{617}$ | 0.417622 | $\frac{585}{592}$ | 0.562638 | $\frac{604}{610}$ | 0.485159 | $\frac{600}{606}$ | 0.531121 | $\frac{614}{620}$ |
| 14 | 2 | 0.348869 | 989 | 0.530120 | 991 | 0.769527 | 987 | 0.344048 | 988 | 0.417219 | 994 |
| | | 0.383827 | 991 | 0.576961 | 989 | 0.952152 | 990 | 0.162606 | 988 | 0.138860 | 993 |
| 15 | 1 | 0.884671 | 989 | 0.771469 | 991 | 0.246750 | 993 | 0.753844 | 985 | 0.096578 | 993 |

Table 7.6: [MaD3] Statistical Testing Results (Diehard)

| Test | NoP | rand() | | RC4 | | SHA1 | | QRNG | | MaD3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *P-value*$_T$ | x-p | *P-value*$_T$ | x-p | *P-value*$_T$ | x-p | *P-value*$_T$ | x-p | *P-value*$_T$ | x-p |
| 1 | 11 | 3.5e-008* | 0 | 5.3e-010* | 0 | 8.4e-005* | 0 | 1.1e-007* | 0 | 5.0e-008* | 0 |
| 2 | 2 | 0.455937 | 0 | 0.816537 | 0 | 0.637119 | 0 | 0.383827 | 0 | 0.514124 | 0 |
| 3 | 33 | 0* | 72* | 0.170722 | 0 | 0.468595 | 0 | 0.739918 | 0 | 0.329332 | 0 |
| 4 | 5 | 3.9e-010* | 0 | 0.534146 | 0 | 0.419021 | 0 | 0.502247 | 0 | 0.350485 | 0 |
| 5 | 2 | 0.924076 | 0 | 0.024356 | 0 | 0.311542 | 0 | 0.883171 | 0 | 0.798139 | 0 |
| 6 | 26 | 0.387795 | 1 | 0.711915 | 0 | 0.437985 | 0 | 0.145643 | 0 | 0.563063 | 0 |
| 7 | 20 | 0.247382 | 0 | 0.100709 | 0 | 0.644407 | 0 | 0.305599 | 0 | 0.550347 | 0 |
| 8 | 82 | 0* | 2093* | 0.339928 | 1 | 0.428095 | 0 | 0.206253 | 1 | 0.133031 | 0 |
| 9 | 1 | 0.935716 | 0 | 0.289667 | 0 | 0.964295 | 0 | 0.003996 | 0 | 0.759756 | 0 |
| 10 | 25 | 0.522900 | 1 | 0.815812 | 0 | 0.048591 | 0 | 0.416859 | 1 | 0.886162 | 0 |
| 11 | 11 | 0.087499 | 0 | 0.492614 | 0 | 0.937721 | 0 | 0.035812 | 0 | 0.130301 | 0 |
| 12 | 11 | 0.285811 | 1 | 0.415748 | 0 | 0.100327 | 0 | 0.180658 | 0 | 0.24698 | 0 |
| 13 | 21 | 0.689510 | 0 | 0.587668 | 0 | 0.807850 | 0 | 0.270523 | 0 | 0.413032 | 0 |
| 14 | 1 | 0.616305 | 0 | 0.494392 | 0 | 0.534146 | 0 | 0.289667 | 0 | 0.249284 | 0 |
| 16 | 3 | 0.077290 | 0 | 0.334538 | 0 | 0.110952 | 0 | 0.449672 | 0 | 0.228764 | 0 |
| 17 | 4 | 0.194289 | 0 | 0.875539 | 0 | 0.600729 | 0 | 0.549331 | 0 | 0.44476 | 0 |

### 7.4.2 Diehard Battery of Tests

The testing results are given in Table 7.6. ISO C function rand() failed several Diehard tests. All other generators passed all the tests except for test 1, which we ignore for the reason mentioned in subsection 4.3.2.2.

### 7.4.3 TestU01 Batteries of Tests

We did not test QRNG, since we were unable to download the required amount of true random data from the server. The testing results are given in Table 7.7. ISO C function rand() failed big in TestU01 tests. RC4 had 2 suspect *P-values* during the first run but both were cleared during retries. SHA1 and MaD3 passed all the tests with no suspect *P-values*.

Table 7.7: [MaD3] Statistical Testing Results (TestU01)

| Battery | Parameters | Tests | NoP | Failures | | | |
|---|---|---|---|---|---|---|---|
| | | | | rand() | RC4 | SHA1 | MaD3 |
| SmallCrush | Built-in | 10 | 15 | 2 | 0 | 0 | 0 |
| Crush | Built-in | 96 | 144 | 39 | 0 | 0 | 0 |
| BigCrush | Built-in | 106 | 160 | 84 | 0 | 0 | 0 |
| Rabbit | $32 \times 10^9$ bits | 26 | 40 | 15 | 0 | 0 | 0 |
| Alphabit | $32 \times 10^9$ bits | 9 | 17 | 12 | 0 | 0 | 0 |
| BlockAlphabit | $32 \times 10^9$ bits | $6 \times 9$ | 102 | 70 | 0 | 0 | 0 |

Table 7.8: [MaD3] Pseudorandom Number Generation Speed (cycle/byte)

| Generator | Sequence size (KB) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 5 | 10 | 100 | 1000 | 10000 |
| RC4 | 9.53 | 7.67 | 7.09 | 6.98 | 7.04 | 7.04 |
| HC-128 | 55.21 | 13.27 | 7.96 | 3.58 | 3.15 | 3.11 |
| MaD3 (32-bit) | 51.43 | 12.01 | 7.54 | 3.19 | 2.86 | 2.83 |
| MaD3 (64-bit) | 47.04 | 10.27 | 5.36 | 1.29 | 0.85 | 0.82 |

## 7.5 Performance Testing

The performance testing is carried out in the same way as in section 4.4. We compare 32-bit and 64-bit MaD3 executables with RC4 and HC-128. HC-128 is the fastest stream cipher among the four software-efficient finalists of eStream. We do not compare with the CSPRNGs introduced in subsection 2.1.2 because they are far slower than stream ciphers like RC4.

The state initialization of MaD3 is slower than that of RC4 due to the use of larger state. For this reason, RC4 beats MaD3 for short sequences. For long sequences, MaD3 is faster than RC4. For sequences of 100 KB or more, the 32-bit MaD3 is more than twice faster than RC4 and the 64-bit MaD3 is 5 to 8 times faster than RC4. HC-128 is slower than RC4 for sequences up to 10 KB, but reaches a speed twice faster than RC4 for sequences longer than 100 KB. Both the 32-bit and 64-bit MaD3 executables outperform HC-128 in

all cases. The 64-bit MaD3 is more than 3 times faster than HC-128 when the sequence size is 1 MB or more.

## 7.6 Conclusion

In this chapter we have presented a new CSPRNG called MaD3. It has a huge internal state of 10520 bits and an expected period of $2^{5923}$. It resists various known attacks. On a typical Intel Core i3 personal computer, MaD3 can run into one clock cycle per byte, which is several times faster than any existing CSPRNGs we know. We have also tested MaD3 using the NIST statistical testing suite, the new Diehard battery of tests, and the TestU01 batteries of tests. The testing did not raise any red flag.

# Chapter 8

# Non-Deterministic Pseudorandom Number Generation

A PRNG usually generates pseudorandom numbers in a deterministic way. This makes it possible to reproduce a pseudorandom sequence, which is necessary or useful in some applications, such as data encryption and simulations. A true random number generator (TRNG), on the other hand, works in a non-deterministic way. Non-deterministic random number generation is preferred in applications such as gambling and lottery, where fairness is essential and manipulation should not be possible. TRNGs, however, are still too expensive, relatively slow, and not generally available. In this chapter we focus on how to turn MaD0 and MaD3 into non-deterministic PRNGs. The design goal and approach are given in section 8.1. Algorithm modification is described in section 8.2 and its effects are discussed in section 8.3. The chapter is wrapped up with a short summary in section 8.4.

## 8.1 Design Goal and Approach

A non-deterministic PRNG needs to collect non-deterministic and/or true random sources and use them as entropy inputs. What entropies to collect and how to use them are two basic design questions that have to be answered. Here we opt to use generally available entropies only, since any dedicated device will increase the cost and limit the

application of our PRNGs. We have a couple of choices: user interactions with the machine [34,111]; hard drive latency [112]; disk timings, interrupt timings, CPU cycle count, and jiffies count [34]; number of threads/processes, memory/disk utilization, and other system information. What actually to choose depends on the design goals and may not be unique.

Randomness, performance, and security (in the case of a cryptographic PRNG) are important factors that should be considered in the design of a PRNG. Since entropy inputs are usually independent of a PRNG, their introduction will not impact the randomness and security of the generator in a negative way. Using the language of information theory, the addition of two independent entropies cannot be less than any of them. Therefore, if we are satisfied with the randomness and security that our deterministic PRNGs have already provided, we can only focus on the non-deterministic feature here. When randomness and security are not a concern, we do not need to choose as many entropy inputs as other PRNGs do. We do not need to use special entropy accumulation, evaluation, processing, and distribution methods like in other generators [34, 35] either.

Our design goal is to introduce non-deterministic feature into our PRNGs without affecting their availability and performance. For this goal, we decide to choose CPU cycle count as the only entropy input. Nowadays most processors have a register, usually 64 bits, for storing CPU cycles. This register can be accessed from any program, not like most other entropy sources mentioned above, which can only be accessed from the kernel of the operating system. It keeps on changing at a relatively high rate. This is different from user inputs and other entropy sources, which change slowly and sometimes may be even not available. In addition, the cost to add this entropy source is trivial, since there is a single instruction to read the CPU cycle count on most platforms. Last, it is very difficult to manipulate or predict the CPU cycle count, because the value is affected by all the processes running on the processor.

## 8.2 Algorithm Modification

In this section, we modify MaD0 and MaD3 and turn them into non-deterministic PRNGs. To distinguish the modified PRNGs from the original ones, we will refer to the modified MaD0 and MaD3 as MaD0-nd and MaD3-nd respectively. The modification is limited to the pseudorandom generation algorithm. Specifically, we get the CPU cycle count at the beginning of each generation round, preprocess the count value, and then use the result to modify integers $a$, $b$, $c$, and $d$.

The modified MaD0 pseudorandom generation algorithm is shown in Listing 8.1. The CPU cycle count is first read into the 64-bit integer *ta*. On Intel, AMD, and some other platforms this can be done using a single instruction. Then some simple preprocessing is performed, which basically mixes the lower bytes of the CPU cycle count into the higher bytes. The lower bytes change faster than the higher ones, so this step helps diffuse the change. Although we can alternatively use a hash function to achieve a better mixing effect here, it will substantially slow down the pseudorandom number generation. Finally, the processed count value is used to modify integers $c$ and $d$, and from there to further affect integers $a$ and $b$, the internal state, and the pseudorandom output.

The pseudorandom generation algorithm of MaD3 is modified in a similar fashion and given in Listing 8.2. The CPU cycle count is processed right before the original reseed function of MaD3. Functionally the algorithm now contains two reseed functions: the newly added non-deterministic reseed function and the original deterministic reseed function. Since these two reseed functions are executed one after another, they can also be combined into a single reseed function, which makes the functional model shown in Figure 7.1 still valid for MaD3-nd.

Listing 8.1: [MaD0-nd] One Round of Pseudorandom Number Generation

```
# read CPU cycle count
ta = readCCC();

# preprocess the cycle count
ta = ta + (ta << 7);
ta = ta + (ta << 19);
ta = ta + (ta << 37);

# use the preprocessed value to modify c and d
c = c ^ ta;
d = d ^ ta;

# continue with the original algorithm
ta = a = a + c
tb = b = b + d
for i = 0 to 31
    T[2i] = c = c ^ (S[i] + a)
    c = c + (ta ^ tb)
    d = d ^ (c + b)
    ta = ta <<< 3
    T[2i+1] = d = d + (ta ^ tb)
    S[i] = d
    tb = tb >>> 5
endfor
```

Listing 8.2: [MaD3-nd] One Round of Pseudorandom Number Generation

```
# read CPU cycle count
e = readCCC();

# preprocess the cycle count
e = e + (e << 7);
e = e + (e << 19);
e = e + (e << 37);

# use the preprocessed value to modify a, b, c, and d
a = a ^ e;
b = b ^ e;
c = c ^ e;
d = d ^ e;

# continue with the reseed function in MaD3 PRGA
```

## 8.3 Overall Effects and Non-Deterministic Feature

In this section, we first briefly summarize the effects of the modification we made in previous section and then test the non-deterministic feature of the modified PRNGs.

### 8.3.1 Overall Effects of the Modification

The overall effects of the modification are summarized in Table 8.1. Because the newly introduced entropy input, CPU cycle count, can be reasonably assumed to be independent of the original PRNG, its introduction should increase the overall entropy of the generator and therefore improve the generator's randomness, security, and period. That is, the impact on these properties should be positive. As we mentioned before, however, it is not our design goal to improve these properties. The improvement comes as a byproduct and we will not try to analyze or quantitatively measure it. The bottom line is that the new PRNGs are no worse than their deterministic counterparts in terms of those properties. Obviously, the introduction of the new entropy input has some negative impact on the performance. Nonetheless, the magnitude of the impact is far from enough to be a concern – we are unable to reliably measure the trivial performance differences between the modified generators and the original ones. The modified PRNGs maintain the general availability, ease of use, and low cost of the original PRNGs. For the sake of completeness, we have also listed the non-deterministic feature in the table. It is what we try to achieve in this chapter and will be further investigated in the next subsection.

### 8.3.2 Non-Deterministic Feature

We evaluate the non-deterministic feature of MaD0-nd and MaD3-nd in this subsection. For each generator, we do the following:

1. Measure the differences between an output sequence from the original generator and another output sequence from its non-deterministic counterpart, both initialized with

Table 8.1: [MaD0-nd & MaD3-nd] Overall Effects of the Modification

| Property | Impact |
|---|---|
| Randomness | Positive |
| Security | Positive |
| Period | Positive |
| Performance | Negative |
| Availability | Same |
| Ease of use | Same |
| Cost | Same |
| Non-deterministic feature | Added |

the same key. The approach is to XOR these two output sequences to produce a new sequence, which reflects the differences between the two output sequences. This new sequence, denoted as d-nd, is then tested using TestU01 batteries of tests.

2. Generate another sequence, which reflects the differences between two output sequences from two instances of the non-deterministic generator, initialized with the same key. This sequence, denoted as nd-nd, is then tested using TestU01 batteries of tests.

Figures 8.1 (a) and (b) illustrate the generation of sequence d-nd and sequence nd-nd respectively for MaD0-nd. The testing results are given in Table 8.2. Both MaD0-nd and MaD3-nd demonstrate excellent non-deterministic feature. Each generator can generate a completely different and statistically independent pseudorandom sequence during each run, irrespective of the initial seed or key. This is like a true random number generator. Obviously, the new non-deterministic feature also fortifies MaD3's prediction resistance. Even the entire internal state is compromised at a point in time, MaD3-nd will still be able to recover and come back to normal quickly.

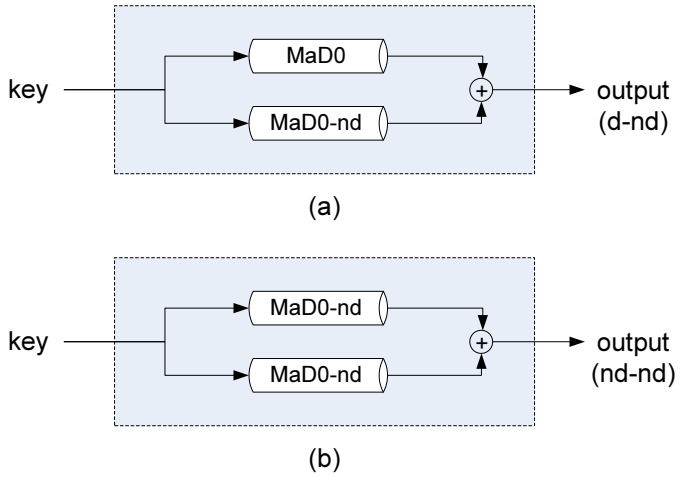



Figure 8.1: [MaD0-nd] Non-Deterministic Feature Test

Table 8.2: [MaD0-nd & MaD3-nd] Statistical Testing Results (TestU01)

| Battery | Parameters | Tests | NoP | Failures | | | |
|---|---|---|---|---|---|---|---|
| | | | | MaD0-nd | | MaD3-nd | |
| | | | | d-nd | nd-nd | d-nd | nd-nd |
| SmallCrush | Built-in | 10 | 15 | 0 | 0 | 0 | 0 |
| Crush | Built-in | 96 | 144 | 0 | 0 | 0 | 0 |
| BigCrush | Built-in | 106 | 160 | 0 | 0 | 0 | 0 |
| Rabbit | $32 \times 10^9$ bits | 26 | 40 | 0 | 0 | 0 | 0 |
| Alphabit | $32 \times 10^9$ bits | 9 | 17 | 0 | 0 | 0 | 0 |
| BlockAlphabit | $32 \times 10^9$ bits | $6 \times 9$ | 102 | 0 | 0 | 0 | 0 |

## 8.4 Conclusion

A simple non-deterministic pseudorandom number generation scheme is proposed and used to convert MaD0 and MaD3 into non-deterministic PRNGs in this chapter. The new PRNGs demonstrate very good non-deterministic feature and behave like a true random number generator. Properties of their deterministic counterparts, such as high performance, excellent randomness, long period, general availability, ease of use, low cost, and, in the case of MaD3, security, are all preserved. While they still belong to the category of PRNGs, they are good for many applications where TRNGs are otherwise needed.

# Chapter 9

# Conclusions

This thesis presents a new design paradigm, which can be used to design high speed and high quality pseudorandom number generators. At the core of this paradigm is a two-layer approach, which closely combines a slow byte-oriented layer for initialization and reseeding and a fast integer-oriented layer for state transition and pseudorandom number generation. The first layer maintains a small pseudorandom permutation table and the second layer maintains a large pseudorandom mapping table. The first layer is initialized with a seed or key and the second layer is bootstrapped through the first layer. After initialization, the first layer may remain as a separate layer or be mixed into the second layer. The slow-start strategy of the first layer ensures high quality initialization, which shows to have an avalanche effect comparable to that of standard hash functions. The second layer takes the advantages of modern 64-bit platforms and uses integer operations for high speed pseudorandom number generation.

Several PRNGs are designed using the above approach and presented in this thesis. They are several times faster than other popular PRNGs currently in use when tested on a 64-bit Intel Core i3 platform. The high quality of the generated pseudorandom numbers is verified by several well-known statistical testing tools. The proposed non-cryptographic PRNG also demonstrates other desirable properties such as long period, high-dimensional equidistribution, fast recovery from biased states, and ease of use. The proposed crypto-

graphically secure PRNG is designed to resist various cryptanalytic attacks and to meet the requirement of withstanding state compromise extensions.

Besides deterministic pseudorandom number generation, a simple non-deterministic pseudorandom number generation scheme is also proposed. This scheme enables our PRNGs to generate pseudorandom numbers like a true random number generator at almost no additional cost. Most applications currently relying on true random number generators can benefit from this scheme.

# Appendix A

# Test Vectors

The first 64 bytes of pseudorandom output for different keys are provided for each PRNG in this appendix.

1. The key is set to number 0

   **MARC**

   ```
   029aa08d  74643f19  7e7d3ac5  4cd142af  1567755f  a8aa13d3  87e0dfe0  fc9a6dee
   f56d657a  b1f84cd8  e95dd274  4e0d8e04  f9f5cb25  8a3f237f  a5c54a8c  1612e298
   ```

   **MaD0**

   ```
   4f24db01  b7a0771e  e5071685  1ce25ed0  c5dbe467  04c9ef13  8b0c7fe2  eaeacf45
   95bc7de7  60c45a04  dedd23cc  d8458da3  fc2a4b46  ca388f53  4308c0c8  f24bdf81
   ```

   **MaD3**

   ```
   bb43fed0  c47752d1  361c8a57  82bf55c2  a0ac38e2  2e691240  fc2e5f46  2e178717
   9773ec88  18970bb0  13e4a967  792f3f70  80da358b  8fe7820f  cc46b4c1  7c429860
   ```

2. The key is set to string “0” (number 0x30)

   **MARC**

   ```
   76ecb358  8f244922  017c30fb  cd8c9f3b  3fb77af3  03d505df  1305750a  aec888b0
   b24e1600  89148891  f904431e  f2ffd709  d1dde89a  66317294  d10778a0  318d2ce1
   ```

**MaD0**

```
c52e9854  bc082a9c  e55ddb46  bd49bd3e  f5bf890a  2348b48e  be59871c  acf29878
47a18780  68367e3a  d98089cd  2e06eae2  5b56e51f  a119e21e  4315e0f8  6654bd9a
```

**MaD3**

```
db3fee64  25815bf5  5f1baa2b  044eff72  ffdbbb88  32114406  69a7f5c2  f08bcd0d
bd84bfc8  0895c05c  d730b048  5136827a  f1d25635  24d73050  fa082a6a  17d0da96
```

# Bibliography


[1] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[2] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[3] F. Panneton, P. L'ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 1, pp. 1–16, 2006.

[4] M. Saito and M. Matsumoto, "SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pp. 607–622, Springer, 2008.

[5] National Institute of Standards and Technology, *Advanced encryption standard (AES)*, 2001. Federal Information Processing Standards Publication 197.

[6] B. Schneier, *Applied cryptography. Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc, 1996.

[7] H. Wu, "The stream cipher HC-128," in *New Stream Cipher Designs*, pp. 39–47, Springer, 2008.

[8] M. Boesgaard, M. Vesterager, T. Christensen, and E. Zenner, "The stream cipher Rabbit," *ECRYPT Stream Cipher Project Report*, vol. 6, 2005.

[9] D. J. Bernstein, "Salsa20/8 and Salsa20/12," *eSTREAM, ECRYPT Stream Cipher Project*, 2006.

[10] C. Berbain, O. Billet, A. Canteaut, *et al.*, "Sosemanuk, a fast software-oriented stream cipher," in *New Stream Cipher Designs*, pp. 98–118, Springer, 2008.

[11] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2010.

[12] P. L'ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Mathematics of Computation of the American Mathematical Society*, vol. 68, no. 225, pp. 249–260, 1999.

[13] P. L'Ecuyer and R. Simard, "Testu01: A C library for empirical testing of random number generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 22, 2007.

[14] G. S. Fishman, "Multiplicative congruential random number generators with modulus $2^{\beta}$: an exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$," *Mathematics of Computation*, vol. 54, no. 189, pp. 331–344, 1990.

[15] P. L'Ecuyer, F. Blouin, and R. Couture, "A search for good multiple recursive random number generators," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 3, no. 2, pp. 87–98, 1993.

[16] I. G. Dyadkin and K. G. Hamilton, "A study of 64-bit multipliers for lehmer pseudorandom number generators," *Computer Physics Communications*, vol. 103, no. 2, pp. 103–130, 1997.

[17] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Third Edition*, vol. 2. Addison Wesley Longman, 1998.

[18] P. L'ecuyer, "Good parameters and implementations for combined multiple recursive random number generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.

[19] P. L'Ecuyer and R. Touzin, "Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$," in *Proceedings of the 32nd conference on Winter simulation*, pp. 683–689, Society for Computer Simulation International, 2000.

[20] L.-Y. Deng, "Efficient and portable multiple recursive generators of large order," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 15, no. 1, pp. 1–13, 2005.

[21] G. Marsaglia, "A current view of random number generators," in *Computer Science and Statistics, Sixteenth Symposium on the Interface. Elsevier Science Publishers, North-Holland, Amsterdam*, pp. 3–10, 1985.

[22] R. P. Brent, "On the periods of generalized fibonacci recurrences," *Mathematics of Computation*, vol. 63, no. 207, pp. 389–401, 1994.

[23] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: A scalable library for pseudorandom number generation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 3, pp. 436–461, 2000.

[24] M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. Robinson, "A fast, high quality, and reproducible parallel lagged-fibonacci pseudorandom number generator," *Journal of Computational Physics*, vol. 119, no. 2, pp. 211–219, 1995.

[25] F. Panneton and P. L'ecuyer, "On the xorshift random number generators," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 15, no. 4, pp. 346–361, 2005.

[26] R. P. Brent, "Some long-period random number generators using shifts and xors," *ANZIAM Journal*, vol. 48, pp. C188–C202, 2007.

[27] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM Journal on computing*, vol. 15, no. 2, pp. 364–383, 1986.

[28] E. Barker and J. Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. National Institute of Standards and Technology, 2012. NIST Special Publication 800-90A.

[29] A. C. Yao, "Theory and application of trapdoor functions," in *Foundations of Computer Science, 1982, 23rd Annual Symposium on Foundations of Computer Science*, pp. 80–91, IEEE, 1982.

[30] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby, "A pseudorandom generator from any one-way function," *SIAM Journal on Computing*, vol. 28, no. 4, pp. 1364–1396, 1999.

[31] American National Standards Institute, *ANSI X9.17: Financial Institution Key Management (wholesale)*, 1985.

[32] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, 2000. Federal Information Processing Standards Publication 186-2.

[33] National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, 2008. Federal Information Processing Standards Publication 180-3.

[34] P. Lacharme, A. Röck, V. Strubel, and M. Videau, "The linux pseudorandom number generator revisited," *IACR Cryptology ePrint Archive*, vol. 2012, p. 251, 2012.

[35] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator," in *Selected Areas in Cryptography*, pp. 13–33, Springer, 2000.

[36] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural, and medical research*. Hafner Pub. Co.(New York), 1970.

[37] R. Durstenfeld, "Algorithm 235: random permutation," *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.

[38] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC Press, 2008.

[39] M. Luby and C. Rackoff, "How to construct pseudorandom permutations from pseudorandom functions," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 373–386, 1988.

[40] S. Moriai and S. Vaudenay, "Comparison of randomness provided by several schemes for block ciphers," *Preprint*, 1999. Available at http://csrc.nist.gov/archive/aes/round2/conf3/papers/34-smoriai.pdf.

[41] I. Mironov, "(Not so) random shuffles of RC4," in *Advances in Cryptology–CRYPTO 2002*, vol. 2442 of *Lecture Notes in Computer Science*, pp. 304–319, Springer, 2002.

[42] V. F. Kolchin, *Random mappings*. Optimization Software, Incorporated, Publications Division, 1986.

[43] P. Flajolet and A. M. Odlyzko, "Random mapping statistics," in *Advances in cryptology–EUROCRYPT'89*, vol. 434 of *Lecture Notes in Computer Science*, pp. 329–354, Springer, 1990.

[44] J. Li and J. Zheng, "MaD2: an ultra-performance stream cipher for pervasive data encryption," in *Foundations and Practice of Security*, vol. 7743 of *Lecture Notes in Computer Science*, pp. 1–17, Springer, 2013.

[45] D. Kahn, *The codebreakers: the story of secret writing*. Macmillan, 1967.

[46] National Institute of Standards and Technology, *Data Encryption Standard*, 1993. Federal Information Processing Standards Publication 46-2.

[47] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, *Recommendation for key management–part 1: General (revision 3)*. National Institute of Standards and Technology, 2011. NIST special publication 800-57.

[48] A. Rukhin, J. Soto, J. Nechvatal, *et al.*, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards and Technology, 2001. NIST special publication 800-22.

[49] G. Marsaglia, *The Marsaglia random number CDROM including the Diehard battery of tests of randomness*, 1995. New version available at http://www.csis.hku.hk/ diehard/.

[50] G. Marsaglia and W. W. Tsang, "Some difficult-to-pass tests of randomness," *Journal of Statistical Software*, vol. 7, no. 3, pp. 1–9, 2002.

[51] B. D. McCullough, "Assessing the reliability of statistical software: Part I," *The American Statistician*, vol. 52, no. 4, pp. 358–366, 1998.

[52] J. E. Gentle, *Random number generation and Monte Carlo methods*. Springer, 2003.

[53] A. Webster and S. E. Tavares, "On the design of S-boxes," in *Advances in Cryptology–CRYPTO'85 Proceedings*, vol. 218 of *Lecture Notes in Computer Science*, pp. 523–534, Springer, 1986.

[54] M. J. Robshaw, "Stream ciphers," *RSA Data Security, Inc*, 1995.

[55] S. Mister and S. E. Tavares, "Cryptanalysis of RC4-like ciphers," in *Selected Areas in Cryptography*, vol. 1556 of *Lecture Notes in Computer Science*, pp. 131–143, Springer, 1998.

[56] E. Dawson, H. Gustafson, M. Henricksen, and B. Millan, *Evaluation of RC4 Stream Cipher*. Information Security Research Centre Queensland University of Technology, 2002.

[57] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4," in *Selected areas in cryptography*, pp. 1–24, Springer, 2001.

[58] A. Roos, "A class of weak keys in the RC4 stream cipher," 1995. Posting to sci.crypt.

[59] J. D. Golić, "Linear statistical weakness of alleged RC4 keystream generator," in *Advances in Cryptology–EUROCRYPT'97*, vol. 1233 of *Lecture Notes in Computer Science*, pp. 226–238, Springer, 1997.

[60] A. Klein, "Attacks on the RC4 stream cipher," *Designs, Codes and Cryptography*, vol. 48, no. 3, pp. 269–286, 2008.

[61] E. Tews, R.-P. Weinmann, and A. Pyshkin, "Breaking 104 bit WEP in less than 60 seconds," in *Information Security Applications*, vol. 4867 of *Lecture Notes in Computer Science*, pp. 188–202, Springer, 2007.

[62] R. Rivest, *RSA security Response to weaknesses in key scheduling algorithm of RC4*. RSA Data Security, Inc, 2001.

[63] B. Harris, "RFC 4345: Improved arcfour modes for the secure shell (SSH) transport layer protocol," 2006. Available at http://tools.ietf.org/html/rfc434.

[64] F. Leitner, "Source code optimization," 2009. Available at http://www.linux-kongress.org/2009/slides/compiler_survey_felix_von_leitner.pdf.

[65] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. National Bureau of Standards Applied Mathematics Series 55, Tenth Printing, ERIC, 1972.

[66] C. Lomont, "Random number generation," in *Games Programming Gems 7, Course Technology*, 2008. Available at http://www.lomont.org/Math/Papers/2008/Lomont_PRNG_2008.pdf.

[67] K. Chen, M. Henricksen, W. Millan, *et al.*, "Dragon: A fast word based stream cipher," in *Information Security and Cryptology–ICISC 2004*, vol. 3506 of *Lecture Notes in Computer Science*, pp. 33–50, Springer, 2005.

[68] R. J. Jenkins Jr, "ISAAC," in *Fast Software Encryption*, pp. 41–49, Springer, 1996.

[69] IBM, "System/360 scientific subroutine package, version III, programmer's manual," 1968.

[70] M. Matsumoto, M. Saito, H. Haramoto, and T. Nishimura, "Pseudorandom number generation: Impossibility and compromise.," *J. UCS*, vol. 12, no. 6, pp. 672–690, 2006.

[71] A. Fog, *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Copenhagen University College of Engineering, 2011. Available at http://www.agner.org/optimize/instruction_tables.pdf.

[72] A. Biryukov and A. Shamir, "Cryptanalytic time/memory/data tradeoffs for stream ciphers," in *Advances in Cryptology–ASIACRYPT 2000*, vol. 1976 of *Lecture Notes in Computer Science*, pp. 1–13, Springer, 2000.

[73] J.-C. Faugere and A. Joux, "Algebraic cryptanalysis of hidden field equation (HFE) cryptosystems using Gröbner bases," in *Advances in Cryptology-CRYPTO 2003*, pp. 44–60, Springer, 2003.

[74] N. T. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations," in *Advances in Cryptology–ASIACRYPT 2002*, pp. 267–287, Springer, 2002.

[75] N. T. Courtois, "Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt," in *Information security and cryptology–ICISC 2002*, vol. 2587 of *Lecture Notes in Computer Science*, pp. 182–199, Springer, 2003.

[76] F. Armknecht, "A linearization attack on the Bluetooth key stream generator," 2002. IACR eprint server, http://www.iacr.org.

[77] O. Billet and H. Gilbert, "Resistance of SNOW 2.0 against algebraic attacks," in *Topics in Cryptology–CT-RSA 2005*, vol. 3376 of *Lecture Notes in Computer Science*, pp. 19–28, Springer, 2005.

[78] F. Armknecht, "Algebraic attacks on stream ciphers," in *Fourth European Congress on Computational Methods in Applied Sciences and Engineering, Finland*, pp. 24–28, 2004.

[79] W. Meier, E. Pasalic, and C. Carlet, "Algebraic attacks and decomposition of boolean functions," in *Advances in Cryptology-EUROCRYPT 2004*, vol. 3027 of *Lecture Notes in Computer Science*, pp. 474–491, Springer, 2004.

[80] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[81] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of symbolic computation*, vol. 9, no. 3, pp. 251–280, 1990.

[82] V. V. Williams, "Breaking the Coppersmith-Winograd barrier," *Unpublished manuscript*, 2011. Available at http://www.cs.berkeley.edu/ virgi/matrixmult.pdf.

[83] B. Buchberger, "A criterion for detecting unnecessary reductions in the construction of Gröbner-bases," in *Symbolic and Algebraic Computation*, vol. 72 of *Lecture Notes in Computer Science*, pp. 3–21, Springer, 1979.

[84] N. T. Courtois, "Fast algebraic attacks on stream ciphers with linear feedback," in *Advances in Cryptology-CRYPTO 2003*, vol. 2729 of *Lecture Notes in Computer Science*, pp. 176–194, Springer, 2003.

[85] K. K.-H. Wong, G. Carter, and E. Dawson, "An analysis of the RC4 family of stream ciphers against algebraic attacks," in *Proceedings of the Eighth Australasian Conference on Information Security-Volume 105*, pp. 67–74, Australian Computer Society, Inc., 2010.

[86] E. Mehrabi, A. Sharifi, P. Abdollahifard, and A. Montazeri, "Guess-and-determine attack and algebraic attack," *International Journal of Algebra*, vol. 4, no. 12, pp. 551–560, 2010.

[87] D. Coppersmith, S. Halevi, and C. Jutla, "Cryptanalysis of stream ciphers with linear masking," in *Advances in Cryptology–CRYPTO 2002*, vol. 2442 of *Lecture Notes in Computer Science*, pp. 515–532, Springer, 2002.

[88] D. Watanabe, A. Biryukov, and C. De Canniere, "A distinguishing attack of SNOW 2.0 with linear masking method," in *Selected Areas in Cryptography*, vol. 3006 of *Lecture Notes in Computer Science*, pp. 222–233, Springer, 2004.

[89] K. Nyberg and J. Wallén, "Improved linear distinguishers for SNOW 2.0," in *Fast Software Encryption*, vol. 4047 of *Lecture Notes in Computer Science*, pp. 144–162, Springer, 2006.

[90] J.-K. Lee, D. H. Lee, and S. Park, "Cryptanalysis of SOSEMANUK and SNOW 2.0 using linear masks," in *Advances in Cryptology-ASIACRYPT 2008*, vol. 5350 of *Lecture Notes in Computer Science*, pp. 524–538, Springer, 2008.

[91] A. L. Grosul and D. S. Wallach, "A related-key cryptanalysis of RC4," Tech. Rep. TR-00-358, Department of Computer Science, Rice University, 2000.

[92] I. Mantin, "A practical attack on the fixed RC4 in the WEP mode," in *Advances in Cryptology-ASIACRYPT 2005*, vol. 3788 of *Lecture Notes in Computer Science*, pp. 395–411, Springer, 2005.

[93] E. Biham and O. Dunkelman, "Differential cryptanalysis in stream ciphers.," *IACR Cryptology ePrint Archive*, vol. 2007, p. 218, 2007.

[94] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[95] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pp. 312–320, ACM, 2007.

[96] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Topics in Cryptology–CT-RSA 2007*, vol. 4377 of *Lecture Notes in Computer Science*, pp. 225–242, Springer, 2006.

[97] D. J. Bernstein, "Cache-timing attacks on AES," 2005. Available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[98] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo, "AES power attack based on induced cache miss and countermeasure," in *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, vol. 1, pp. 586–591, IEEE, 2005.

[99] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology–CT-RSA 2006*, pp. 1–20, Springer, 2006.

[100] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems-CHES 2006*, vol. 4249 of *Lecture Notes in Computer Science*, pp. 201–215, Springer, 2006.

[101] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[102] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, vol. 6225 of *Lecture Notes in Computer Science*, pp. 110–124, Springer, 2010.

[103] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *Advances in Cryptology–ASIACRYPT 2009*, vol. 5912 of *Lecture Notes in Computer Science*, pp. 667–684, Springer, 2009.

[104] T. Chardin, P.-A. Fouque, and D. Leresteux, "Cache timing analysis of RC4," in *Applied Cryptography and Network Security*, vol. 6715 of *Lecture Notes in Computer Science*, pp. 110–129, Springer, 2011.

[105] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[106] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudo random bits," in *Foundations of Computer Science, 1982, 23rd Annual Symposium on Foundations of Computer Science*, pp. 112–117, IEEE, 1982.

[107] A. Schrift and A. Shamir, "Universal tests for nonuniform distributions," *Journal of Cryptology*, vol. 6, no. 3, pp. 119–133, 1993.

[108] B. Sadeghiyan and J. Mohajeri, "A new universal test for bit strings," in *Information Security and Privacy*, vol. 1172 of *Lecture Notes in Computer Science*, pp. 311–319, Springer, 1996.

[109] J. Hernandez, J. Sierra, C. Mex-Perera, D. Borrajo, A. Ribagorda, and P. Isasi, "Using the general next bit predictor like an evaluation criteria," *proceedings of NESSIE workshop*, 2000.

[110] A. Lavasani and T. Eghlidos, "Practical next bit test for evaluating pseudorandom sequences," *Electrical Engineering*, vol. 16, no. 1, pp. 19–33, 2009.

[111] P. R. Zimmermann, *The official PGP user's guide*. MIT press, 1995.

[112] D. Davis, R. Ihaka, and P. Fenstermacher, "Cryptographic randomness from air turbulence in disk drives," in *Advances in Cryptology–Crypto'94*, pp. 114–120, Springer, 1994.