JAIST Repository
https://dspace.jaist.ac.jp/

Title	Robust and Cryptographically Secure Pseudo-Random Bit Generation
Author(s)	Mpho, Tjabane
Citation	
Issue Date	2003-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1703
Rights	
Description	Supervisor:Hong Shen, 情報科学研究科, 修士

JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Robust and Cryptographically Secure Pseudo-Random Bit Generation

By Mpho Tjabane

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Prof. Hong Shen

March, 2003

Robust and Cryptographically Secure Pseudo-Random Bit Generation

By Mpho Tjabane (110060)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Prof. Hong Shen

and approved by
Prof. Hong Shen
Prof. Susumu Horiguchi
Prof. Koji Nakano

February, 2003 (Submitted)

Copyright © 2003 by Mpho Tjabane

I declare that this thesis is my own, unaided work. It is being submitted for the Degree of Master of Science in the Japan Advanced Institute of Science and Technology (JAIST), Ishikawa, Japan. It has not been submitted before for any degree or examination in any other University.

____
(Signature of Candidate)

_________ day of ____________________ 2003.

Abstract

A *new* two-stage combination bit generator, named TM3w, that has cryptographic features is constructed, analyzed, and implemented. The first stage of the generator consists of sub-generators formed by combining outputs of *three* lagged Fibonacci recurrences using mixed operators that do not satisfy the distributive and associative rules of group algebra. The Fibonacci recurrences are defined by the primitive trinomials of the form $x^{r_i}+x^{s_i}+1\ mod\ 2$, with $1 \leq s_i \leq r_i-1$. Values for r_i are chosen such that $\forall\ i \neq j$ and $j \neg k$, $r_i > r_j > r_k$ and $gcd(r_i, r_j) = gcd(r_j, r_k) = 1$. The second stage concatenates outputs of the sub-generators to give a combined output for the entire generator. For a computer whose wordlength is w, the generator outputs $3 \times w$ bits at every iteration. The initial states of the Fibonacci recurrences are derived from a seed vector *and* from each other by a scheme that performs bit rotation, bit omission, and incremental modulo addition operations. This scheme is efficient in attaining cryptographic properties of *Confusion* and *Diffusion*. The combination generator passes all statistical tests for randomness to which it is subjected. Implementation of this generator is done for $w = 32$ on a 64-bit processor using multithreaded programming with the aim being to attain concurrency on a uniprocessor machine. The application programming interfaces used are those of the POSIX threads library; other thread libraries exist. Consequently the implementation is portable to machines with POSIX-compliant operating systems only. The threads performing computations use *named* UNIX pipes called FIFOs (first in, first out) for interprocess communication. The unresolvable problems of contention for mutexes by threads due to the absence of a guranteed order in which mutex locks are acquired; program deadlock due to threads trying to acquire mutex locks that they are *already* holding; and the absence of time-slicing mechanism in the POSIX API, complicate the task of attaining concurrency on a uniprocessor quite significantly. The author recommends additional features to the POSIX API to address the aforementioned problems. This combination generator attains a minimum period in the order of 10^{51} and the maximum period in the order of 10^{87}, sufficient for most cryptographic applications. Lastly, no known attack is computationally feasible against the TM3w generator.

In memory of my beloved brother

Teboho Henry Tjabane
1974 - 2000

Contents

1 A Review of Current Random Number Generators **1**
1.1 Congruential Generators 1
1.1.1 Linear Congruential Generators 1
1.1.2 Polynomial Congruential Generators 2
1.2 Feedback Shift Registers 2
1.3 Lagged Fibonacci Generators 2
1.4 Combination Generators 3
1.4.1 Combining by Shuffling 3
1.4.2 Bit mixing 3
1.4.3 Combining by Shrinking 4

2 The TM3w Bit Generator **5**
2.1 Fibonacci Generators 5
2.1.1 Primitive Polynomials 5
2.1.2 Recurrences 6
2.2 Creating Generator Initial States 7
2.2.1 Test Input Data 8
2.3 Combination Generators 10
2.4 Algorithm Analysis 11
2.4.1 Initialization Phase 11
2.4.2 Recurrences 12
2.4.3 Combination Generators 13
2.5 Lags and Periods 13

3 Bitwise Behaviour of the TM3w Generator **15**
3.1 Outputs and Randomness 15
3.1.1 Frequency Test 17
3.1.2 Serial Test 17
3.1.3 Poker Test 17
3.1.4 Runs Test 18
3.1.5 Autocorrelation Test 18
3.2 Randomness tests results 18
3.3 Vector Components Correlations 19

4 The Security of TM3w **21**
4.1 Generating Functions 21
4.1.1 Recurrences and Generating Functions 22
4.1.2 Extracting coefficients 23
4.2 Mixed Operators 24
4.3 Other Primitive Trinomials 25
4.4 TM3w Initialization 26
4.5 Combination Generators 27

5 Implementing the Algorithm **29**
5.1 User Interface 29
5.1.1 Password to Seed Vector 29
5.2 Initial States 33
5.3 Fibonacci Recurrences 37
5.4 Combination Generators 38

6 TM3w Program Concurrency **40**
6.1 Shared Address Space 40
6.2 Producer-Consumer Problem 41
6.2.1 Producer and Consumer Threads 43
6.3 Writing and Reading a FIFO 45
6.4 Synchronization 48
6.4.1 Producer to Consumer 48
6.4.2 Consumer to Consumer 48
6.5 Unresolved Problems 49

7 Portability of the Implementation **51**
7.1 Hardware 51
7.2 Operating System 52
7.3 Compiler 53

A Test Distributions **55**

Bibliography **55**

List of Figures

5.1 Program execution sequence 30
5.2 String of 32 bits 32

6.1 Accessing shared data 41
6.2 Producer-Consumer problem 42
6.3 Threads interleaving execution on a uniprocessor 49

7.1 Byte order for a 32-bit integer 52

List of Tables

2.1 Precomputing inputs 11
2.2 Computation steps 12
2.3 Periods of Fibonacci recurrences 13
2.4 Periods of Combination generators 14
2.5 Minimum and Maximum periods 14

3.1 Fibonacci generators sample output 16
3.2 Combination generators sample output 16
3.3 Randomness tests results 19
3.4 Pairwise correlations coefficients 20

Acknowledgements

First and foremost, I thank the LORD for keeping me in good health and out of harms way everyday of my life. I thank my wonderful mother, Faith Tjabane, whose love, support, and faith in me, give me strength to persevere in any situation.

This thesis was made possible by the involvement of a number of people, some more involved than others. All are worthy of mention.
I would like to thank my supervisor, Prof. Hong Shen, for the guidance and support during the course of my research. Special thanks to Prof. Yoichi Shinoda for guiding me through *named pipes* and a maze of *threads*. Thanks to Dr. Masakazu Soshi for the conversations and many suggestions during our informal meetings. Thanks also to Dr. Jiang Xiaohong with whom I had many fruitful discussions on polynomials and randomness.
Thanks also to the JAIST Library Staff who delivered on all of my many requests for journals and books, some long out of print. I am also deeply appreciative of the International Student Section Staff who take the paper-work out of our lives. Last but not least, thanks to all the fine friends, near and far, for their support. Special thanks to Hafizur Rahman and Carlos Troncoso for being my "ears" during my entire tenure.

Chapter 1

A Review of Current Random Number Generators

To quote Marsaglia: "...a pseudo random number generator (RNG) is much like sex, when it's good it's wonderful, and when it's bad it's still pretty good"[20]. Sadly many of the traditional random number generators are not good enough especially when it comes to cryptographic applications such as digital signature schemes, session key generation by key distribution centers or principals in a network, reciprocal authentication schemes as in TCP's three-way handshake, and the generation of large prime numbers. In general, random number generators produce a sequence of elements by means of a linear or non-linear transformation on some algebraic structure.
Necessary requirements placed on RNG's in order to ensure that their output is random. Two criteria that are used to validate that a sequence of numbers is random are *uniform distribution* and *statistical independence*. The first criterion means that the frequency of occurence of each of the numbers in a sequence should be approximately the same. The second criterion means that values forming a sequence should not be somehow correlated. Well-defined tests exist for determining that a sequence of numbers matches a particular statistical distribution such as a uniform distribution. However there are no such tests to determine statistical independence. Rather, a number of tests can be applied to demonstrate that a sequence does not exhibit dependence. Such tests are applied until the confidence that independence exists is sufficiently strong.
A number of methods for generating random sequences have been developed over the years. The methods which have been dominant are outlined in the following sections.

1.1 Congruential Generators

1.1.1 Linear Congruential Generators

A pseudorandom number generator that has received much attention is the linear congruential generator which was introduced by D.H. Lemmer in 1949[17]. This generator uses a transformation

$$X_{n+1} = (aX_n + c)\ mod\ m,\ n \geq 0$$

where $m > 0$ is called the modulus, $0 \leq a < m$ is the multiplier, $0 \leq c < m$ is the increment, and $0 \leq X_0 < m$ is the starting value. This generator is called a maximal period generator if its period is m. This type of generator was broken by, among others, Joan Boyar[5] and thus cannot be used for cryptography as it is predictable.

1.1.2 Polynomial Congruential Generators

Linear congruential generators have been generalized to quadratic generators

$$X_n = (aX_{n-1}^2 + bX_{n-1} + c)\ mod\ m$$

and cubic generators

$$X_n = (aX_{n-1}^3 + bX_{n-1}^2 + cX_{n-1} + d)\ mod\ m$$

H. Krawczyk[18] and others[13] have extended Boyar's work to break any polynomial congruential generator. The evidence is that congruential generators are not useful for cryptography.

1.2 Feedback Shift Registers

A feedback shift register is made up of two parts: a shift register and a feedback function. The shift register is a sequence of bits. Each time a bit is needed, all of the bits in the shift register are shifted one bit to the right. The new left-most bit is computed as a function of the other bits in the register. The output of the shift register is one bit. If the number of elements in a binary vector is n then we have an n-bit shift register. The maximum possible period of a such a register is $2^n - 1$.
The feedback function can be linear or nonlinear. When this function is linear, it can simply be the XOR of certain bits in the register, the list of which is called a *tap sequence*. For linear feedback shift registers, sequential bits are linear. Also, large random numbers generated from sequential bits of this sequence are highly correlated. By the Berlekamp-Massey algorithm[21] the feedback function, when not known, can be determined from only $2n$ output bits of the generator. Nonlinear feedback shift registers with sparse feedback polynomials facilitate correlation attacks, and dense feedback polynomials are inefficient. The preceeding considerations make feedback shift registers *per se* weak candidates for fast bit generation.

1.3 Lagged Fibonacci Generators

A Fibonacci series satisfies a linear recurrence

$$F_n = F_{n-1} + F_{n-2}$$

A Fibonacci random number is a generalized recurrence of the form

$$X_n = X_{n-a} \; op \; X_{n-b} \; op \; X_{n-c} \; op \ldots \; op \; X_{n-m} \; mod \; 2^w$$

where *op* is any of addition, subtraction, multiplication, or the *exclusive*-or operations. The values s and r are called lags, hence lagged Fibonacci generator. The initial state of this generator is an array of w-bit words: $X_1, X_2, \ldots, X_m$. This initial state is the key. The period of this generator will be $2^n - 1$ if the coefficients are such that the polynomial

$$x^a + x^b + \ldots + x^m \tag{1.1}$$

is a primitive polynomial[1].

1.4 Combination Generators

There are many possible variations of combining simple generators. These are described in the subsections below.

1.4.1 Combining by Shuffling

Given two pseudo-random sequences $X = (x_0, x_1, \ldots)$ and $Y = (y_0, y_1, \ldots)$, a buffer V of size , say, B is filled using the sequence X. The sequence Y is used to generate indices into the buffer. If the index is j then the generator returns $V[j]$ and replaces $V[j]$ by the next number in the X sequence. In other words, one generator is used to shuffle the output of another generator. However, according to Knuth[17], shuffling methods have an inherent defect in that they change only the order of the generated numbers, not the numbers themselves.

1.4.2 Bit mixing

Here elements of two or more sequences are combined using some logical or arithmetic operation. If, for instance, we have sequences $X = (x_0, x_1, \ldots)$ and $Y = (y_0, y_1, \ldots)$, these can be combined such that elements r_i of the resulting sequence are of the form $r_i = x_i \; op \; y_i$, where *op* is some suitable operation. Empirical studies suggest that combining two or more simple generators in this fashion provides a composite generator with better randomness than the component generators.

Pike

This is a combination generator based on Fibonacci recurrences and created by Ross Anderson[2]. It uses three additive generators whose relations are

$$a_i = a_{i-55} + a_{i-24} \; mod \; 2^{32}$$

[1]Defined later

$$a_i = a_{i-57} + a_{i-7} \ mod\ 2^{32}$$

$$a_i = a_{i-58} + a_{i-19} \ mod\ 2^{32}$$

A keystream word is generated by observing the addition carry bits. If all three agree (all are 0 or all are 1), then all the three generators are clocked. If they do not, only the two generators that agree are clocked. The carry bits are saved for next time. The final output is the XOR of the three generators.

1.4.3 Combining by Shrinking

With this method one sequence is used to "shrink" another sequence. Given two pseudo-random sequences $(x_0, x_1, \ldots)$ and $(y_0, y_1, \ldots)$, $x_i, y_i \in GF(2)$, suppose $y_i = 1$ for $i = s_0, s_1, \ldots$. A sequence $(z_0, z_1, \ldots)$ is defined to be the subsequence $(x_{s_0}, x_{s_1}, \ldots)$ of $(x_0, x_1, \ldots)$. In other words, one sequence of bits (y_i) is used to decide whether to "accept" or "reject" elements of another sequence (x_i). Combining two generators by shrinking is slower than combining the sequences by, say, $\oplus$ but is less amenable to mathematical analysis.

Mush

Mush is a mutual shrinking generator which uses the two additive generators

$$a_i = a_{i-55} + a_{i-24} \ mod\ 2^{32}$$

$$a_i = a_{i-52} + a_{i-19} \ mod\ 2^{32}$$

If the carry bit of the first generator is set, the second generator is clocked. If the carry bit of the second genearator is set, the first generator is clocked. The final output is the XOR of of the outputs of the two generaors. To the author's knowledge, this algorithm and Pike have not been broken as yet.

Chapter 2

The TM3w Bit Generator

This chapter is the beginning of the description of a new combination random bit generator designed and implemented by the author. The preceeding letters in the name of this algorithm (TM3w) are the same as those in the author's name. The latter part (3w) is indicative of the fact that this algorithm outputs $3 \times w$ bits, where w is the word length of a particular computer. This means theoretically that the length of the algorithm output is determined by the length of the CPU registers of the hardware on which it is running. For this work a 32-bit computer was used in the implementation. The analysis and statistical tests carried out later are based on the output of 96 bits per algorithm cycle.
It is emphasized that this generator is fundamentally a *cryptographic* algorithm, to distinguish it from some of the traditional generators described in chapter 1. As such, it is important that its outputs be unpredictable to an attacker even when he knows the full details of the algorithm except the initial seed material.
For completeness and correct perspective, the mathematical theory underlying the design of our algorithm is oulined in the beginning subsection of the section below. Sections following the first then go into great details about the workings of the algorithm and its properties.

2.1 Fibonacci Generators

The core component of the combination generator which we propose consists of three additive lagged Fibonacci generators. The lags of these generators are powers of an indeterminate in a trinomial called a primitive trinomial.

2.1.1 Primitive Polynomials

A polynomial $P(x)$ is *irreducible* if it cannot be expressed as a product of two or more other polynomials whose degree is less than that of $P(x)$. Otherwise, such a polynomial is called *reducible*.
Definition. A polynomial $P(x)$ of degree $r > 1$ is *primitive* if $P(x)$ is irreducible and $x^j \neq 1 \ mod \ P(x)$ for $0 < j < 2^r - 1$.

The primitive trinomials used in constructing our generators are over a finite field GF(2).[1] These primitive trinomials (as they contain only three terms) have the general form $x^r + x^s + 1$. For our construction they are

$$x^{47} + x^5 + 1$$

$$x^{41} + x^3 + 1$$

and

$$x^{35} + x^2 + 1$$

To test if an irreducible polynomial of degree r is primitive, the factorization of $2^r - 1$ is needed. Algorithms exist for such testing.
The above three trinomials were chosen so that
(i) their degrees are pairwise relatively prime.
(ii) the pairwise difference in their degrees is the same.
To this author's knowledge, a primitive trinomial of the highest known degree is

$$x^{3021377} + x^{1010202} + 1$$

discovered by Brent *et al.*[8] in August 2000.

2.1.2 Recurrences

Our construction uses the following Fibonacci recurrences which are based on primitive trinomials in the preceeding section. The first recurrence is

$$A_i = (A_{i-47} + A_{i-5}) \ mod \ 2^{32} \tag{2.1}$$

which will be initialized with an array of integers $\{A_0, A_1, \dots, A_{46}\}$. The second recurrence is

$$B_i = (B_{i-41} + B_{i-3}) \ mod \ 2^{32} \tag{2.2}$$

with the initialization array of integers $\{B_0, B_1, \dots, B_{40}\}$. The third and last recurrence is

$$C_i = (C_{i-35} + A_{i-2}) \ mod \ 2^{32} \tag{2.3}$$

Its intializing array will be $\{C_0, C_1, \dots, C_{34}\}$. The modulus used is 2^{32} because the implementation is done in the C programming language with the data model ILP32. The algorithm itself would work without modification on a 16-bit or a 64-bit machine. What's important is that data *and* arithmetic operators be represented in *single precision*.
In the following discussion, the generators in equations (2.1), (2.2), and (2.3) will be referred to as generators A, B, and C respectively.

[1]Galois field of order 2.

2.2 Creating Generator Initial States

The initial states of the generators in the preceeding section are arrays containing 32-bit integers. These arrays are derived from a *seed vector*, which is derived from a secret value which the user provides. The arrays differ from each other by a circular bit rotation of the array elements. We require that the seed vector (SV) be as long as the vector representing the initial state of a generator whose primitive trinomial has the highest degree.
The highest such degree is 47 and therefore the seed vector will be of the form

$$SV = (k_0, k_1, \dots, k_{46}) \tag{2.4}$$

where $\forall i = 0 : 46$, k_i is a 32-bit array element. Before using this seed vector to create the initial states of generators, a bit rotation is performed on it as shown below:

$$\{k'_0, k'_1, \dots, k'_{46}\} = \{k_0, k_1, \dots, k_{46}\} <<< 20 \tag{2.5}$$

The symbol $<<<$ is used to denote bit rotation. The value 20 is the number of positions by which bits in the array are shifted cyclically. This value is obtained from adding the lags of the generator A and then subtracting the word length, i.e., $(47 + 5) - 32 = 20$. The initial state of the generator A is obtained by the assignment,

$$\{A_0, A_1, \dots, A_{46}\} = \{k'_0, k'_0 \uplus k'_1, \dots, \uplus_{i=0}^{46} k'_i\} \tag{2.6}$$

Here $\uplus$ denotes *integer addition* modulo 2^{32}.
The initial state of generator B is then derived from that of generator A in the following manner. First we rotate bits in the array that is A's intial state,

$$\{A'_0, A'_1, \dots, A'_{46}\} = \{A_0, A_1, \dots, A_{46}\} <<< 12 \tag{2.7}$$

Here too the value 12 is obtained from adding the lags of generator B and subtracting the word length, i.e., $(41 + 3) - 32 = 12$. Since the array for B's initial state consists of 41 elements, elements A'_{41} to A'_{46} are discarded so that we get

$$\{B_0, B_1, \dots, B_{40}\} = \{A'_0, A'_0 \uplus A'_1, \dots, \uplus_{j=0}^{40} A'_j\} \tag{2.8}$$

as the initial state of generator B.
The initial state of generator C is derived from that of generator B by a similar process. We perform bit rotation on the initial state of B thus:

$$\{B'_0, B'_1, \dots, B'_{40}\} = \{B_0, B_1, \dots, B_{40}\} <<< 5 \tag{2.9}$$

where, as above, the value 5 is simply $(35 + 2) - 32$. To get the intial state of C from the preceeding equation, we discard elements B'_{35} to B'_{40} inclusive. Then generator C has

$$\{C_0, C_1, \dots, C_{34}\} = \{B'_0, B'_0 \uplus B'_1, \dots, \uplus_{k=0}^{34} B'_k\} \tag{2.10}$$

as its initial state.

2.2.1 Test Input Data

We illustrate the workings of the initializing scheme outlined above using actual 32-bit integer values. The tests for randomness performed in the next chapter will be based on the outputs resulting from the data input used here. The 7-bit ASCII characters stored in 8-bit **char** data type are used as user input. The input is an arbitrary string of 16 *hexadecimal* digits: $7E7B6F646635325D$.
From this intial value we construct the seed vector which must contain $47 \times 32 = 1504$ bits or 188 8-bit characters. The number of bits in the seed vector is determined by the longest lag and the word length. Constructing the seed vector consists of concatenating a string of eight characters (input by the user) with itself $\lceil \frac{188}{8} \rceil$ times, resulting in a string with 192 characters. This is more for implementation convenience, as described in chapter 4, than part of the algorithm specification. The seed vector is then given by

$$\begin{aligned} SV \;=\; & \{7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325d, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64, 6635325D, \\ & 7E7B6F64, 6635325D, 7E7B6F64\} \end{aligned}$$

Vector elements are similar because we simply concatenated a string of 32 hexadecimal digits (8 characters).

Performing bit rotation using equation (2.5) gives

$$\begin{aligned}\{k'_0, k'_1, \ldots, k'_{46}\} = \{&F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, F6466353, 25D7E7B6, F6466353,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353, 25D7E7B6,\\ &F6466353, 25D7E7B6, F6466353\}\end{aligned}$$

By equation (2.6) we find the initial state of generator A to be

$$\begin{aligned}\{A_0, A_1, \ldots, A_{46}\} = \{&353B653B, B0A0A0A0, E5DC05DB, 61414140,\\ &967CA67B, 11E1E1E0, 471D471B, C2828280,\\ &F7BDE7BB, 73232320, A85E885B, 23C3C3C0,\\ &58FF28FB, D4646460, 099FC99B, DFF25848,\\ &D738BB9B, FD10A351, F35706A4, 019EEE5A,\\ &0F7551AD, 354D3963, 2B939CB6, 516B846C,\\ &47B1E7BF, 6D89CF75, 63D032C8, 89A81A7E,\\ &BFEE7DD1, A5C66587, 9C0CC8DA, C1E4B090,\\ &B82B13E3, DE02FB99, D4495EEC, FA2146A2,\\ &F067A9F5, 163F91AB, 0C85F4FE, 325DDCB4,\\ &28A44007, 4E7C27BD, 44C28B10, 6A9A72C6,\\ &60E0D619, 86B8BDCF, 7CFF2122\}\end{aligned}$$

By equations (2.7) and (2.8) we find the initial state of generator B to be

$$\begin{aligned}\{B_0, B_1, \ldots, B_{40}\} \;=\; &\{663531C1, 4AE5C2E7, 95CB866A, 5F2CA952,\\ &8EC2FE97, 3CD4B341, 511B9A48, E3DDE0B3,\\ &DCD5597C, 544831A9, 31F03C34, 8E13A623,\\ &506C4270, 91403E21, 38496C20, 5DCDF993,\\ &E987B964, F3BCD899, 642718B2, 530CB9A9,\\ &A8278CFD, 7BBDBFB6, B58924CC, 6DCFE947,\\ &8C4BE01F, 29433657, 2C6FBEF6, AE17AAF4,\\ &95F4C550, FC4D3F10, C8DAEB2E, 13E3F6B0,\\ &C5223490, F4DBD1D4, 8ACAA176, 9F34D07C,\\ &19D421DF, 12EED2A7, 723EB5CC, 5009F856,\\ &940A6D3D\}\end{aligned}$$

The initial state of generator C is the computed using equations (2.9) and (2.10) giving

$$\begin{aligned}\{C_0, C_1, \ldots, C_{34}\} \;=\; &\{C6A63829, 235E951B, DCCF6266, C2648CB7,\\ &9AC45F9E, 355AC7C8, 58CE10E4, D48A275F,\\ &E23556E9, 6B3B8C0F, A93312A0, 6BB7D70A,\\ &7940251C, A147E943, A5DEAD4E, 5F9DDFCB,\\ &78950C69, F0301F95, 751335DF, D6AA6B14,\\ &DB9C0AC3, 35540199, 36789B26, A075C417,\\ &29F1C7F7, CEFE92DC, 5CF671B1, 1FEBD043,\\ &DE847A62, 682C5C7B, 8389C23D, 00089855,\\ &A44F2A73, 3FC96504, 991D93D7\}\end{aligned}$$

2.3 Combination Generators

Random bit generation consists of recursion using generators A, B, and C and the initial states specified in the section above. For the TM3w design, the outputs of our three Fibonacci generators are in turn used as inputs to another set of three generators. First, we let $\{A_k\}$ be the sequence of words produced by generator A, $\forall k \geq 46$; $\{B_l\}$ is sequence produced by generator B, $\forall l \geq 40$; and $\{C_m\}$ the sequence produced by generator C, $\forall m \geq 34$. Then the *three* 32-bit output words $W_{1,j}$, $W_{2,j}$, and $W_{3,j}$ are generated using the equations,

$$W_{1,j} = ((A_k \odot B_l) \uplus A_{k+1}) \oplus C_m \tag{2.11}$$

$$W_{2,j} = ((B_l \odot C_m) \uplus B_{l+1}) \oplus A_k \tag{2.12}$$

$$W_{3,j} = ((C_m \odot A_k) \uplus C_{m+1}) \oplus B_l \tag{2.13}$$

where the operators $\odot$, $\uplus$, and $\oplus$ denote integer multiplication modulo 2^{32}, integer addition modulo 2^{32}, and *exclusive*-or respectively. To initialize these combination generators we use the ordered pairs, $\{A_{46}, A_{47}\}$, $\{B_{40}, B_{41}\}$, and $\{C_{34}, C_{35}\}$. To compute the 32-bit words using the equations (2.11) to (2.13), we need to have outputs from generators A, B, and C available for ready use. This would necessitate precomputing a table such as the following.

A_k	A_{k+1}	B_l	B_{l+1}	C_m	C_{m+1}
A_{46}	A_{47}	B_{40}	B_{41}	C_{34}	C_{35}
A_{47}	A_{48}	B_{41}	B_{42}	C_{35}	C_{36}
A_{48}	A_{49}	B_{42}	B_{43}	C_{36}	C_{37}
A_{49}	A_{50}	B_{43}	B_{44}	C_{37}	C_{38}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 2.1: Precomputing inputs

But as will be seen in section 2.5, Fibonacci generators have fairly large periods, and this makes precumputing a table such as the above infeasible because of large memory space required to store it. This problem is addressed in Chapter 5, "Implementing the Algorithm".

2.4 Algorithm Analysis

This section describes the quantitative behaviour of our algorithm which is a reflection of its performance characteristics. We quantify this behaviour in terms of *computing time* represented by *bit* operations the algorithm has to perform to complete its task.
We will perform a *worst case* analysis since it gives an upper bound on the running time of the algorithm and thus produces a performance quarantee. We then know that for any input the running time of the algorithm will not exceed the determined bound.
Our algorithm consists of *three* subalgorithms: Initialization, Recurrences, Compound generators. We will analyze these in turn.

2.4.1 Initialization Phase

This is the subalgorithm by which generator intial states are derived from the seed vector. The dominant operations here are bit rotations, modulo additions, and assignment instructions.
A byte contains eight bits; if n_i with $i = A, B, C$ is the total number of bits in the respective arrays, then for each rotation $(n_i/8) + 1$ *shifts* are performed to the left. We let β_A, β_B, and β_C be the number bits rotated to the left in the creation of initial states

for generators A, B, and C respectively. Then the running time of this subalgorithm to perform bit rotation is

$$O(max(n_A\beta_A, n_B\beta_B, n_C\beta_C)) = O(n\beta_A)$$

since, in this case, $n_A\beta_A > n_B\beta_B > n_C\beta_C$. The runtime complexity for the bit rotation is therefore bounded by the largest product of the length of the array and number of bits rotated.

Modulo addition and assignment operations are carried out in equations (2.6), (2.8), and (2.10). We let c_i be an upper bound on the time required by the i^{th} computation step. The table below lists such steps and times taken by each.

Step	Cost	Times
for $i \leftarrow 1$ **to** n **do**	c_1	n
$sum \leftarrow sum + k'[i]$	c_2	$\sum_{j=1}^{n}(j+1)$
$sum\ mod\ 2^{32}$	see below	
$A[i] \leftarrow sum$	c_4	n

Table 2.2: Computation steps

In row 2 the time required to perform addition varies because the number of operands increases with the indexing variable. In row 3, let the dividend (sum) have k bits and the divisor have l bits. If $k \geq l$ then it takes $O(kl)$ to obtain the quotient and the remainder. Otherwise if $k < l$ the quotient is zero and the remainder is all the k-bit number. The worst-case running time T for modulo additions and assignments is obtained by adding the products of the *cost* and *times* columns, thus

$$T(n) = c_1 n + c_2 \sum_{j=1}^{n}(j+1) + c_4 n + O(kl)$$

This simplifies to

$$T = c_1 n + c_2 \frac{n(n+1)}{2} + c_4 n + O(kl)$$

By the maximum rule [2] the worst-case running time is then

$$T = O(max(n^2, kl))$$

2.4.2 Recurrences

Here we are adding two integers of the same size (32 bits) and then taking the modulos. For analysis, let each integer be k bits long. Adding them requires k bit operations, resulting in a sum with k or $k+1$ bits. Let the modulus have l bits. The running time T is given by

$$T = k + O(kl) \in O(kl)$$

[2] O(f(n) + g(n)) = O(max(f(n), g(n)))

2.4.3 Combination Generators

The combination generators in equations (2.11) to (2.13) perform modulo multiplication, modulo addition, and XOR operations in this order. For each generator, multiplication and taking a modulus will have a complexity

$$O(max(k^2, kl))$$

assuming that multiplicants each have k bits and the modulus has l bits. Adding and then taking a modulus will require

$$k + O(kl) \in O(kl)$$

The XOR operation will take time $O(k)$.

2.5 Lags and Periods

It is well known that Lagged Fibonacci generators based on primitive trinomials of the form $x^m + x^k + 1, m > k$ have maximal period $(2^{w-1})(2^m - 1)$, where w is the length of a computer word. We let $\rho_i = (2^{w-1})(2^{m_i} - 1), i = A, B, C$ be the periods of our three generators respectively. Values of these periods are tabulated below, where the computer word length $w = 32$.

Fibonacci recurrence	Period
$A_i = (A_{i-47} + A_{i-5})\ mod\ 2^{32}$	3.022×10^{23}
$B_i = (B_{i-41} + B_{i-3})\ mod\ 2^{32}$	4.722×10^{21}
$C_i = (C_{i-35} + C_{i-2})\ mod\ 2^{32}$	7.379×10^{19}

Table 2.3: Periods of Fibonacci recurrences

For our generators the values of $m_i = 47, 41, 35$ are *pairwise relatively prime*. This implies that the periods ρ_i are also pairwise relatively prime. Using a basic identity concerning the *greatest common divisor* (gcd) and the *least common multiple* (lcm) of two integers y and z,

$$y \cdot z = gcd(y, z) \cdot lcm(y, z), \quad y, z \geq 0$$

it can be shown that the combination generators in equations (2.11) to (2.13) have the periods as shown in the table below.
Substituting the values for ρ_i in the expressions in the right column of the above table will indicate that each combination generator has the longest and shortest possible periods. For instance the generator in row 1 has a period $\rho_1 = (2^{47} - 1)^2(2^{41} - 1)(2^{35} - 1)(2^{4(w-1)})$ which is an upper bound on this period for a fixed w. The lower bound on this period is $\rho_1 = (2^{47} - 1)^2(2^{41} - 1)(2^{35} - 1)$. This is true for the other two generators. For a

Combination generator	Period
$((A_k \odot B_l) \uplus A_{k+1}) \oplus C_m$	$\rho_A^2 \rho_B \rho_C$
$((B_l \odot C_m) \uplus B_{l+1}) \oplus A_k$	$\rho_A \rho_B^2 \rho_C$
$((C_m \odot A_k) \uplus C_{m+1}) \oplus B_l$	$\rho_A \rho_B \rho_C^2$

Table 2.4: Periods of Combination generators

Combination generator	Min	Max
$((A_k \odot B_l) \uplus A_{k+1}) \oplus C_m$	1.5×10^{51}	3.2×10^{88}
$((B_l \odot C_m) \uplus B_{l+1}) \oplus A_k$	2.3×10^{49}	5.0×10^{86}
$((C_m \odot A_k) \uplus C_{m+1}) \oplus B_l$	3.7×10^{47}	7.8×10^{84}

Table 2.5: Minimum and Maximum periods

32-bit machine, approximate values for shortest and longest periods of our combination generators are shown in the following table.
A comparison of values in tables 2.3 and 2.5 show that there is a significant improvement in terms of periods between the simple Fibonacci generators and their combinations. For most cryptographic applications the periods shown in the preceeding table should be sufficient.

Chapter 3

Bitwise Behaviour of the TM3w Generator

3.1 Outputs and Randomness

For a pseudo-random number generator to be *cryptographically secure*, there exists necessary and sufficient conditions that must be satisfied. These conditions are that such a generators must:
• pass all statistical tests for randomness
• provide cryptographic security
Cryptographic security simply means that the generator produces a sequence of numbers that cannot be predicted even if an attacker knows the full details of the algorithm except the seed vector.
In this chapter we will subject the output of our algorithm to a battery of statistical tests to see if our sequence possess the necessary atrributes that a truly random sequence would exhibit. For the initial states derived in chapter 2, the following table gives a sample output sequence of the three Fibonacci generators.
We then substitute values in table 3.1 into the equations (2.11), (2.12), and (2.13) to compute the final outputs. These outputs will then be subjected to five standard tests to determine their randomness. These tests are:
• Frequency test (mono-bit test)
• Serial test (two-bit test)
• Poker test
• Runs test
• Autocorrelation test
The outcome of each of the tests above is probabilistic and not definite. This means that a positive outcome simply gives us more confidence in the randomness of a sequence without giving the guarantee that the sequence will pass any additional tests. Passing the above tests is therefore a necessary *but not* sufficient condition for sequences to be considered random.
The following table shows a subset of the outputs produced by our combination genera-

generator A	generator B	generator C
3B08EE63	D873E78D	066F9D2D
86B8BDCF	9AEFBB3D	BC7C28F2
73458475	29D5F3A7	E33EFF93
BEF553E1	37A090DF	7EE0B5A9
AB821A87	29B2B9D9	7E035F31
8F63CF7E	66AAA6E8	B43B7D71
D15A023D	88BC2B27	D6D17015
E3BEB099	0D909A87	88C5A4D0
25B4E358	48DFFDD1	B906C6FE
B44B91B4	DD045CD0	F40130DF
1241A9FE	3F80D6BB	6239D99E
7A0FC473	D6F3A3F4	5FB907E9
82BAD622	2D709F40	DB79FEBA

Table 3.1: Fibonacci generators sample output

tors. The randomness tests are based on this sample.

$W_{1,j}$	$W_{2,j}$	$W_{3,j}$
DAD64130	7245A564	079E9B86
BE57C27B	4B7B0665	9E5E20D4
864DDB3A	15FAC69E	95217E7C
035C34A7	01E556B1	EBC7BC7F
B8E40E2F	EE40080B	12854365
D328097B	91D3FEFB	B83D139C
CE3EBD1C	4A64D0F1	2E902A5B
257036F1	BDE23687	6F53A8F6
9855AAD7	7081B718	1745DDC9
B249E272	40C33B76	79BF2FFE

Table 3.2: Combination generators sample output

Our algorithm is a random *bit* generator and as such we are interested in the statistical independence and lack of bias between *binary* digits. Each row in table 3.2 contains 96 bits and the ten row are concatenated to form a string containg 960 bits like,

$$11011010110101100100000100110000\ldots\ldots01111001101111110010111111111110$$

The goal here is simply to show that our random bit generator satisfies conditions for randomness as imposed by the tests.

3.1.1 Frequency Test

This test determines whether the number of 0's and 1's in a sequence are approximately the same, as would be the case for a random sequence. The statistic used is

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

where n_0 and n_1 denote the number of 0's ans 1's respectively, and $n = n_0 + n_1$. The above statistic approximately follows a χ^2 distribution [1] with 1 degree of freedom if $n \geq 10$. For the output sequence in table 3.2 $n_0 = 474$, $n_1 = 489$, the significance level $\alpha = 0.025$, the threshold $x_\alpha = 5.0239$. The statistic $X_1 = 0.15$.

3.1.2 Serial Test

This test determines whether the number of occurences of 00, 01, 10, and 11 as subsequences of the sample output sequence are approximately the same, as would be the case for a random sequence. As for the frequency test, n_0 and n_1 denote the numbers of 0's and 1's in the output sequence respectively. Furthermore, let n_{00}, n_{01}, n_{10}, and n_{11} denote the occurences of 00, 01, 10, and 11 in this sequence. The subsequences are allowed to overlap and therefore $n_{00} + n_{01} + n_{10} + n_{11} = (n-1)$. The statistic used is

$$X_2 = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0^2 + n_1^2) + 1$$

This statistic approximately follows a χ^2 distribution with 2 degrees of freedom if $n \geq 21$. For table 3.2 output, $n_{00} = 227$, $n_{01} = 227$, $n_{10} = 231$, and $n_{11} = 274$; $n_0 = 474$ and $n_1 = 486$. The significance level is $\alpha = 0.025$ and the threshold value $x_\alpha = 7.3778$. The statistic $X_2 = 6.420$.

3.1.3 Poker Test

A positive integer m is chosen such that $\lfloor \frac{n}{m} \rfloor \geq 5 \cdot (2^m)$; let $k = \lfloor \frac{n}{m} \rfloor$. This test determines whether the subsequence of length m each appear approximately the same number of times in the output sequence, as would be expected for a random sequence. The statistics used is

$$X_3 = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k$$

This statistic follows a χ^2 distribution with $2^m - 1$ degrees of freedom. For our output sample, $m = 5$, $k = 192$, and $\sum_{i=1}^{32} n_i^2 = 1317$. The significance level $\alpha = 0.025$ and the threshold value $x_\alpha = 48.2319$. The statistic $X_3 = 27.5$.

[1] see Appendix A

3.1.4 Runs Test

The runs test determines whether the number of runs[2] of various lengths in the output sequence is as expected for a random sequence. The expected number of *gaps* (a run of 0's) or *blocks* (a run of 1's) in a random sequence of length n is $e_i = (n-1+i)/2^{i+2}$. In defining a statistic to be used, let k be the largest integer i for which $e_i \geq 5$. Also, let B_i and G_i be the number of blocks and gaps, respectively, of length i in the output sequence for each $i, 1 \leq i \leq k$. The statistic used is

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

This statistic follows a χ^2 distribution with $2k-2$ degrees of freedom. For the sample output sequence the significance level $\alpha = 0.025$ and the threshold value $x_\alpha = 17.5345$. The statistic $X_4 = 12.187$.

3.1.5 Autocorrelation Test

The autocorrelation test checks for correlations between a sequence and its non-cyclic shifted versions. Let d be a fixed integer, $1 \leq d \leq \lfloor n/2 \rfloor$. The number of bits in the original sequence not equal to their d-shifts is $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$, where $\oplus$ denotes the XOR operator. For this test the statistic is

$$X_5 = 2(A(d) - \frac{n-d}{2})/\sqrt{n-d}$$

This statistic follows a standard normal distribution if $n-d \geq 10$. For our sample output sequence, the significance level $\alpha = 0.025$ and the thrshold $x_\alpha = 2.2950$. The statistic $X_5 = 0.2740$.

3.2 Randomness tests results

Except for the X_5 statistic, which has a standard normal distribution, all the other statistics have a χ^2 distribution with varying degrees of freedom (DOF) as can be seen in the next table. This table summarizes the outcomes of all the tests outlined in the previous subsections.

For all the tests the significance level was set at $\alpha = 0.025$. For a statistic in each row of the table, a comparison of the values under the columns denoted *Stat Value* (value of a statistic from a test) and *Threshold* (A value for which $P(X > x_\alpha) = \alpha$) shows that in *all* cases, values in the former column are less than those in the latter. For any particular statistic, this is a necessary condition for a sequence to pass a test asscociated with that statistic. The last column shows the outcome of each test.

[2]A run of a sequence is subsequence consisting of consecutive 0's and 1's which is neither preceeded nor succeeded by the same symbol [].

Statistic	Stat Value	Threshold	DOF	Outcome
X_1	0.2042	5.0239	1	Pass
X_2	6.4200	7.3778	2	Pass
X_3	27.500	48.232	31	Pass
X_4	12.187	17.535	8	Pass
X_5	0.2740	2.295	-	Pass

Table 3.3: Randomness tests results

From table 3.3 it is clear that our sample output sequence satisfies conditions for randomness as specified by the tests. As noted earlier, while passing all the tests for randomness is not a guarantee that the sequence was produced by a random bit generator, it *does* give us more confidence in the randomness of the sequence.

3.3 Vector Components Correlations

In this section we consider variations in the dependencies between the seed vector, the initial states of the driving generators, and the final outputs. These variations are quantified in terms of the *correlation coefficient*. When an increase in one set of values is accompanied by an increase in another set of values, the two sets are said to be directly (or positively) correlated. When one increases and other decrease, they are inversely (or negatively) correlated. In addition, quantities may be highly correlated or only slightly correlated; these are called thr degrees of correlation. Given two *samples spaces* $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$, The correlation coefficient r between X and Y is given by the equation

$$r = \frac{\sum_{j=1}^{n}(x_j - \overline{x})(y_j - \overline{y})}{\sqrt{\sum_{j=1}^{n}(x_j - \overline{x})(y_j - \overline{y})}}$$

where

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

and

$$\overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

are the *means* or average values of the two sample spaces.
The correlation coefficient r is such that:
♠ If there is no correlation, $r = 0$.
♠ If there is perfect linear correlation $r = \pm 1$.

♠ r cannot be numerically greater than 1.
No conditions were placed on the seed vector other than that it should be of the same size as the vector representing the initial state of generator A. So at times *all* the components of this vector may all be even. An undesirable side effect of this might be the propagation of powers of 2 throughout the initializing phase. We want the seed vector, the initial states, and the final outputs, to be pairwise uncorrelated as this complicates attempts to estimate the seed vector from intermediate states or outputs.

Transition	Correlation coefficient
$Seed\ Vector \leftrightarrow initial\ state\ A$	0
$initial\ state\ A \leftrightarrow initial\ state\ B$	0
$initial\ state\ B \leftrightarrow initial\ state\ C$	0

Table 3.4: Pairwise correlations coefficients

The coefficient values on the right column in the preceeding table indicate the upsence of any correlations between the vector integer components of the quantities on the left colummn. This is an indication that our initializing scheme is effective in destroying any dependencies between components of the respective vectors. From the preceeding discussion it is cleat that our algorithm is succesful in dissipating *any* statistical structure between the inputs, the intermediate stages, and the final output. This is in favour of its cryptographic strength.

Chapter 4

The Security of TM3w

4.1 Generating Functions

In chapter 2 three Fibonacci recurrences were used for our sub-generators. These were $A_i = (A_{i-47}+A_{i-5})\ mod\ 2^{32}$, $B_i = (B_{i-41}+B_{i-3})\ mod\ 2^{32}$, and $C_i = (C_{i-35}+C_{i-2})\ mod 2^{32}$, respectively. In this chapter is demonstrated the infeasibility of predicting these generators in the absence of the knowledge about their initial conditions.
Given a sequence $(a_r) = (a_0, a_1, a_2, \dots)$ of numbers, the *generating function* for this sequence is defined to be the power series

$$G(z) = \sum_{r=0}^{\infty} a_r z^r = a_0 + a_1 r + a_2 r^2 + \dots$$

The following theorems, stated without proof, formulates the techniques relevant to this chapter used in algebraic manipulations of generating functions.

Theorem 1 *Let G(z) and H(z) be generating functions for the sequences (a_r) and (b_r) respectively.*
(i) Addition. *For any numbers α and β, $\alpha G(z) + \beta H(z)$ is the generating function for the sequence (c_r), where*

$$c_r = \alpha a_r + \beta b_r,\ \forall r;$$

(ii) Multiplication. *G(z)H(z) is the generating for the sequence (c_r), where*

$$c_r = a_0 b_r + a_1 b_{r-1} + a_2 b_{r-2} + \dots + a_{r-1} b_1 + a_r b_0,\ \forall r;$$

The sequence (c_r) is called the convolution of the sequences (a_r) and (b_r).
(iii) Shifting. *$z^m G(z)$, $m \in \mathbf{N}$, is the generating function for the sequence (c_r), where*

$$c_r = \begin{cases} 0 & if 0 \le r \le m-1 \\ a_{r-m} & if r \ge m; \end{cases}$$

Theorem 2 $(1+z)^n = 1 + nz + \frac{n(n-1)}{2} z^2 + \dots = \sum_{k \ge 0} \binom{n}{k} z^k$

The second result is called the Binomial theorem.

4.1.1 Recurrences and Generating Functions

Consider the recurrence used as the first sub-generator,

$$A_j = A_{j-47} + A_{j-5}$$

with $\mod 2^{32}$ removed for brevity. This can be expressed as

$$A_j - A_{j-47} - A_{j-5} = 0$$

By theorem 1, the sequence A_j has the generating function

$$G(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_n z^n + \ldots$$

The sequence $-A_{j-47}$ has the generating function

$$-z^{47}G(z) = -A_0 x^{47} - A_1 x^{48} - \ldots - A_{n-47} x^n - \ldots$$

For the sequence $-A_{j-5}$ the generating function is

$$-z^5 G(z) = -A_0 z^5 - A_1 z^6 - A_2 z^7 - \ldots - A_n x^{n+5} - \ldots$$

By taking the sum of the three generating functions, we have

$$\begin{aligned}(1 - z^5 - z^{47})G(z) &= A_0 + A_1 z + A_2 z^2 \ldots + (A_5 - A_0)z^5 + \ldots \\ &\quad + (A_n - A_{n-5} - A_{n-47})z^n + \ldots\end{aligned}$$

For all $n \geq 47$, $A_n - A_{n-5} - A_{n-47} = 0$, by the recurrence relation. Therefore

$$\begin{aligned}(1 - z^5 - z^{47})G(z) &= A_0 + A_1 z + A_2 z^2 \\ &\quad + (A_5 - A_0) + (A_6 - A_1) + \ldots\end{aligned}$$

After simplifying and tidying, we arrive at the result

$$G(z) = \frac{\sum_{j=0}^{46} A_j z^j}{(1 - z^5 - z^{47})} \tag{4.1}$$

The values A_i are the initial conditions for the recurrence relation. The denominator

$$1 - z^5 - z^{47}$$

is the *characteristic polynomial* of the sequence (A_j). The solutions of the characteristic polynomial are called the characteristic roots.

Lemma 1 *Let $\gamma_1, \gamma_2, \ldots, \gamma_r$ be distinct roots of the characteristic polynomial, then the solution of a general recurrence relation $c_0 x_n + c_1 x_{n-1} + \ldots + c_r x_{n-r} = 0$ is*

$$x_n = X_1(\gamma_1)^n + X_2(\gamma_2)^n + \ldots + X_r(\gamma_r)^n$$

The X_i's are constants.

The generating function of the second sub-generator $B_j = B_{j-41} - B_{j-3}$ is found to be

$$H(z) = \frac{\sum_{j=0}^{40} B_j z^j}{1 - z^3 - z^{41}} \tag{4.2}$$

The B_i's are the initial conditions. Similarly for the third sub-generator, its generating functions is given by

$$J(z) = \frac{\sum_{j=0}^{34} C_j z^j}{1 - z^2 - z^{35}} \tag{4.3}$$

The denominators of the latter generating functions are also the characteristic polynomials of the sequences (B_j) and (C_j) respectively, and they are insoluble in the set of integers.

4.1.2 Extracting coefficients

By lemma 1 the solution to the recurrence relation for the sequence (A_i)

$$A_n = A_1(\gamma_1)^n + A_2(\gamma_2)^n + \ldots + A_{47}(\gamma_{47}) \tag{4.4}$$

where the γ_i's are the characteristic roots of the equation $1 - z^5 - z^{47} = 0$ In the set of integers $\mathbf{Z}$ the characterisic polynomial for the sequence (A_j) is *irreducible*, hence the γ_i's are indeterminate.
An alternative approach is to use a result from the theory of complex variables called Cauchy's integral, by which any desired coefficient of the generating function can be extracted. Such a coefficient is expressed in terms of a contour integral thus

$$A_n = \frac{1}{2\pi i}\oint_{|x|=r} \frac{G(z)}{z^{n+1}}dz, \quad i^2 = -1 \tag{4.5}$$

if G(z) converges for $z = z_0$ and $0 < r < |z_0|$
Substituting (4.1) into (4.5) yields

$$A_n = \frac{1}{2\pi i}\oint_{|x|=r} \frac{\sum_{j=0}^{46} A_j z^j}{(1 - z^5 - z^{47})z^{n+1}}dz, \tag{4.6}$$

if G(z) converges for some $z = z_0$ and $0 < r < |z_0|$. The generating function G(z) has a singularity whenever the characteristic polynomial $(1 - z^5 - z^{47})$ has roots. In both equations (4.4) and (4.6) the initial value $(A_0, A_1, \ldots, A_{46})$ are required to carry ou the computations. By a similar calculation, the n^{th} terms of sequences (B_j) and (C_j) are

$$B_n = \frac{1}{2\pi i}\oint_{|x|=r} \frac{\sum_{j=0}^{40} B_j z^j}{(1 - z^3 - z^{41})z^{n+1}}dz, \tag{4.7}$$

and

$$C_n = \frac{1}{2\pi i}\oint_{|x|=r} \frac{\sum_{j=0}^{34} C_j z^j}{(1 - z^2 - z^{35})z^{n+1}}dz \tag{4.8}$$

respectively. For the latter two equations as well initial conditions are required to get the n^{th} term of the sequences. By inspection, the characteristic polynomials in (4.6) to (4.8) do *not* have integer roots; their complex roots, if they have, are not trivial to find.
There exist 2^{32} *distinct* 32-bit integers. As an example, consider the equation (4.4) or (4.6); an exhaustive search for the correct combination of such 32-bit integers requires that the attacker tries all

$$\binom{2^{32}}{47} = \frac{2^{32}!}{(2^{32}-47)!47!} > 10^{393}$$

This includes the number of combinations where *all* the initial values are even. If the attacker wants the combinations so that not all the initial values are even, he has to try

$$2^{r(w-1)}(2^r - 1)$$

where r is the lag and w is the computer word length. In this case $r = 47$ and $w = 32$ and therefore the total number of all such combinations is

$$2^{47\times31}(2^{47} - 1) > 10^{452}$$

In both cases the attacker has to try on average 50% of all the possible combination. To appreciate the above numbers, if their attacker has a CPU that can try 10^{10} combination in 1μs (1 microsecond), such a CPU will finish its task in about 10^{374} years! These number suggest that exhaustive search of the initial states is computationally infeasible.

4.2 Mixed Operators

In the prceeding sections it was shown that an exhaustive search on Fibonacci generators is infeasible. To this infeasibility another complication is added in the form of combining output from the sub-generators using algebraic operators with the properties as below. Using these operators the algorithm is able to capture properties that define good cryptographic behaviour:
♣**Bit omission.** In the initialization phase, bits are discarded as the initial state of one generator is derived from that of another. This omission of bits means that uncertainty is added regarding the original bit pattern of the seed vector.
♣**Difussion.** By using modulo addition and modulo multiplication, we are able to dissipate the statistical structure of the seed vector into the long long-range statistics of the generator outputs. That is, the value of each digit of the seed vector affects the values of many generator output digits. Likewise the the value of each generator output digit is affected by many digit values of the seed vector.
♣**Confusion.** By using outputs of the three Fibonacci generators for each combination generator, we succeed in making the relationship between the statistics of the final outputs and the seed vector as complex as possible.
Our algorithm attains the latter two properties by using the three operators ($\uplus$, $\oplus$, and $\odot$) that do not satisfy the ordinary laws of group algebra. These operators are such that:

♠*No* pair of the three operatots satisfy a distributive law, i.e.

$$x \uplus (y \odot z) \neq (x \uplus y) \odot (x \uplus z)$$

$$x \odot (y \oplus z) \neq (x \odot y) \oplus (x \odot z)$$

$$x \oplus (y \uplus z) \neq (x \oplus y) \uplus (x \oplus z)$$

♠*No* pair of the three operators satisfy an associative law, i.e.

$$x \uplus (y \odot z) \neq (x \uplus y) \odot z$$

$$x \odot (y \oplus z) \neq (x \odot y) \oplus z$$

$$x \oplus (y \uplus z) \neq (x \oplus y) \uplus z$$

Using operators that have the properties as above in combination provides for a complex transformation of the input data into the output data. This should make attempting to predict the algorithm output infeasible.

4.3 Other Primitive Trinomials

Lemma 2 *If a polynomial $p(x)$ is a primitive polynomial, then so is $x^n p(1/x)$*

By applying this lemma to the primitive trinomials in the previous section, we can deduce that $x^{47} + x^{42} + 1$, $x^{41} + x^{38} + 1$, and $x^{35} + x^{33} + 1$ are also primitive trinomials modulo 2. If we keep the word length at 32 bits, defining our recurrences in terms of these primitive trinomials would result in more bit rotations during the initialization phase and possibly a much improved mixing of the individual bits. The periods of such recurrences would remain unchanged as they only depend on the longer lag, which remains unchanged, and the word length. The time to initialize the recurrences would also remain unchanged.
A number of researchers have been discovering primitive trinomials of extremely large degrees. Based on the information avalable to this author at the time of writing, two trinomials with large degrees are

$$x^{3021377} + x^{361604} + 1$$

and

$$x^{3021377} + x^{1010202} + 1$$

At the time that the source paper was written, the search for the third trinomial

$$x^{6972593} + x^s + 1$$

was continuing, with the value of s still unknown. A quick calculation reveals that the recurrence defined by the foregoing trinomials would have a period in the order of $10^{1168240}$! Clearly recurrences with such long lags are slow to initialize and, although the periods are extremely long, this does not enhance the security in any meaningful way. In fact because of a large number of intial values required, a brute force attack, assuming the existence of combinations with all values positive, is reduced in terms of effort compared to the values calculated previously. Brute force effort is greatest in the case where initializing words are not all even. For such long lags, our intializing scheme *would* be inefficient.

4.4 TM3w Initialization

The algorithm used in the initializing phase of the TM3w generator exhibits properties considered in this section. These properties differ markedly from those of message authenticating codes (MAC) and hash functions. With MACs and hash functions, the input can be of any length, but the output has a *fixed* length. For instance the hash algorithm MD-5 takes input of arbitrary length and output 128-bit message digest. The SHA-1 algorithm takes any input up to 2^{64} bits and output a 160-bit message digest. RIPEMD-160 also takes input of any length and produces output 160 bits long. There are others with a similar property.
The beauty of our initializing algorithm is that its output is *variable* depending only on the degree of a primitive trinomial defining a particular recurrence. In addition, our initializing algorithm is a:
♠ **One-to-many function**. The domain consists of one vector of fixed length and the range consists of many vectors of variable lengths.
For the following discussion let's say the function defining our initializing algorithm is TM3w_init(). Then

$$TM3w\_init(v) = \{f_0(v_0), f_1(v_1), f_2(v_2), \ldots, f_n(v_n)\} \tag{4.9}$$

where f_i defines (bit) operations carried out at the i^{th} stage according to the primitive polynomial $x^{r_i} + x^{s_i} + 1$. At each stage the vector v_i is computed from the preceeding vector v_{i-1} by an equation of the form

$$v_{i+1} = f_{i+1}(f_i(v_i)),\ for\ i\ =\ 0\ to\ n \tag{4.10}$$

when $i = 0$, f_i is simply a bit rotation and v_i is the seed vector. The primitive trinomials defining the recurrences are chosen such that if

$$x^{r_i} + x^{s_i} + 1,\ with\ i\ = 1\ to\ n$$

is a *set* of primitive trinomials then

$$r_1 > r_2 > r_3 > \ldots > r_n$$

and

$$r_1 - r_2 = r_2 - r_3 = \ldots = r_{n-1} - r_n = constant$$

and also

$$gcd(r_1, r_2) = gcd(r_2, r_3) = \ldots = gcd(r_{n-1}, r_n) = 1$$

where gcd(r_i, r_j) is the *greatest common divisor* of the two integer values r_i and r_j. Since all the longer lags of the recurrences are pairwise relatively prime, then each recurrence contributes its maximum period to the combination generator.
♠ **One-way function**. The n^{th} stage initial state is computable by

$$v_n = f_n(f_{n-1}f_{n-2}(\ldots f_1(v_1))\ldots))) \qquad (4.11)$$

Within each stage there is reduction by modulus operations and from one stage to the next there is bit omission. This means that inverting equation (4.11), which generalizes (4.10), is infeasible since the attacker would *not* know which bits are omitted.
The two properties of one-to-manyness and one-wayness enhance significantly the security of the TM3w generator. The both imply that even if an attacker has access to one of the initial states he is unlikely to succed in infering other states from the one he has. Also he will be unsuccesful in trying to work backwards to the seed vector.

4.5 Combination Generators

Combining a number of generators by some suitable operators certainly provides for composite sequences with better randomness properties. In this section an analysis of the combination generators is done using some results from section (4.1). We need to see how much information about the sequences produced by the combination generators can be found deduced. This is important from the security point of view. From section (4.2) we saw that the algebraic operators used in the combination generators do not satisfy the distributive and associative rules of group algebra. This is a serious limitation on an attempt to study the behaviour of these combination generators using the generating functions of the component generators. A simplifying assumption is that the operators used are *ordinary* addition and multiplication. This assumption is necessary here for a further analysis. Again we will consider only one out of the three combination generators, and then make inferences regarding the other two generators. From equation (2.11), the first combination generators is given by

$$W_{1,j} = ((A_k \odot B_l) \uplus A_{k+1}) \oplus C_m$$

Substituting the recurrence relations for A_n, B_n, and C_n in the above relationship and *replacing* modulo operations with ordinary addition and multiplication gives

$$W_{1,n} = (A_{n-47}B_{n-41} + A_{n-47}B_{n-3} + A_{n-5}B_{n-41} + A_{n-5}B_{n-3}) + A_{n+1}) + C_n \qquad (4.12)$$

As can be seen, the recurrence of (4.12) is *non-linear* and this makes analysis much more difficult. Let K(z) be the generating function of the sequence defined by equation (4.12), then

$$K(z) = \sum_{n=0}^{\infty} W_{1,n} z^n \tag{4.13}$$

Substituting equation (4.12) into (4.13) gives

$$K(z) = \sum_{n=0}^{\infty} (((A_{n-47}B_{n-41} + A_{n-47}B_{n-3} + A_{n-5}B_{n-41} + A_{n-5}B_{n-3}) + A_{n+1}) + C_n) z^n \tag{4.14}$$

It should be clear from the equation above that in the upsence of initial conditions for the constituent recurrences, and the difficulty of solving their characteristic polynomials, obtaining the value of W_n by the Cauchy integral

$$W_n = \frac{1}{2\pi i} \oint_{|x|=r} \frac{K(z)}{z^{n+1}} dz, \quad i^2 = -1 \tag{4.15}$$

is computationally infeasible. So the only hope for the attacker is to attempt to break either the initialization algorithm, or the constituent Fibonacci generators. If we let M(z) and N(z) to be the generating functions of the other two combination generators, $W_{2,n}$ and $W_{3,n}$, then the n^{th} term of each sequence will be expressed by

$$M(z) = \sum_{n=0}^{\infty} (((B_{n-41}C_{n-35} + B_{n-41}C_{n-2} + B_{n-3}C_{n-35} + B_{n-3}C_{n-2}) + B_{n+1}) + A_n) z^n \tag{4.16}$$

and

$$N(z) = \sum_{n=0}^{\infty} (((C_{n-35}A_{n-47} + C_{n-35}A_{n-3} + C_{n-2}A_{n-47} + C_{n-2}A_{n-3}) + C_{n+1}) + B_n) z^n \tag{4.17}$$

For the latter two equations also, intial conditions to Fibonacci recurrences are required to obtain expressions for the n^{th} terms. Since these initial conditions are not available to the attacker, using methods such as ewuation (4.15) to do estimates *will* be infeasible.

In this chapter it was shown mathematically that at least the TM3w generator *is* comptationally secure. The effort to do exhaustive search of the initial conditions is just too great to discourage even the most committed of “hackers”. As yet it is not obvious (to this author) how an efficient mathematical attack can be carried out.

Chapter 5

Implementing the Algorithm

This chapter describes the implementation of the various components of the combination generator algorithm. Figure 4.1 is a flow diagram outlining the sequence of execution of the various algorithm stages. The algorithm itself is language independent, but the implementation was done in the the C programming language. The code has been tested on machines running SunOS 5.7 and IBM AIX. These two operating systems have a UNIX flavour, and as is well known C and UNIX have a symbiotic relationship. The implementation code is described from the user interface to the final output generating functions. However, this is not production code; hence the aim is demonstrating how different components are implemented and what their outputs are. Difficulties due to limitations imposed by the programming language are noted as well as approaches around them.
The entire algorithm primarily consists of three sub-algorithms. The first sub-algorithm is the initialization phase for the Fibonacci recurrences. The second sub-algorithm is the Fibonacci recurrences themselves. The third sub-algorithm is the three combination generators.

5.1 User Interface

5.1.1 Password to Seed Vector

The user of this bit generator is required to enter a (secret) password from the standard input stream device (keyboard). The password should be no more or no less than *eight* characters. It is this password from which the seed vector is formed.
Every character on the keyboard has an ASCII encoding, which is a decimal or hexadecimal digit by which the character is represented within the computer. Each such digit has a binary representation consisting of *seven* bits, however each character is stored in an *eight* bits byte. We saw in chapter 2 (eqn (2.4)) that the seed vector has as many components as the longest lag of the Fibonacci recurrence, which is 47 in this case. Our implementation is on a 32-bit computer and therefore the seed vector consists of $47 \times 32 = 1504$ bits. This is 188 single-byte characters. A user-specified string of 8 characters is the concatenated

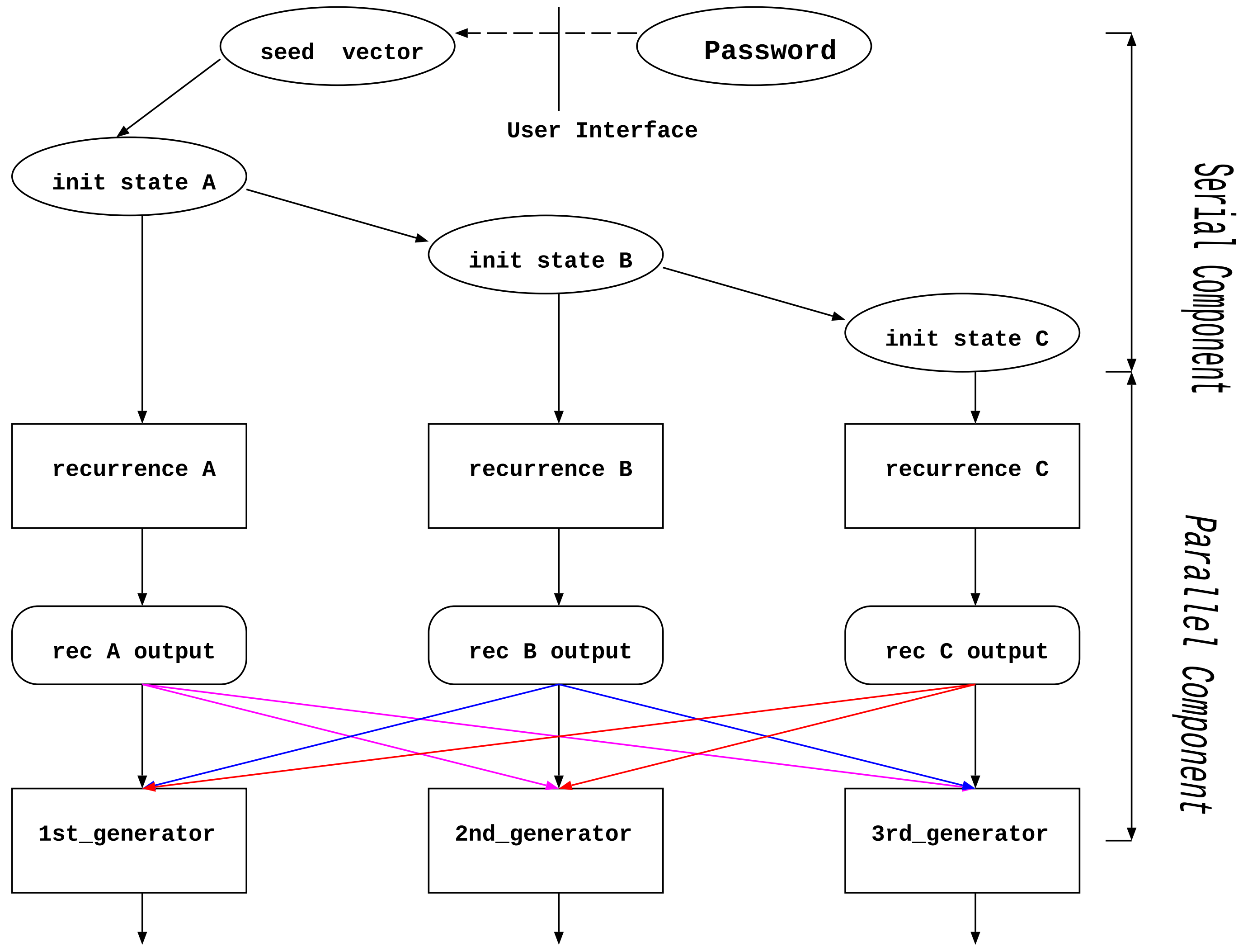

Figure 5.1: Program execution sequence

with itself $\lceil \frac{188}{8} \rceil = 24$ times, resulting in a string with a total of 192 characters. Details like bounds checking, of which much will be said later, and input prompts will be left out for brevity and clarity. Only portions of the programs encapsulating relevant functionality are included and described.

```
#define NUM 24
void main() {
 char *tempo, *ptr, *buffer;
 int j;
 tempo = (char *)malloc(sizeof(char) * 8);
 buffer = (char *)malloc(strlen(tempo) * NUM + 1);

 while ( *(ptr = gets(buffer)) != NULL)
 strcpy(tempo, buffer);
 for (j = 0; j < NUM; j++)
 strcat(buffer, tempo);
}
```

When this program is run, it requests a string of 8 characters and outputs the same string concatenated NUM times. From the characters in this string, we form an array of 32-bit integers by concatenating the ASCII hexadecimal digits of every four consecutive characters. This we do by first initializing a two dimensional array by the $(NUM \times 2) \times 4$ characters from the concatenated string. We end up with an array with 48 rows and 4 columns.

```
void stringtoint(char *string) {
  const int ROW = 47;
  const int COL = 4;
  char *ptr, array[ROW][COL];
  int i, j, k;
  ptr = string;
 for (i = 0; i < ROW; i++) {
   for (j = 0; j < COL; j++)
     while (k < strlen(string))
            array[i][j] = ptr[k++]

 }/*end for (i = ...) loop*/
}
```

Each row of `array` has four characters. Each character's ASCII decimal representation is converted into a 7-bit binary string. The resulting strings are stored in an array as shown in the figure 4.1. The following code converts an integer to its binary representation.

```
void int_to_bits(char leta) {
 int dividend, quotient = 0,
 const int divisor = 2;
 char remainder[7];

 dividend = (int) leta;

 for (j = 0; j < 7; j++) {
  if (dividend >= divisor){
    quotient = dividend/base;
    *(remainder + j) = dividend % divisor;
    dividend = quotient;
  }
  else {
    *(remainder + j) = dividend;
  }
 } /*end for(...) loop*/

}
```

Each character is first typecast into its decimal representation and this decimal is converted to a binary string. This is made necessary by the upsence of a hexadecimal data type in C, in which case 8 hexadecimal digits would be concatenated to form a 32 bit integer. By a well known binary to decimal conversion formula

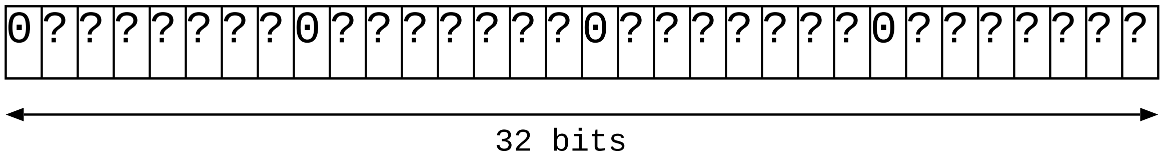

Figure 5.2: String of 32 bits

$$\sum_{i=0}^{n-1} X_i 2^i = X_0 + X_1 2 + X_2 2^2 + \ldots + X_{n-1} 2^{n-1}$$

a 32 bit integer is then constructed. This formula is implemented as

```
int bits_to_int(void) {
  int i, num, sum = 0;
  const int size = 47;
  unsigned long seedvector[size];
  int bitarray[32] = { ....}; /*bit string initialization*/

  for (i = 0; i < 32; i++){
   num = bitarray[i] * (unsigned)pow(2,i);
   sum += num;
 ........

   for (j = 0; j < size; j++){
     seedvector[j] =  sum;
   }
 }/*end for (i = ...)*/
}
```

The 32-bit integers formed from the password are then used to initialize the seed vector. Because of the concatenation of 8 character strings with themselves, components of the seed vector are basically *two* integers alternating with each other. The concatenation of the same string with itself is done so that the user does not have to input 192 ASCII characters. From the point of view of security, the scheme adopted for the user interface weak makes for a weak link. The ASCII character chart contains a total of 256 characters, printable and non printable. If 8 characters are chosen out of 128 printable characters, an attack by exhaustive search will have a work factor of

$$\binom{128}{8} = \frac{128!}{(128-8)!8!} \geq 1.4 \times 10^{12}$$

If the entire 256 ASCII characters are used as the alphabet, performing exhaustive search has a work factor of

$$\binom{256}{8} = \frac{256!}{(256-8)!8!} \geq 4.1 \times 10^{15}$$

If the user had to input all the 192 characters to form the 48 32-bit integers, the work factor for exhaustive search would be

$$\binom{256}{192} = \frac{256!}{(256-192)!192!} \geq 1.9 \times 10^{61}$$

The last option has the highest work factor and thus good for security, but it is clearly inconvenient from the human user's point of view.

5.2 Initial States

The most basic unit of computer data storage is the bit. To derive the intial states of the three Fibonacci subgenerators from the seed vector and from each other, it is necessary to carry out a number of operations at the bit level. The C bitwise operators can be used only with *integral* types: char, int, and long. The initializing scheme oulined in chapter 2 requires that bits be rotated so that the left-most bit in an array can be moved to a right-most position, or vice versa. In this regard the C bitwise operators are inadequate. Three functions that together provide bit rotation were adapted from [ref]. These functions carry out bitwise operations on buffers containing *any* number of bits. They are `get_bit()`: it gets the state of a bit at a particular position in a bit string, `set_bit()`: it sets the state of a bit at a given position in the buffer to a specified bit value, `lbit_rot()` rotates the bits in a buffer to the left a specified number of positions. `lbit_rot()` begins by saving the leftmost bit of the leftmost byte and then shifting the each byte one bit to the left. As each byte is shifted, the rightmost bit of the preceeding byte is set to the bit shifted of the left of the current byte. Once the last byte is shifted, its rightmost bit is set to the bit shifted off the first byte. This process is repeated as many times as the number of bits to be rotated. Bit rotation to the left is performed by the operation,

```
void lbit_rot(unsigned char *buffer, int size, int count){

 int firstbit, lastbit, i, j;

 if (size > 0){
   for (j = 0; j > count; j++){
    for (i = 0; i < (size/8); i++){
     lastbit = get_bit(&buffer[i], 0);
   if (i == 0){
    firstbit = lastbit;
   }
 else {
   set_bit(&buffer[i-1], 7, lastbit);
   buffer[i] = buffer[i] << 1;
 }
   set_bit(buffer, size-1, firstbit);
  }
 }
}
```

The `lbit_rot` function rotates the bits within `buffer`, containing `size` bits, to the left `count` bits. After the operation, the leftmost `count` bits become the `count` rightmost bits in the `buffer`, and all other bits are shifted accordingly. Setting the bits is done by,

```
void set_bit(unsigned char *buffer, int pos, int state){

 unsigned char mask;
 int i;

 mask = 0x80;
 for (i = 0; i < (pos % 8); i++)
  mask = mask >> 1;
 if (state)
  buffer[pos/8] = buffer[pos/8] | mask;
 else
  buffer[pos/8] = buffer[pos/8] & (~mask)
}
```

This operation sets the state of a bit in position `pos` of the `buffer` to a state specified by `state` using a mask. The leftmost position in the buffer is 0. The state will be 1 or 0. Getting the state of a bit at a particular position is done by,

```
int get_bit(const unsigned buffer, int pos){

 unsigned char mask;
 int i;
 mask = 0x80;

 for (i = 0; i < (pos % 8); i++)
   mask = mask >> 1;

 return (((mask & buffer[(int)pos /8)]) == mask) ? 1 : 0);
}
```

The above operation gets the state of a bit in `buffer` by determining in which byte the desired bit resides, and then using a mask to get a specific bit from that byte. The bit set to 1 in `mask` determines which bit will be read from the byte. A loop is used to shift this bit into a proper position. A desired bit is fetched by indexing to the appropriate byte in `buffer` and applying a mask. In all the following the constant `MOD` is defined as `(unsigned)pow(2, 32)`, where pow() is a function from header math.h file. The function below computes the initial state of the first Fibonacci generator.

```
#define SIZE_1 47

unsigned long *init_state_of_1(unsigned long *seedvec, int n){

 int i;
 unsigned long sum = 0;
 unsigned long state_1[SIZE_1];

 state_1 = seedvec;
 if (n < SIZE_1 || n > SIZE_1){
   exit(1);
 }
 else {
  lbit_rot(seedvec, SIZE_1 * 32, 20);

   for (i = 0; i < SIZE_1; i++){
     sum = (sum % MOD + state_1[i] % MOD) % MOD;
     state_1[i] = sum;
   }
 }
return state_1;
}
```

This function implements equation (2.6) from chapter2 by taking the seed vector and performing incremental modulo addition operations on its components. It is passed a pointer to the seed vector and an integer to do the bound checking. Bit are rotated by $(47 + 5) - 32 = 20$ positions. The bound checking is also done by using a property of modulo addition

$$(a + b) \mod n = (a \mod n\ +\ b \mod n) \mod n$$

Dealing with integers whose bit representation uses the entire computer word length calls for *defensive programming*; single precision computations are enforced and possible *buffer overflows* are prevented. In a similar manner, the initial state for the second Fibonacci generator is computed by the function,

```
#define SIZE_2 41

unsigned long *init_state_of_2(unsigned long *state_1, int n){

 int i;
 unsigned long sum = 0;
 unsigned long state_2[SIZE_2];

 state_2 = state_1;
 if (n < SIZE_2 || n > SIZE_2){
   exit(1);
 }
 else {
  lbit_rot(seedvec, SIZE_2 * 32, 12);

   for (i = 0; i < SIZE_2; i++){
     sum = (sum % MOD + state_2[i] % MOD) % MOD;
     state_2[i] = sum;
   }
 }
return state_2;
}
```

This send function takes the initial state of the first sub-generator, perform a bit rotation by $(41 + 3) - 32$ positions, discards vector elements 41 to 46 from the previous state, and then perform incremental modulo addition on the remain 41 vector elements. The same bounds checking as before is done here too. For the third and last sub-generator, the process is repeated as below,

```
#define size_3 35

unsigned long *init_state_of_3(unsigned long *state_2, int n){

 int i;
 unsigned long sum = 0;
 unsigned long state_3[SIZE_3];

 state_3 = state_2;
 if (n < SIZE_3 || n > SIZE_3){
   exit(1);
 }
 else {
  lbit_rot(seedvec, SIZE_3 * 32, 5);

   for (i = 0; i < SIZE_1; i++){
     sum = (sum % MOD + state_3[i] % MOD) % MOD;
     state_3[i] = sum;
   }
 }
return state_3;
}
```

This last function implements equation (2.10) by rotating bits from the second state, and then discards vector components 35 to 40 before rotating bits by $(35+2)-32$ positions. The outputs from the preceeding three functions form inputs to the recurrences described in the next section.

5.3 Fibonacci Recurrences

The recurrences of equations (2.1) to (2.3) are implemented each as a cyclic list. For each implementation, we start with an array of dimension r (longer lag) and two indices i, j with memory locations X[1], X[2], ..., X[r] initially set to values $x_{r-1}, x_{r-2}, \ldots, x_0$, respectively, $i = r$ and $j = s$. Then a new element in the sequence is obtained by

$$\begin{aligned} X(i) &= X(i) + X(j) \\ & \quad\; output\ X(i) \\ i &= i-1;\ if\ i=0\ then\ i=r \\ j &= j-1;\ if\ j=0\ then\ j=r \end{aligned}$$

The pseudo code doing the above is the same for all the Fibonacci recurrence and therefore it is shown for only one of the generators.
For any recurrence the sequence is generated thus,

```
#define LISTSIZE
#define MAXLAG
#define MINLAG

void fibgen_1(const unsigned long *state){
 int i, j, r, s;
 unsigned long output, recur[LISTSIZE];

 for (i = 0, j = LISTSIZE; i < LISTSIZE; i++, j--){
    recur[j] = *(state + i);
 }
 r = MAXLAG; s = MINLAG;
 while(1){
    *(recur + r) = ((*(recur + r)%MOD) + (*(recur + s)%MOD))%MOD;
 output = (recur + r);
   r -= 1;
 if (r == 0) {r == MAXLAG}
   s -= 1;
 if (s == 0) {s == MAXLAG}
 }
}
```

Excluding the programs, the cyclic lists for the recurrences (2.1) to (2.3) require only 188 bytes, 164 bytes, and 140 bytes of memory respectively.

5.4 Combination Generators

The three combination generators in equations (2.11) to (2.13) inclusive form the backbone of the TM3w pseudo-random bit generator. For each 32-bit integer, a function defining each combination generator takes three parameters. Since these functions are similar in form *but not* substance, code for only one such functions defining the first combination generator is shown below.

```
long word1_generator(unsigned long *fib1, unsigned long *fib2,
unsigned long *fib3){
unsigned long word1, A_0, A_1, B, C;

  A_0 = *fib1 % MOD;
  A_1 = *(fib2 + 1) % MOD;
  B = fib2 % MOD;
  C = *fib3 % MOD;
  word1 = ((A_0 * B) + A_1) ^ C;
  return word1;
}
```

The parameters passed to these functions are the return values of the Fibonacci recurrences described previously. For the generator to output 96 bits, these combination generator functions must return, or appear to return, the results of their computations simultaneously. This is the subject of the next chapter: concurrency.

Chapter 6

TM3w Program Concurrency

Software that performs non-trivial tasks is complex. In designing an application, consideration of the following three issues help in the management of complexity. Identifying the core problem to be solved, choosing the programming language in which to implement the solution, and Programming *for* the hardware on which the program will run. The first two considerations have already been addressed in the previous chapters. The latter consideration is the subject of the current chapter.
Computer hardware comes in two flavours: sequential and parallel. In the former, computer instructions are executed in sequence, and in the latter they are executed simultaneously. A sequential computer is driven by a single processor, it is therefore called a uniprocessor. A parallel computer can have many processors on board, hence the term multiprocessor. On the other hand, by a parallel computer we could be referring to uniprocessors connected together by a network such as a LAN. The concept of parallelism is therefore meaningful only in the context of multiprocessors or networked uniprocessors. Parallelism means that two or more processes or *threads* actually run at the same time on different CPUs. Parallelism is attained through *parallel programming*. On networked computers, parallel programming requires a *computing language* and a *coordination language*. The former allows computation and manipulation of data, the latter allows creation of simultaneous activities and their communication and control. For uniprocessors, a meaningful concept is *concurrency*. Concurrency means that two or more processes (threads) can be in the middle of executing code at the same time. They may or may not be actually executing at the same time, but they are in the middle of it (i.e, one started executing, it was interrupted, and the other one started) .
For the TM3w algorithm, the programming is done for a uniprocessor. The goal here is attaining concurrency.

6.1 Shared Address Space

As seen in the last section of chapter 5, the C functions defining the combination generators require *three* parameters to be passed. Each parameter takes a value that is returned by each of the three Fibonacci generators, and the are countably infinite values return.

As a procedural language, C does not allow an infinite sequence of values to be returned from the functions. Indeed it does not allow a return from within an infinite loop. Three C language constructs, **break**, **continue**, and **return** cannot be used to accomplish returning infinitely many values from a particular function. It would have been convenient if these constructs allowed breaking and then later continuing from the same point.
As shown in figure 6.1, the three combination generator functions need to access values which they share. In other words they all have to see the same memory. The prob-

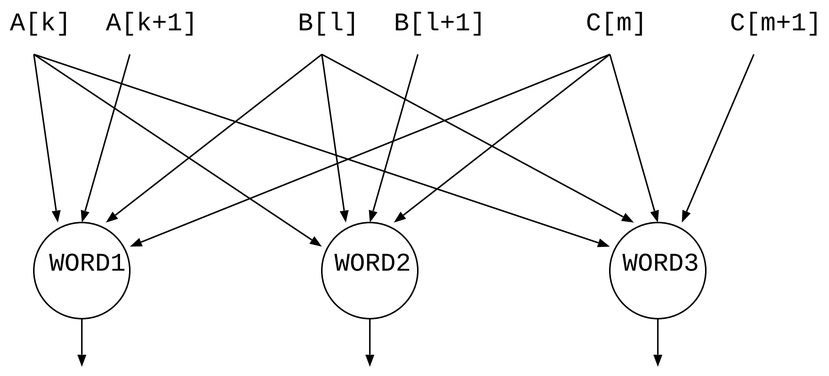

Figure 6.1: Accessing shared data

lem is, how can the functions, denoted $Word1$, $Word2$, and $Word3$ in the figure, have equal access to same memory locations without evoking *segmentation faults*. These are run-time errors resulting from programs attempting to access memory not allocated to them and *core dumping* with a segmentation violation error message. These violations are sometimes called bus errors and often result from an improper usage of pointers. For this reason, the solution where three Fibonacci generators are called from within each combination generator function makes for a *bad* solution.

6.2 Producer-Consumer Problem

The problem of figure 6.1 is a synchronization problem: how to control access of reads and writes to buffers. Figure 6.2 is a refined version of 6.1 and shows what is commonly referred to as *producer-consumer* problem. In this case three producers are creating data items that are processed by three consumers. For our implementation, both the consumers and producers will be *threads*.[1] The data items are passed between the producers and consumers using some type of IPC (interprocess communication).
For the problem of figure 6.2, the form of IPC used is the *named pipe* or FIFO (first in, first out). This is a special type of a UNIX pipe that has a pathname associated with

[1]POSIX.1c defines a threads as: "A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, errno value, thread-specific key/value bindings, and the required system resources to support flow of control. Anything whose address may be determined by a thread ... shall be accessible to all threads in the same process"

it and that can be used for IPC between unrelated processes. A FIFO is half-duplex, meaning that it provides for a one-way flow of data only. The three FIFOs for IPC are

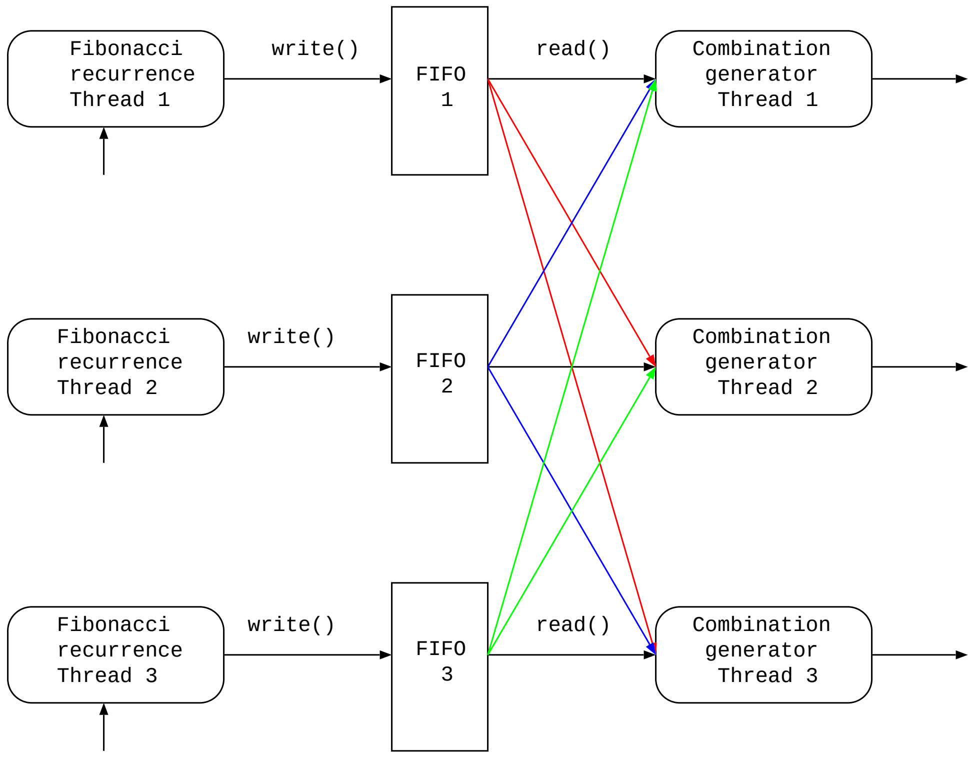
Fibonacci recurrence Thread 1
write()
FIFO 1
read()
Combination generator Thread 1
Fibonacci recurrence Thread 2
write()
FIFO 2
Combination generator Thread 2
Fibonacci recurrence Thread 3
write()
FIFO 3
read()
Combination generator Thread 3

Figure 6.2: Producer-Consumer problem

created by a function whose prototype is
`int mkfifo(const char *`*pathname*`, mode_t` *mode*`),`
where *pathname* is a normal Unix pathname, which is the name of the FIFO. The three FIFOs are created in a similar manner. The code below show the creation of the first FIFO.

```
#define FIFO1 ‘‘/tmp/fifo.1’’

int main(){
  int fif;
 if (access(FIFO1, F_OK) == -1){
   fif = mkfifo(FIFO1, FILE_MODE);
     if (fif != 0){
   fprintf(stderr, ‘‘could not create fifo %s\n’’, FIFO1);
   exit(EXIT_FAILURE);
  }
}
 ...
unlink(FIFO1);
}
```

The routine `access()` with the F_OK constant defined as zero, tests for the existence of a file called FIFO1. FILE_MODE specifies the permission bits which will be modified by the file creation mask of the process. The `unlink()` system call removes the FIFO after it is used.
Our three FIFOs are created within the `main()` function, and exist as *named* files. They can then be opened by any process, using the same **open()** and **close** functions used with files. The `open` call is passed the path name of the FIFO, e.g `/tmp/fifo.1`, and a flag indicating the mode in which it is to be opened .
Our implementation opens these FIFOs for writing from within secondary modules that contain functions defining the Fibonacci generators. As mentioned previously, FIFOs are half-duplex and therefore *must* be opened for writing in the `O_WRONLY` mode to prevent producer threads from reading back their output from pipes.

6.2.1 Producer and Consumer Threads

Traditionally, UNIX processes represent programs each having a single thread of control that has sole possession of the process memory and other resources. In order to attain concurrency in an application, a set of cooperating sequential tasks each assigned to a specific aspect of the problem is required. The cooperating tasks are implemented by dividing the application into multiple threads. In particular, an application is allowed to establish concurrent threads of control within a *single process*. These threads of control share process memory, file descriptors, and other resources. To attain concurrency for the TM3w generator, the POSIX[2] Pthreads interface standard is used. This standard specifies a set of thread facilities and introduces new application program interfaces in areas like thread creation, thread synchronization, thread cancellation *et cetera*.
Each of the three Fibonacci recurrence routines, described in the previous chapter, is executed by a separate thread. These threads are created using the `pthread_create` function from the POSIX thread library. The sample code for thread creation is,

```
/*include other relevant header files*/
#include <pthread.h>

int main() {

 int res1, res2, res3;
 pthread_t prod1, prod2, prod3;

 /* starting producer threads*/
 res1 = pthread_create(&prod1, NULL, fib_recurrence_1, param1);
```

[2]Other thread libraries exist. See the next chapter.

```
 if (res1 != 0){
   perror(``Producer 1 thread creation failed'');
    exit(EXIT_FAILURE);
 }
  res2 = pthread_create(&prod2, NULL, fib_recurrence_2, param1);
 if (res1 != 0){
   perror(``Producer 2 thread creation failed'');
    exit(EXIT_FAILURE);
 }
  res3 = pthread_create(&prod3, NULL, fib_recurrence_3, param1);
 if (res1 != 0){
   perror(``Producer 3 thread creation failed'');
    exit(EXIT_FAILURE);
 }

exit (0);
```

The first argument of the thread creating function is a pointer pointing to a variable to which a thread identifier is written. The identifier enable references to a particular thread. Since no special thread attributes are needed, the next argument is set to NULL. The final two arguments are the function that the thread should start executing, as well as the argument to be passed to this function. When the producer threads have been created, each of them will write to a separate FIFO. Since a FIFO is a name pipe, it can be opened by processes other than the one that created it. In this case the FIFO will be opened from within the modules that contain the Fibonacci recurrences.
Within the same `main()` function, consumer threads are started immediately after the producer threads. This is necessary to allow the consumers to process the data as it is being generated. Each of the consumer functions takes three parameters. therefore in order to create threads for each, a `struct` must be defined whose members are the parameters of the consumer functions. Such a structure is defined in a header file as

```
 struct {
   unsigned long *fib1;
   unsigned long *fib2;
   unsigned long *fib3;
 }
```

The consumer threads are started by the code,

```
  int word1, word2, word3;
  pthread_t cons1, cons2, cons3;
  struct three_args *point;

 word1 = pthread_create(&cons1, NULL, word_1_generator, (void
 *)point);
```

```
if (word1 != 0){
  perror(‘‘Consumer 1 thread creation failed’’);
   exit(EXIT_FAILURE);
}

 word2 = pthread_create(&cons2, NULL, word_2_generator, (void
*)point);
if (word2 != 0){
  perror(‘‘Consumer 2 thread creation failed’’);
   exit(EXIT_FAILURE);

 word3 = pthread_create(&cons3, NULL, word_3_generator, (void
*)point);
if (word3 != 0){
  perror(‘‘Consumer 3 thread creation failed’’);
   exit(EXIT_FAILURE);
}
```

After the producer and consumer threads are created, the main thread must wait for them by calling the `pthread_join()` function whose arguments are a pointer to a particular producer or consumer function and NULL. The consumer threads must then be synchronized with the producer threads to ensure that the former process *only* the data that have already being stored by the latter. This is considered in the next section.

6.3 Writing and Reading a FIFO

The way the functions `write()` and `read()` behave is affected by the manner in which a FIFO is opened. The `open_flag`, which is the second parameter to `open()`, has four legal combinations of `O_RDONLY`, `O_WRONLY`, and the O_NONBLOCK flag. If a FIFO is opened as:

♠ `open(const char *path, O_RDONLY)`, the open call does not return until a process open the same FIFO for writing.
♠ `open(const char *path, O_RDONLY | O_NONBLOCK)`, then open call returns immediately, even if a FIFO had not been opened for writing by any process.
♠ `open(const char *path, O_WRONLY)`, the open call does not return until a process open the same FIFO for reading.
♠ `open(const char *path, O_WRONLY | O_NONBLOCK)`, in this case open returns immediately and if no process has the FIFO open for reading, an error (-1) will be returned.

For this TM3w implementation, although each producer writes to a *separate* FIFO, it will be seen later that there is a level of synchronization that is still required. We open a FIFO with a `open(const char *path, O_WRONLY)` which does not return until another process open the same IPC for reading. This will help in the synchronization of consumer threads as will be explained in the following section. The required ordering is for producer 1 to write to the first FIFO and have a consumer thread read therefrom, then for producer 2 to write to the second FIFO and have a consumer thread read it, and lastly producer 3 writes to the third FIFO and a consumer thread reads it. The producers open and then write to a FIFO as shown below,

```
#define FIFO
#define BUFFER_SIZE
#define MEGA

unsigned long *producer_function(){
   int pipe_fd, res;
   int byte_sent = 0;
   unsigned long buffer[BUFFER_SIZE];

 pipe_fd = open(FIFO, O_WRONLY);
  if (pipe_fd != 1){
    while (bytes_sent < MEGA){
      res = write(pipe_fd, buffer, BUFFER_SIZE);
   if (res == -1){
     fprintf(stderr, ``Write error on pipe'');
     exit(EXIT_FAILURE);
   }
   bytes_sent += res;
  }
 (void)close(pipe_fd);
 }
 else {
  exit(EXIT_FAILURE);
}
```

The fact that without the `O_NONBLOCK` flag `open()` does not return from a call partially takes care of synchronizing the producer threads. Partially because we need each producer to write a *single* integer value to a buffer at a time. This means explicit synchronization is required for producers in the form of *condition variables*. Before considering issues of synchronization we need to look at how each consumer reads data from the buffer. Each consumer must open each of the three FIFOs in the `O_RDONLY` mode without the `O_NONBLOCK` option so that the call does not return until a producer writes to the buffer. This is accomplish as below,

```
#define FIFO1
#define FIFO2
#DEFINE FIFO3
#define BUFFER_SIZE PIPE_BUF /*defined in limits.h*/
```

```
 unsigned long buffer1[BUFFER_SIZE];
 unsigned long buffer2[BUFFER_SIZE];
 unsigned long buffer3[BUFFER_SIZE];

void consumer_function(){
  int pipe_fd1, pipe_fd2, pipe_fd3;
  int res1, res2, res3;
  int bytes_read = 0;

 memset(buffer1, 0, sizeof(buffer1));
 memset(buffer2, 0, sizeof(buffer2));
 memset(buffer3, 0, sizeof(buffer3));

 pipe_fd1 = open(FIFO1, O_RDONLY);
 pipe_fd2 = open(FIFO2, O_RDONLY);
 pipe_fd3 = open(FIFO3, O_RDONLY);
  if (pipe_fd1 != -1 && pipe_fd2 != -1 && pipe_fd3 != 1){
   do {
       res1 = read(pipe_fd1, buffer, BUFFER_SIZE);
       bytes_read += res;
       res2 = read(pipe_fd2, buffer2, BUFFER_SIZE);
       bytes_read2 += res2;
       res1 = read(pipe_fd3, buffer3, BUFFER_SIZE);
       bytes_read3 += res3;
   } while (res1 > 0 && res2 > 0 && res3 > 0);
   close(pipe_fd1);
   close(pipe_fd2);
   close(pipe_fd3);
  }
  else {
   exit(EXIT_FAILURE);
 }
...
```

The code above shows that each consumer will read a data item from each of the FIFOs. Reading when the three ANDed conditions are satisfied mean that calls to `open()` will return immediately since producers will have written to the buffers. A *single* mutex lock can then be used to lock the area where the reads are carried out. Condition variables are *always* associated with a mutexes. Typically mutex provides exclusive access to shared data . Each of our producers has exclusive access to each of the FIFOs. In this case a producer thread will lock the mutex and then use a condition variable to wait for some condition (e.g., buffer not full) to become true. After writing to a buffer, the thread will use another condition variable to signal that it has caused some other condition (e.g., buffer not empty) to become true, and then unlock the mutex.

6.4 Synchronization

6.4.1 Producer to Consumer

Mutexes are for *locking* and condition variables are for *waiting*. Both these mechanisms are required for synchronizing producers and consumers of the TM3w generator. For condition variables, POSIX implements the following APIs:
`pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr )`, and
`pthread_cond_signal(pthread_cond_t *cptr)`.
The three producer threads write to three buffers from which a single consumer thread reads. For each of the producer threads and a consumer thread, use of the bounded buffers need to be coordinated. Between each producer and a consumer two condition variables (non-empty and non-full) and a mutex are defined. The mutex protects the buffer and its attributes which are, count of unconsumed characters, producer's index, and consumer's index. For each of the three producers the execution cycle is as follows:
1. Produce a data item.
2. Lock the mutex.
3. Check the condition that the count of unconsumed data items in the buffer is less than the number of slots in the buffer. If the condition is false, wait on the condition variable non-full and then repeat the check upon return from the wait. When the condition is true, go to 4.During a wait on the condition variable, a mutex is unlocked.
4. Put a data item in the buffer, and increment both the producer's index into the buffer and the count of unconsumed data items in the buffer.
5. If the count of unconsumed data items is noe equal to one, inform the consumer by signaling via the condition variable non-empty. If the signal was blocked on non-empty, this should unblock it.
6. Unlock the mutex.

For the consumers the execution cycle is as follows:
1. Lock the mutex.
2. Check to see if the counter of data items ready for processing to be nonzero.
3. If the value is zero, `pthread_cond_wait()` is called to put a thread to sleep.
4. The mutex is unlocked and the thread is put to sleep until some other thread calls `pthread_cond_signal` for this condition variable.
5. Before returning, `pthread_cond_wait()` locks the mutex.
6. When `pthread_cond_wait()` returns, a data item is processed and the counter is decremented.

6.4.2 Consumer to Consumer

Each consumer requires access to the three FIFOs shared by *all* the consumers. As already mentioned, a consumer will lock the critical area, process the data, and then release the mutex for the next consumer thread to use it. Figure 6.3 is a sequence which would ensure

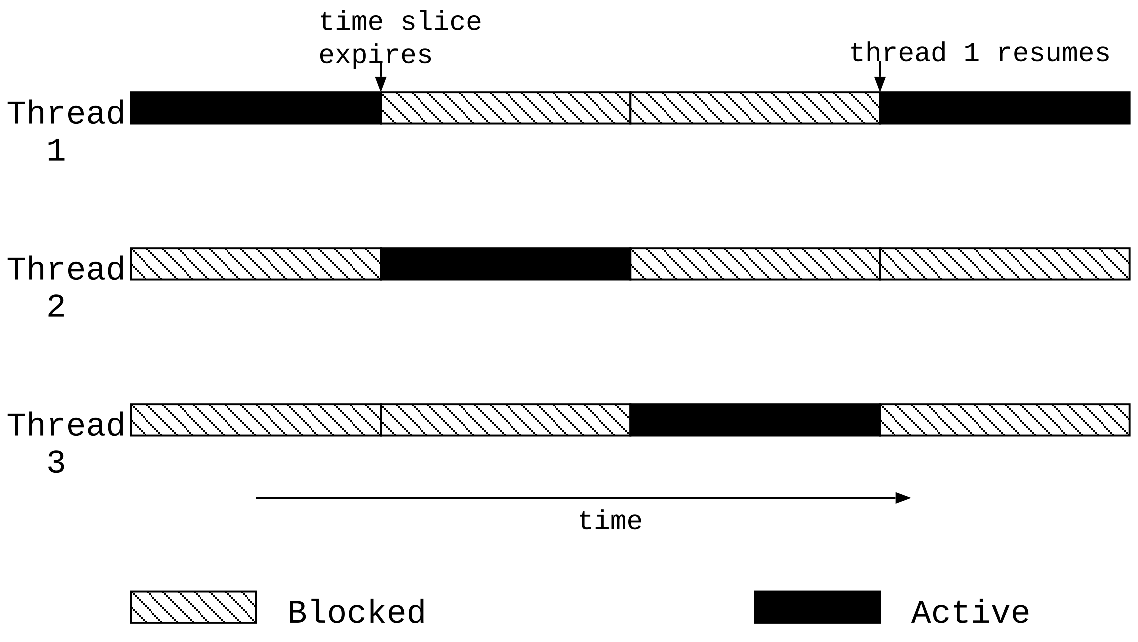

Figure 6.3: Threads interleaving execution on a uniprocessor

that each consumer thread gets a chance to access and process data in the shared memory area. *Assuming* that there is a guaranteed order in which locks are acquired, each thread will:
1. Lock the mutex.
2. Handle buffer attributes.
3. Perform a computation and return result.
4. Release the mutex lock.

All the preceeding sections are a description of the intended functionality for a concurrent implementation. The observed functionality diverges significantly from the expected as explained in the next section.

6.5 Unresolved Problems

The intended functionality of the proram was to perform computations in a concurrent manner on a uniprocessor machine and output results an infinite number of times as computed by threads in the order: thread 1, thread2, thread 3. The problem is that there is *no* guaranteed order in which mutex locks are acquired. On its own, each of the combination generator outputs a countably infinite sequence of 32-bit integers. The attempt here was to make each generator output *one* integer, and then pass control to the next generator thread in a circular fashion starting with generator 1 to generator 3. ♠ Because each generator computes consecutive values in a very small time, as each thread

releases a mutex lock a small amount of time passes before it has to reacquire it again. This results in a thread that was holding the lock, reacquiring it again and *preventing* other threads from acquiring the lock. This is because nothing blocks the thread holding the lock, and so it continues to run from the time it releases the lock until it reacquires the same lock. In this way no other thread gets to run. POSIX, whose APIs were used here, *does not* have a concurrency setting feature like Solaris.
♠ Given that there is no guaranteed order for acquiring mutex locks, how to implement POSIX's priority scheduling policies to solve this problem.
♠ POSIX thread library provides no time-slice mechanism for scheduling threads. This is similar to the first point in that if a single thread is active on the CPU and does not block, it can run indefinately.

The producer-consumer problem with three on each side and in which each consumer is fed on the output of all the producers was only partially solved. The implementation worked if all producers fed a *single* consumer at a time, and the program deadlocks if all producers are in the cycle. This is clearly due to failure to properly time and synchronize the producer threads.

Chapter 7

Portability of the Implementation

The term *portability* refers to the ease with which a program's source code can be moved from one computing platform to the other. This chapter outlines factors that affect the portability of the TM3w algorithm's C implementation. The kinds of portability that are of relevance in this regard are:

♠ Hardware portability.
♠ Operating system portability.
♠ Compiler portability.

From the discussion that follow it will be seen that the implementation for the TM3w algorithm requires significant changes for portability between, say, a PC, a UNIX workstation, and a Macintosh. But for all UNIX type operating systems running on 32-bit or 64-bit platforms, the implementation *should* be easily portable.

7.1 Hardware

The programs for the TM3w generator were developed on two workstations, one driven by a 32-bit Ultra SPARC IIi processor and running Solaris, the other driven by a POWER3-II processor and running AIX. Computations performed by the algorithm are single precision and during bit rotations a lot of bit setting and bit getting is carried out. The 4 bytes (32 bits) word length in both machines is convenient. Figure 7.1 shows two ways in which byte storage is accomplished in memory for different processor architectures. For little-endian storage, the least significant bits (LSBs) of variables are stored at the lowest addresses. For big-endian the most significant bits (MSBs) of variables are stored in the lowest addresses. On both machines host byte ordering is *big-endian*. This is significant because in *all* the cases of parameter passing to functions, pointers are used. For the same seed vector, different host byte orderings imply different generator initial states.
The functions defining each of the combination generators has the parameters in its argument list. The `pthread_create()` function takes as one of its arguments, a function with only *one* parameter in its argument list. In order to create threads to execute consumer routines, a structure was defined whose members are the three arguments of the consumer functions. For instance the prototype for one of the functions is

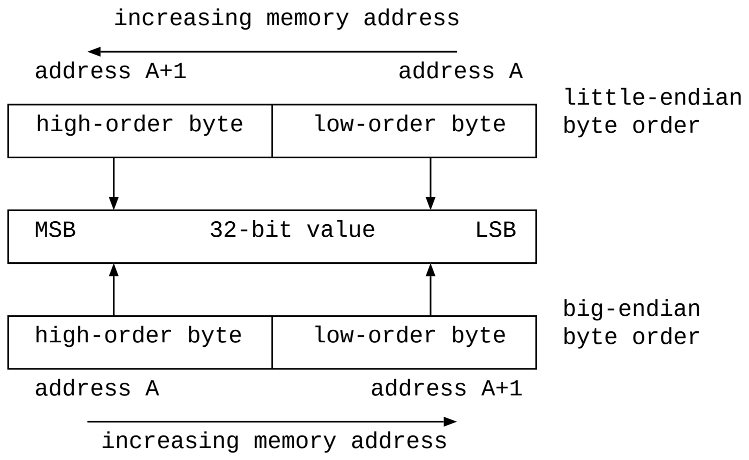

Figure 7.1: Byte order for a 32-bit integer

```
void word_1_generator(unsigned long *fib1, unsigned long *fib2,
unsigned long *fib3);
```

and the prototype for the function `pthread_create()` *without* the attributes parameter is

```
int pthread_create(pthread_t *thread, NULL, void
*(*some_function)(void *), void *arg);
```

As explained in chapter 6, a structure is created to hold the argument list of the `word_1_generator` function, say

```
 struct three_args {
         unsigned long *fib1;
         unsigned long *fib2;
         unsigned long *fib3;
 }
```

On both the SPARC IIi 650 MHz and the POWE3-II 450MHz a *word* is 64 bits. However when a pointer to the above structure is set up, its size *does* vary between the two processors, possibly because of how the compilers add padding bytes between the members of the structure.
Since word alignment is an important factor in the portability of structures, no assumptions were made about whether this alignment could be the same for both processors or not.

7.2 Operating System

On the SPARC IIi the operating system is Solaris (SVR 4.0) and on the PowerPC it is AIX-5. Both these operating systems are compliant with the X/Open and POSIX stan-

dards. To this end they both have thread support and provide the `pipe()` and `mkfifo()` functions.
Regarding threads, there exist *four* threading libraries: POSIX, Solaris, OS/2, and Windows NT. The SVR 4.0 has both the Solaris and POSIX thread libraries as part of the normal system library. AIX has a POSIX threads library. The application programming interfaces (APIs) of Solaris and POSIX have a number of differences. The most relevant for the TM3w implementation is that Solaries *is able to set and get a level of concurrency* whereas POSIX has *priority scheduling*. For a multithreaded application the Solaris API, `thr_setconcurrency()`, tells the system how many threads to run concurrently. A call to this function gives each of the multiple threads a chance to execute. POSIX does not implement `thr_setconcurrency` but has scheduling policies. With these, the priority of a thread can be set and retrieved from an attribute object by the `pthread_attr_setschedparam()` and `pthread_attr_getschedparam()` respectively. For the consumer threads of figure 6.2 , there are two policies from which to choose to do thread scheduling. These are SCHED_FIFO and SCHED_RR. The SCHED_FIFO defines a first in, first out scheduler. Each priority that can be assigned to a thread is associated with a FIFO queue. As each thread becomes runnable, it is added to the associated priority queue. As the thread in a given priority moves to the head of the queue, it is scheduled on the next available processor.
The SCHED_RR policy defines a round-robin scheduling algorithm. The round-robin policy is similar to the SCHED_FIFO policy, except for the addition of a time quota associated with each thread. As the thread for a given priority is running, if its time quota is used up, it is put back on the tail of the associated queue.
When writing to a FIFO, the size thereof is an important consideration. Both systems impose a limit on how much data can be in a FIFO at any one time. This limit is the #define PIPE_BUF found in the header file limits.h. On Solaris this constant is 5120 bytes, and on AIX it is 32768 bytes. Both systems define the constant _POSIX_PIPE_BUF as 512 bytes. The system guarantees that writes of PIPE_BUF or fewer bytes on a FIFO that has been opened O_WRONLY will either write all or none of the bytes.
Although Solaris and POSIX are virtually similar, moving from one API to the other presents difficulties to the programmer who is not thorougly familiar with one or the other. As far as this author could tell, the Windows 2000 and Windows XP operating systems do not implement the POSIX API.

7.3 Compiler

Compilers present on the systems used in the development of code are the ANSI compiler cc and the GNU compiler gcc. Both compilers are available on Solaris and only the latter is available on AIX. The compilation enviroment that meets the ISO (International Organization for Standardization) C language stand defines the preprocessor symbol _STDC_. For the ANSI compiler this preprocessor symbol is defined as 1. For the GNU compiler the following bit of code is added to enable standard C compilation.

```
#ifdef _STDC_
```

```
  int some_function(parameters);
#else
  int some_function();
#endif
```

The code written is meant to be compliant with the POSIX and X/Open standards. For this reason the symbol _XOPEN_SOURCE is defined before including any header files. This definition is meant to modify the behaviour of the included files to bring them in line with the X/Open standard. In chapter 2 it was mentioned that the algorithm is data size neutral, i.e. words can be 16, 32, or 64 bits. This is in theory. In practice the C programming language does not provide a mechanism for adding new fundamental data types. For 64-bit addressing and integer arithmetic capabilities, the mappings of the existing data types have to be changed or new data types have to be added to the language. The ISO/IEC 9899 (1990) Programming Languages - left the definition of **short**, **int**, **long int**, and **pointer** delibarately vague to avoid artificially constraining hardware architectures. Both the ANSI and GNU compilers on the machines used by the author use the ILP32 data model in which

$$sizeof(int) = sizeof(long) = sizeof(pointer)$$

Thus for both the ANSI and GNU compilers, TM3w's single precision computations are performed based on the ILP32 programming model. The extra 32 bits of data space on the 64-bit processors appear to be wasted.
Lastly, the password to seed vector conversion does not assume any ASCII encoding. Characters on a particular machine are converted to their decimal or hexadecimal equivalents based on the encoding used by that machine. The (implicit) requirement is that the size of the character's value *cannot* be larger than the size of the `char` type. So that in an 8-bit system, 255 is the maximum value that can be stored in a single `char` variable. The next requirement is that each character must be represented by a positive number. Therefore the portable characters within the ASCII character set are those from 1 to 127. The extended characters are *not* guaranteed to be portable because the `signed char` has only 127 positive values.

Appendix A

Test Distributions

Test Distributions

Definition. A *random variable* is a mapping from a sample space to the values space. If this mapping assumes a finite number of different values in an interval (or in several distinct intervals), it is called *discrete*, otherwise it is *continuous*. With each value that a discrete random variable assumes, say $X = x_k$, is associated a probability, say $P(X = x_k)$, and denoted by $p_k(x),\ k = 0, 1, 2, \dots$.

Definition. The quantities $p_k(x)$ are jointly called the *probability function* of the random variable X.

The *distribution function* is then determined from the probability function by the equation

$$F_X(x) = \sum_{j \le x} p_X(j)$$

For a continuous random variable, the interval may be unbounded. In this case the distribution function becomes

$$F_X(x) = \int_{-\infty}^{x} f_X(t)dt$$

Definition. The function $f_X(x)$ is called a *probability density function*.

Definition. The solution $x = x_\alpha$ of the equation $F_X(x) = 1 - \alpha$ is called the α-percentile of the random variable X.

The normal distribution

Definition. A continuous random variable X has a *normal distribution* with *mean* μ and *variance* σ^2 if its probability density function is defined by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

When $\mu = 0$ and $\sigma^2 = 1$ the random variable X is said to have a *standard normal distribution*. Let the α-percentile of a standard normal distribution be denoted as x_α. Then the area under the density function to the *right* of x_α is equal to α, that is,

$$P(X > x_\alpha) = \alpha$$

Below is a table of selected percentiles of the standard normal distribution.

α	0.0005	0.001	0.005	0.01	0.025	0.05	0.10
x_α	3.29	3.09	2.58	2.33	1.96	1.64	1.28

The χ^2 distribution

Definition.A continuous random variable X has a χ^2 distribution with v degrees of freedom if its probability density function is defined by

$$f(x) = \begin{cases} \frac{1}{\Gamma(v/2)2^{v/2}} x^{(v/2)-1} e^{-x/2}, & 0 \leq x < \infty, \\ 0, & x < 0. \end{cases}$$

where Γ is the gamma function.[1] The χ^2 distribution is *nonsymmetric* and its shape depends only on the *number of degrees of freedom* v. Below is a table of selected percentiles for the χ^2 distribution.

v	α					
	0.100	0.050	0.025	0.010	0.005	0.001
1	2.7055	3.8415	5.0239	6.6349	7.8794	10.8276
2	4.6052	5.9915	7.3778	9.2103	10.5966	13.8155
8	13.3616	15.5073	17.5345	20.0902	21.9550	26.1245
31	41.4212	44.9853	48.2319	52.1914	55.0027	61.0983

[1]The *gamma function* is defined by $\Gamma(t) = \int_0^\infty x^{t-1}e^{-x}dx$, for t ¿ 0.

Bibliography

[1] N. S. Altaman, "*Bit-wise Behaviour of Random Number Generators*", SIAM J. SCI. STAT. COMPUT., vol. 9, **5**, (1988).

[2] R. J. Anderson, "*On Fibonacci Keystream Generators*", K. U. Leuven Workshop on Cryptographic Algorithms, Springer-Verlag, 1995.

[3] S. L. Anderson, "*Random number generators on vector supercomputers and other advanced architectures*", SIAM Rev. **32** (1990).

[4] M. Blum, S. Micali, "*How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits*", SIAM Jour. on Computing, vol 13, 1984.

[5] J. Boyar, "*Inferring Sequences Produced by Pseudo-Random Number Generators*", Jour. of ACM, vol 36, no 1, 1989.

[6] G. Brassard, P. Bratley, "*Fundamentals of Algorithmics*", Prentice-Hall Inc., (1996).

[7] R. P. Brent, "*On The Periods Of Generalized Fibonacci Recurrences*", Math. Of Comp., vol. 63, No. 207, July 1994.

[8] R. P. Brent, S. Larvala, P. Zimmermann, "*A Fast Algorithm for Testing Irreducibility of Trinomials mod 2*", Report PRG-TR-13-00, Programming Research Group, 30 Dec. 2000.

[9] W. G. Chambers, "*On Random Mappings and Random Permutations*", K. U. Leuven Workshop on Cryptographic Algorithms, Springer-Verlag, 1995.

[10] D. Coppersmith, H. Krawczyk, Y. Mansour, "*The Shrinking Generator*", Advances in Cryptology - CRYPTO '93, Springer LNCS v 773.

[11] D. Gollmann, W. G. Chambers, "*Clock-controlled shift registers : A review*", IEEE J. Selected Areas Commun, vol 7, May 1989.

[12] S. W. Golomb, "*Shift Register Sequences*", Aegean Park Press, (1967).

[13] J. Hastad and A. Shamir, "*The Cryptographic Security of Truncated Linearly Related Varables*", Proceedings of the 17th Annual ACM Symposium on the Theory of Computing, 1985.

[14] F. James, "*A review of pseudorandom number generators*", Comput. Phys. Comm., **60** (1990).

[15] S. Kleiman, D. Shah, B. Smaalders, "*Programming with THREADS*", SunSoft Press, Mountain View, California, 1996.

[16] D. E. Knuth, "*The Art of Computer Programming - Seminumerical Algorithms*", vol. 1 3rd ed., Addison Wesley Longman, (1997).

[17] D. E. Knuth, "*The Art of Computer Programming - Seminumerical Algorithms*", vol. 2 3rd ed., Addison Wesley Longman, (1998).

[18] H. Krawczyk, "*How to Predict Congruential Generators*", Advances in Cryptology - CRYPTO '89 Proceedings, Springer-Verlag, 1990.

[19] B. Lewis and D. J. Berg, "*THREADS PRIMER - A Guide to Multithreaded Programming*", SunSoft Press, Mountainview, California, 1996.

[20] G. Marsaglia, "*A Current View of Random Number Generator*", Computer Science and Statistics : The Interface, L. Billard, ed. (Elsevier, Amsterdam, 1985).

[21] A. J. Menezez, P. C. van Oorschot, S. A. Vanstone, "*Handbook Of Applied Cryptography*, CRC Press, (1996).

[22] C. H. Meyer and W.L. Tuchman, "*Pseudo-Random Codes Can Be Cracked*", Electronic Design, v. 23, Nov., 1972.

[23] The Open Group, "*Threads and the Single UNIX Specification*", Version 2, 1997.

[24] M. O Rabin, "*Probabilistic Algorithms in Finite Fields*", SIAM J. on Computing, vol 9, 1980.

[25] B. Schneier, "*Applied Cryptography*", John Wiley & Sons Inc., (1996).

[26] W. R. Stevens, "*UNIX NETWORK PROGRAMMING - Interprocess Communication*", Vol. 2, 2nd Ed., Prentice Hall International Inc., 1999.

[27] R. Stones and N. Matthew, "*Beginning Linux Programming*", 2nd Ed., Wrox Press Ltd, Birmingham, 2000.

[28] J. Viega and G. McGraw, "*Building Secure Software*", Addison-Wesley Professional Computing Series, 2002.

[29] E. J. Watson, "*Primitive Polynomials (mod 2)*", Math. Comp. **16** (1962).

[30] N. Zierler, "*On Primitive Trinomials*", Information and Control, **13**, (1968).

[31] N. Zierler, "*Primitive Polynomials whose Degree is a Mersenne Exponent*", Information and Control, **15**, (1969).