INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A STUDY OF THE BLOWFISH ENCRYPTION ALGORITHM

by

PATRICIA J. M. FINCH

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

1995

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9521269

Copyright 1995 by
Finch, Patricia J. M.
All rights reserved.

UMI Microform Edition 9521269
Copyright 1995, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI

300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

©1995

PATRICIA J. M. FINCH

All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

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

1/12/95 — Date

[signature] — Chair of Examining Committee

1/17/95 — Date

[signature] — Executive Officer

Professor Izidor Gertner

Professor Stanley Habib

Harold Finz, Ph.D.

Supervisory Committee

The City University of New York

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

A Study of The Blowfish Encryption Algorithm

by

Patricia J. M. Finch

Advisor: Professor Michael Anshel

Bruce Schneier published his Blowfish Encryption algorithm in the April, 1994 issue of *Dr. Dobb's Journal.* I have done an analysis of the encryption portion of this algorithm and have obtained an implementation-independent form of it which can be used to model changes in overhead in either the Blowfish algorithm or in an implementation of it.

Blowfish is a new block-encryption algorithm. It takes as input a 64-bit block of plaintext and returns a 64-bit block of ciphertext. The algorithm is constructed as a Feistel network of 16 rounds. Within each round, the Blowfish function F is executed. The Blowfish function F takes as input a 32-bit string. This input string is broken up into four 8-bit strings. Each of these 8-bit strings is input to its own 8-in 32-out S-box. Additional manipulations of the data are performed and the single result is returned to the Feistel network.

Key generation is lengthy; however, if long messages are encrypted the overhead of the key generation diminishes rapidly. Once the keys have been generated, the performance of the entire encryption portion of Blowfish is heavily dependent on the performance of the Blowfish function F. For this reason, I concentrate on the Blowfish function F.

Blowfish is a heavily implementation-dependent algorithm. In order to gain a better understanding of Blowfish, I remove many of these dependencies. Among other tools, the methods of both parallel and sequential algorithms are used to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

determine work and time according to the dictates of each kind of analysis. For the parallel analysis, I use the perspective of the Work-Time framework.

As a result of the various analyses and removal of dependencies, I have come up with a general form or template for Blowfish. When this template is instantiated, changes to the Blowfish algorithm itself can be exactly measured whereas changes in the implementation of Blowfish are only substantially measured. Three examples are given in support of my template's ability to predict changes in both implementation and modification of the algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

I would like to acknowledge the help of the following people for the stated reasons.

Professors Michael Anshel and Stanley Habib for help above and beyond the call of duty in expediting the completion of this dissertation

Mr. Lyle Bingham of Computer System Architects for his generous help relating to transputers

Mr. Bruce Schneier for graciously helping me with the cryptography aspects of his Blowfish Encryption Algorithm

Professor Shermane A. Austin whose own dissertation layout was so attractive that I based mine on hers

Mrs. Nancy Thalblum for all of the beautiful graphic artwork in this dissertation

Miss Susan N. Levy for her editorial assistance

and

Mr. Leonard Soifer for living through this time of chaos with me, ducking for cover when necessary

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Chapter 1	Introduction	11
1.1	What this Dissertation is About	11
1.2	Overview of the Blowfish Algorithm	11
1.3	Why Are We Examining Blowfish?	12
Chapter 2	The Original Blowfish Algorithm	14
2.1	The Feistel Network	14
2.2	The Blowfish Function F	16
2.3	The Blowfish Algorithm as a Whole	17
Chapter 3	Analysis of the Original Algorithm	20
3.1	Analysis of the Original Blowfish Algorithm	20
3.2	The Feistel Network	23
3.3	Analysis of the Sequential Encryption Portion of Blowfish as a Whole	25
Chapter 4	A Parallelized Version of Blowfish	26
4.1	Parallel Analysis of the Original Blowfish Algorithm	26
Chapter 5	The Implementation-Independent Blowfish Algorithm	34
5.1	Identifying the Dependencies	34
5.2	The Implementation-Independent Algorithm	36
Chapter 6	Algorithmic Analysis of the Implementation-Independent Blowfish Algorithm	40

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.1	Sequential Implementation-Independent Blowfish Function F	40
6.2	Parallel Blowfish Function F	44
6.3	Sequential Implementation-Independent Feistel Network	47
6.4	Parallel Implementation-Independent Feistel Network	53
6.5	Where Do We Go from Here?	57
Chapter 7	Occam and the Transputer	58
7.1	The Transputer	58
7.2	Occam	60
Chapter 8	Examples	62
8.1	A Variant Blowfish Function F	62
8.2	Parallel Blowfish Implemented on Transputers	65
8.3	What Have We Learned From These Examples?	75
Chapter 9	Observations and Conclusions of the Analysis	76
9.1	Results	76
9.2	Limitations and Questions for Further Study	77
9.3	Summary	77
Appendix A	The Original Blowfish Paper	79
Appendix B	Figures Relating to the Blowfish Encryption Algorithm	85
Appendix C	Figures Relating to Occam and the Transputer	91
References		95

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

TABLE	TITLE	PAGE
4.1	A Parallel Feistel Network for the Blowfish Encryption Algorithm	26
4.2	A Parallel Blowfish Function F	28
4.3	Sequential Execution of the Feistel Network	30
4.4	Sequential Blowfish Function F	33
6.1	Sequential Implementation-Independent Blowfish Function F	41
6.2	Parallel Implementation-Independent Blowfish Function F	45
6.3	General Implementation-Independent Blowfish Function F	46
6.4	Sequential Implementation-Independent Feistel Network	48
6.5	Measurements in Terms of Work and Time Using the Sequential Implementation-Independent Feistel Network With Four or More Rounds	52
6.6	Parallel Implementation-Independent Feistel Network	53
6.7	Obtaining a General Implementation-Independent Feistel Network From the Sequential and Parallel Feistel Networks	56

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

FIGURE	TITLE	PAGE
2.1a	16 Rounds of Encryption in Blowfish	15
2.1b	The Remainder of the Encryption	15
2.1, parts a and b	Pseudocode Giving the Encryption Portion of The Blowfish Encryption Algorithm	15
2.2	Pseudocode Giving Schneier's Blowfish Function F	17
5.1	Pseudocode for the Implementation-Independent Blowfish Function F	39
8.1	A Variant Blowfish Function F	63
8.2	A Network of Four Transputers	66
B.1	Blowfish is a Feistel Network Consisting of 16 Rounds	86
B.2	A Round in the Feistel Network	87
B.3	Blowfish Function F	88
B.4	A Variant Blowfish Function F as a 4-in 32-out Function	89
B.5	Schneier's Revised C Code for the Blowfish Function F	90
C.1	Layout of CSA Transputer Kit Board	92
C.2	Transputer Links and the Corresponding Occam Channels	93
C.3	On-chip Scheduler	94

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

INTRODUCTION

Section 1.1 What This Dissertation is About

This dissertation examines the encryption portion of Bruce Schneier's Blowfish Encryption Algorithm. As originally published, this algorithm is highly sequential in nature. The algorithm is viewed in such a way that the sequential nature of the implementation in the original presentation of the algorithm does not matter. Both the traditional sequential and the parallel Work-Time framework analysis methods are used. The purpose of looking at this algorithm from both the sequential and parallel analysis views is to show that both have meaning for this algorithm. Indeed, we find that we can arrive at a generalized, implementation-independent version of the algorithm. The generalized version of this algorithm can then be used as a model to measure the differences incurred for differing implementations.

Section 1.2 Overview of the Blowfish Algorithm

Bruce Schneier published his Blowfish Encryption Algorithm [Schneier94b] in the April, 1994 issue of *Dr. Dobb's Journal.* This article, slightly modified, appears as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A. The algorithm is a block-cipher algorithm configured as a Feistel network with 16 rounds. The algorithm itself is split into two functional parts: key-expansion and data-encryption. We are only concerned with the data-encryption portion of the problem.

This is roughly how the encryption portion of the algorithm works. Data, called plaintext, enters the Feistel network. It is XORed with a key, some P_i , one of 18 such subkeys. The output of this XORing is input to the Blowfish function F where additional operations are done on the data. When the data exits the Blowfish function F it is returned to the Feistel network where it is XORed with the output of the most recent XORing with one of the P_i subkeys. This section of the execution within the Feistel network is termed a "round". Schneier's Feistel network structure [Schneier94b] is shown in Figure B.1. A round within the Feistel network is shown in Figure B.2.

Within the Blowfish function F, shown in Figure B.3, the data, now a function of both the keys and the data, is split up to enter one of four S-boxes. Each S-box, which really contains permuted keys, takes as input a portion of the data and gives as output a substitution and expansion dependent on both the input data to the S-box and the contents of the S-box. The S-box thus gives a key-dependent output. A property of S-boxes is that unique data input to a specific S-box gives a unique output.

Section 1.3 Why Are We Examining Blowfish?

As the Blowfish Encryption Algorithm is so heavily sequential, it becomes obvious that its performance as a whole depends on the efficiency of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Blowfish function F. As the major expense in Blowfish is time which is linear in clock cycles and space which is mainly to store the keys (about 5K bytes), analyzing this particular algorithm in detail gives essentially the same information arrived at just by reading Bruce Schneier's article [Schneier94b]. So, why did we analyze Blowfish in the first place?

Analyzing Blowfish in detail gave us the ability to develop more and more general versions of these two structures which comprise the algorithm. From each generalization we got a further generalization until we had gotten the most general form. What do we do with this general form? Why is it important?

While Blowfish uses small to minute resources, there are other kinds of algorithms to which this method might be applied. These other algorithms might not use minuscule amounts of resources. This analysis of Blowfish can be used as a model to be applied to situations where there are significant resource costs. This dissertation only goes into a specific instance of the derivation of such a model. If a systematic method can be developed through this analysis, it could be a major breakthrough in modeling tools as well as in analytic analysis of algorithms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

THE ORIGINAL BLOWFISH ALGORITHM

We are concerned only with the data-encryption portion of The Blowfish Encryption Algorithm. Thus, we need only concern ourselves with its two major structures: the Feistel network and the Blowfish function F.

Section 2.1 The Feistel Network

The input to the Feistel network is a 64-bit block of plaintext. This block of plaintext is split up into two 32-bit unsigned integers. Looking at Figure B.1 we see that when this 64-bit block of data enters the Feistel network it is split into these two 32-bit segments, one of which goes to the left and the other to the right. The pseudocode in figures 2.1a and 2.1b describes the encryption from here.

Decryption follows directly from the encryption algorithm in a very natural fashion. Our 64-bit input now becomes ciphertext. Our P_i subkeys are applied in reverse order, i. e., first P_{18} is applied, then P_{17} is applied, and so forth. The output from the Feistel network now becomes a 64-bit block of plaintext.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Note that the reversal of the last swap of x_L and x_R after the 16th iteration in the Feistel network plus the added keys P_{17} and P_{18} allow for complete symmetry, thus making encryption and decryption inverse operations.

For the first 16 iterations (or rounds) using the first 16 P_i subkeys, we do the following [Schneier94b] :

For i = 1 to 16

$x_L = x_L$ XOR P_i

$x_R = F(x_L)$ XOR x_R (1)

Swap x_L and x_R

Figure 2.1a

16 Rounds of Encryption in Blowfish

After the 16th iteration, we swap x_L and x_R to reverse the last swap. Then we use the last two subkeys, P_{17} and P_{18} as follows:

$x_R = x_R$ XOR P_{17}

$x_L = x_L$ XOR P_{18}

Figure 2.1b

The Remainder of the Encryption

Figures 2.1a and 2.1b

Pseudocode Giving the Encryption Portion of The Blowfish Encryption Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lastly, we recombine x_L and x_R to form a 64-bit block of ciphertext and we exit the Feistel network.

Section 2.2 The Blowfish Function F

The encryption step identified by (1) in Figure 2.1a above uses the Blowfish function F, $\left(F(x_L)\right)$. As we can see by looking at Figure B.1, the Blowfish function F is the most frequently executed piece of logic in the algorithm. Figure B.3 [Schneier94b] shows the logic of this function. Looking at this figure we see that the input to the Blowfish function F is a 32-bit string. This string is what was output when the most recent XORing of one of the P_i keys with some data took place within the Feistel network structure.

Looking at figure B.3 we see that the structure of the Blowfish function F accesses four S-boxes. Each S-box is actually a two-dimensional array. Each element in each of these S-boxes is 32 bits in length. Because the input data is eight bits long there must be 2^8 or 256 elements per S-box, each of them key-dependent. The contents of the S-boxes are initialized when all of the keys, both the P_i 's and the S-boxes, are generated. The value of each of the keys remains fixed throughout the encryption.

The data entering each S-box is 8 bits in length. For each S-box the output corresponding to an 8-bit input is unique. Functionally, the S-boxes expand, permute, and substitute based on the 8-bit input giving a 32-bit output. The S-boxes are said to be 8-in 32-out.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The output from the first two S-boxes is added mod 2^{32}. The result is then XORed with the output of the third S-box. This result is then added mod 2^{32} with the output of the fourth S-box. This result then goes back into the Feistel network.

Schneier's pseudocode [Shneier94b] for the Blowfish function F follows.

$$
\begin{aligned}
F(x_L) = \; & ((S_{1,a} + S_{2,b} \text{ mod } 2^{32}) \\
& \text{XOR } S_{3,c}) \\
& + S_{4,d} \text{ mod } 2^{32}
\end{aligned}
$$

Figure 2.2

Pseudocode Giving Schneier's Blowfish Function F

The only unknowns at the point where the Blowfish function F is to be executed are the values of the variables a, b, c, and d. These values are needed in order to determine which entry in each S-box is to be fetched.

Section 2.3 The Blowfish Algorithm as a Whole

Blowfish is a new block-cipher encryption algorithm, configured as a Feistel network of 16 rounds. Each round of the Feistel network (Figure B.2) includes the taking in of data which has already been partly encrypted, putting this data through the Blowfish function F, and coming out with data, again changed, which is returned to the Feistel network to be used in the next round. Because of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

way in which Schneier has configured his Feistel network, the encryption and decryption processes are inverse operations.

The Blowfish function F is the heart of the algorithm. It is used in each of the 16 rounds of the Feistel network and is therefore the most frequently executed block of logic in the algorithm. The 32-bit input to the Blowfish function F is broken up into four 8-bit pieces, each of which is input to a different S-box. There need not have been four S-boxes with the properties given above. However, in using four 8-bit segments of data as input to the S-boxes, symmetry can be put to use.

Each S-box entry is a 32-bit word. By necessity, using an 8-bit string as input to an S-box requires that we have 2^8 or 256 entries in an S-box. Each S-box deals with only eight bits of the input and there are 32 bits of input to the Blowfish function F so we need four S-boxes with the above properties in order to have all of the 32 bits of input processed. It is a property of S-boxes that unique input to an S-box gives unique output.

The initial overhead in using this algorithm is done when all of the P_i subkeys and the S-box keys are generated. The expense of this initialization is significant. The relative expense of the initial overhead decreases with the increasing size of the message to be encrypted.

Schneier has developed this algorithm to be optimal for execution on 32-bit word computer systems with significantly large on-chip cache (e.g., a Pentium or PowerPC system). The intent in having the large on-chip cache is that the keys, once generated, can be stored as a table in this cache. This 32-bit system's

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

optimality is fostered by having all operations in the algorithm be executed as word-length logical instructions and table lookups. As 32-bit systems are the largest word size in common use, this class of systems was chosen.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

ANALYSIS OF THE ORIGINAL ALGORITHM

Section 3.1 Analysis of the Original Blowfish Algorithm

Both sequential and parallel analysis techniques are being used to analyze Blowfish. While the sequential analysis, given in terms of time and space requirements, is done in the traditional fashion, the parallel analysis is being done a bit differently.

The parallel analysis is being done using the Work-Time framework. However, this analysis is being dealt with a little differently than in a strict parallel Work-Time framework analysis. In particular, the kinds of work done are being dealt with as discrete units. We are taking some of the kinds of work which are similar to within some scope and are quantizing them into particular units of work much as the epsilon-delta technique of The Calculus does for limits.

Section 3.1.1 The Blowfish Function F

Section 3.1.1.1 The Traditional Sequential Analysis

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 3.1.1.1.1 Time Analysis

Each of the four S-boxes within the Blowfish function F needs to have an entry fetched from it. Let's call the time this takes four data-fetch cycles. The ADD mod 2^{32} instruction is a register-to-register instruction (RR). So is the XOR instruction. These instructions are measured in instruction cycles, or, more precisely, in clock cycles, and are extremely fast to execute. There are few instructions of both types (RR and data-fetch instructions), so it suffices to say that the execution time for the Blowfish function F is linear in the number of clock cycles. More formally, as clock cycles are the least common measure of any kind of instruction cycle, we can say that Blowfish executes in O(n) time where n is clock cycles.

Section 3.1.1.1.2 Space Analysis

There are four S-boxes with 256 entries of length 32 bits each. This requires $4 \times 256 \times 32$ bits to be stored. This can be restated as $2^2 \times 2^8 \times 2^5 = 2^{15}$ bits or 2^{12} bytes or 2^{10} 32-bit words. This is 1K of 32-bit-word memory to store all of the S-boxes. We will also need 18 32-bit words of storage for the 18 P_i keys. We will need one 32-bit word to store the input bits, another 32-bit word for intermediate storage while breaking up the 32-bit input into four 8-bit pieces, and a 32-bit word for the additions mod 2^{32} and the XOR instructions. This gives us a little more than 1K of 32-bit storage for data kept throughout the encryption. Working operational storage is that storage to be used by the pointers and other such data types. This is also small. Essentially, we have o(1K) of 32-bit storage required for the Blowfish function F.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 3.1.1.2 The Work-Time Framework Analysis for the Sequential Algorithm

Section 3.1.1.2.1 Units of Work

Using shift instructions (which can be approximated by RR instructions) we can say that we have four RR units of work to peel off the four 8-bit strings. Using our epsilon-delta analogy, we have quantized the shift and RR instructions together. We have four table lookups to find the proper S-box entries. We have two ADD mod 2^{32} instructions (RR instructions) and one XOR instruction (an RR instruction). In all, we have essentially seven RR instructions and four table-lookup (TLU) instructions. If we classify these units of work by the kind of work done, we can denote the RR units of work by W_1 and the TLU units of work by W_2. This gives us seven units of W_1 work and four units of W_2 work. As the TLU units of work do more work than the RR instructions, we can give a strict upper bound on the measure of work as 11 W_2 units of work or, more formally, $o(W_2)$.

Section 3.1.1.2.2 Units of Time

The Work-Time framework allows us to define an algorithm as a series of timesteps. The work done in each timestep is categorized by a time unit. That is, a sequence of time units , each associated with one or more units of work, defines the algorithm. This is the same thing as defining the algorithm itself in terms of work and time.

For the sequential algorithm, since all of the work for the sequential algorithm is done sequentially, the time units are also strictly sequential. Thus, we have

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

seven units of T_1 time in which seven units of W_1 work are done and four units of T_2 time in which four units of W_2 work is done. We can say that we need O(T_2) units of time to do the o(W_2) work for the sequential algorithm. As we are measuring time in terms of work, we cannot take the liberty of saying that we need only o(T_2) time.

While using the parallel Work-Time framework for the sequential algorithm does not tell us very much in the sequential case, it does allow for some interesting and different analysis in the parallel case.

Section 3.2 The Feistel Network

Section 3.2.1 The Traditional Sequential Analysis

Section 3.2.1.1 Time Analysis

(Note that this analysis is easier to follow while looking at Figure B.1.) We need time to fetch the 18 P_i subkeys. Each fetch takes a data-fetch cycle. The XORs of x_L's with the 16 P_i subkeys, the two XORs with the data and the last two P_i subkeys, and the other 18 XORs in the Feistel network are all RR instructions and execute linearly in clock cycles. Splitting up the data input to the Feistel network and restoring the two halves of ciphertext to exit the Feistel network are also executed as RR instructions and are measured in clock cycles. Taking the first 15 P_i subkeys and swapping them as x_L 's to become x_R 's and the 15 swaps taking x_R 's to become x_L 's takes 30 RR cycles in the Feistel network plus the cost of the 16 executions of the Blowfish function F. It suffices to say

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that the entire encryption portion of the Blowfish algorithm runs linearly in the number of clock cycles.

Section 3.2.1.2 Space Analysis

The 18 P_i subkeys take up 18 32-bit words. They are expected to be in the on-chip cache, reachable by cache table lookup. The storage space taken up by the Blowfish function F must be included. There are only a few more data locations needed for temporary storage. The space needed for the Blowfish function F is several orders of magnitude greater than that used for the rest of the encryption portion of the algorithm; thus it is reasonable to say that the storage requirements for the Feistel network, including the Blowfish function F, is still o(1K) of 32-bit memory.

Section 3.2.2.1 The Work-Time Framework Analysis for the Sequential Algorithm

Section 3.2.2.1.1 Units of Work and Time

As we are still dealing with the sequential algorithm, using the Work-Time framework will mainly change the viewpoint from which we will look at this part of the algorithm. We still have the same two kinds of work to be done, viz., W_1 and W_2, and we still have the same two related units of time in which the work can be done, viz., T_1 and T_2. It is worth noting that since we are using the Work-Time framework we do not have to strictly map the time units to the work units but that the inverse mapping must be maintained.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 3.3 Analysis of the Sequential Encryption Portion of Blowfish as a Whole

Looking at the analysis of the Blowfish function F and the Feistel network we see several things. First and foremost we see that the Blowfish function F dominates the encryption portion of the algorithm. No matter what we do with the Feistel network it will not make as great a difference as making the Blowfish function F more efficient.

We see that we have the option of using parallelism to implement what is an inherently sequential algorithm. Does using parallelism give us any benefit? The answer, which is "yes and no, depending..." comprises the rest of this dissertation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

A PARALLELIZED VERSION OF BLOWFISH

Section 4.1 Parallel Analysis of the Original Blowfish Algorithm

The following units of work need to be accomplished. Assuming that we have sufficient processors to issue each task its own processor, the following table gives a parallel algorithm to execute the encryption portion of the original Blowfish Encryption Algorithm. Note that we are using the parallel Work-Time framework, so the steps are parallel units of time and that the work accomplished is done as a function of these units of parallel time.

The Feistel Network:

Parallel Step (Time)	Action(s) (Work)
1	-- Take 64-bit plaintext into the Feistel network

Table 4.1 (1 of 2)

A Parallel Feistel Network For The Blowfish Encryption Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2	-- Split the plaintext into two strings and send them as x_L and x_R
	-- fetch all P_i subkeys
3	-- XOR P_1 with x_L (after the first execution, P_1 becomes P_i)
4	-- Take the output of step 3 and pass it to the Blowfish function F
	-- Take this same output and take it to be x_R in the next round
5 - 9	-- Wait for the Blowfish function F to finish its 5 units of parallel time and associated work (see Table 4.2)
10	-- Take the data returned from the Blowfish function F back into the Feistel network
11	-- This output from the Feistel network is XORed with the waiting x_R in the current round
12	-- The x_R in step 11 becomes x_L in the next round (This completes a full round.)
13 - 152	-- Repeat steps 3 - 12 14 times (14 complete rounds)
153 -161	-- Repeat steps 3 - 11 (the last round without the last swap in which x_R becomes x_L)
162	-- XOR the output of step 161 with P_{17} (this came from the last execution of step 11)
	-- XOR the output of the XORing with P_{16} with P_{18} (This came from the last execution of step 12)
163	-- Recombine x_L and x_R to form a 64-bit block of ciphertext
164	-- Output the ciphertext from the Feistel network

Table 4.1 (2 of 2)

A Parallel Feistel Network For The Blowfish Encryption Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Similarly, a parallel algorithm for the Blowfish function F is given in Table 4.2. This algorithm also assumes that there are enough processors for all necessary tasks. Its five units of parallel time and the associated work are included in the Feistel network portion of the algorithm in Table 4.1 above.

The logic of Section 3.3 still holds for the parallel algorithm given here because the number of processors is sufficient for all of the necessary processing. The case examined in Chapter 3 had only one processor, so each unit of time was determined by the kind of work it had to do. This is identical to the parallel version of Blowfish in terms of the number of steps which have to be performed with only one processor available. As we are using the parallel Work-Time framework, this also requires the units of time for the parallel algorithm with only one processor to be the same.

Parallel Step (Time)	Action(s) (Work)
1	Take the four 8-bit substrings from the 32-bit input
2	Fetch the appropriate S-box entries for all four S-boxes
3	Take the output of the first two S-boxes and add them mod 2^{32}
4	XOR the output of step 3 with the output of the third S-box
5	Add the result of step 4 to the output of the fourth S-box mod 2^{32}

Table 4.2

A Parallel Blowfish Function F

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For the Blowfish function F, the parallel Work-Time framework's analysis works out fine. There are only two kinds of work to be done. Each kind of work can be done in one of the two kinds of time which are so far defined. Certainly, if we use the more general time: $T = \sum_{i=1}^{\max time} T_i$, we will still satisfy the Work-Time framework's requirements. Thus, we can get an upper bound on the measure of parallel time, if not a least upper bound, for the Blowfish function F. The logic in section 3.2.2.1 still works for the parallel case where the number of processors needed suffices.

What happens, then, if there are not sufficient processors such that each task to be done is assured of its own processor? We look at this in a way similar to any other parallel analysis. We look for the case where n tasks and p processors are as follows.

$n < p$

We have more processors than tasks. We don't use the extra processors and the algorithm runs the same as it does with $n = p$.

$n = p$

The number of processors is equal to the number of tasks. This is the case we have shown.

$n > p$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We have to partition the tasks in such a way that we maximize or minimize whatever we are concerned with - usually time/speedup, throughput, optimal usage of resources (especially processors), and so forth. In the case of this algorithm, if we have fewer processors than we have processes we cut back on the parallelism and the parallel execution approaches the sequential execution as we do so. This is not what usually happens. $T_1(n)$ does not usually approach $T^*(n)$ as closely as it does in this algorithm. It is probable that because the algorithm we are parallelizing is essentially a highly sequential algorithm that this is so.

To compare the parallel algorithm presented with the sequential one, we give a detailed sequential algorithm below. We are using the parallel Work-Time framework for the parallel case, so T(n) =f(n), and f(n) is a measure of work done within a parallel unit of time. For the sequentiai version we are using the standard sequential notion of W(n) = f(T). These sequential versions of Tables 4.1 and 4.2 are Tables 4.3 and 4.4 which follow.

For the Feistel network:

Step/Unit of Work	Action(s) (Time)
1	Take the 64-bit block of input data into the Feistel network

Table 4.3 (1 of 3)

Sequential Execution of the Feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2	Take the left 32-bits as x_L
3	Take the right 32-bits of the 64-bit input as x_R and pass the data to be XORed in the current round
4	Fetch P_1 (later P_i)
5	XOR P_i with x_L
6	Input this result (step 5) to the Blowfish function F
7 - 17	The Blowfish function F is running using 11 units of work
18	Take the output from the Blowfish function F (from step 17) and XOR it with x_R in the current round
19	Take the output of the last XOR with P_i and x_L and have it become x_R in the next round (x_L --> x_R) (from step 5)
20	Take the output of the XOR with the last x_R and bring it left to be x_L in the next round (x_R --> x_L) (from step 18)
21 - 258	Repeat steps 4 - 20 14 times
259 - 274	Repeat steps 4 - 19
275	Fetch P_{18}
276	XOR x_L with P_{18} (x_L is from the last step 20)
277	Take the output of the Blowfish function F and XOR it with the last x_R (from the last step 19)
278	Fetch P_{17}

Table 4.3 (2 of 3)

Sequential Execution of the Feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

279	Take the output of step 277 and XOR it with P_{17}
280	Take x_L to become part of the output (from step 276)
281	Take x_R to become the rest of the output (from step 279) and put the output 64-bit block together
282	Output the 64-bit block of ciphertext and exit the Feistel network.

Table 4.3 (3 of 3)

Sequential Execution of the Feistel Network

For the Blowfish Function F we have:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step/Unit of Work	Action(s) (Time)
1	Take the first 8-bit string from the 32-bit input string
2	Using the value from step 1, fetch the proper S-box entry from the first S-box
3	Take the second 8-bit string from the 32-bit input string
4	Using the value from step 3, fetch the proper S-box entry from the second S-box
5	Add the output from steps 2 and 4 mod 2^{32}
6	Take the third 8-bit string from the 32-bit input string
7	Using the value from step 6, fetch the proper S-box entry from the third S-box
8	XOR the output of steps 5 and 7
9	Take the fourth 8-bit string from the 32-bit input string
10	Using the value from step 9, fetch the proper S-box entry from the fourth S-box
11	Add the results of steps 8 and 10 mod 2^{32}

Table 4.4

Sequential Blowfish Function F

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5

THE IMPLEMENTATION-INDEPENDENT BLOWFISH ALGORITHM

Section 5.1 Identifying the dependencies

Schneier's presentation of his Blowfish Encryption Algorithm is highly implementation-dependent. In particular, his algorithm is finely tuned for 32-bit systems with enough cache to hold all the necessary keys, both the P_i sub-keys and the four S-boxes [Schneier94b]. Note that for this class of computer chips, in order to maintain the integrity of the data in the cache, there can be no other programs running at the same time on the same machine while the algorithm is executing. If other processes were permitted to execute along with the algorithm, the keys stored in the cache could become corrupted. This says that the Blowfish Encryption Algorithm must run as a dedicated system.

The most obvious dependency is the 32-bit system implementation. If the word size in the algorithm were to be changed, the entire algorithm would execute very differently. For instance, if the word size (and CPU) were 16 bits, with the algorithm as described but with any mention of 32-bit systems replaced by equivalent 16-bit system operations, the algorithm would still work but it would run more slowly. If we were to be able to use a 64-bit system, such as one based

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on the Alpha chip, the same kinds of changes could be made and the algorithm would run faster. Clearly, the word size changes the sequential performance of the algorithm.

What if the algorithm were to be implemented on a system which has an unusual word size, such as the DEC-20? Or, what if the algorithm were to be generalized to the point where the implementation can be done using any number-basis we might choose? It need not even be a value which could be implemented on a digital computer, e.g., a system implemented to the base e. This would destroy the usefulness of the algorithm. So, let us make the *assumption* that we will be executing some implementation of the algorithm on a computer.

Is this sufficient? No. There is the assumption that this algorithm will run on a binary digital computer. While this is probably going to be so, it is still an *assumption.* This is a convenient assumption as it allows us to keep the base-2 portion of the algorithm. We will retain the base-2 assumption mainly because it is convenient to do so, but we must keep in mind that it is only an assumption. For example, such instructions as the addition mod 2^{32} depend on having an implementation on a binary system.

The other major *assumption* has to do with the Blowfish function F. We need to be able to break the word-sized input to this function into parts we can deal with. In Schneier's algorithm, we have 8-bit pieces of the 32-bit input to the Blowfish function F. As we have a 32-bit system, the choice of 8-bit pieces allows for symmetry: we can do the same thing when we input one 8-bit string to each S-box for all four S-boxes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The number and sizes of the S-boxes could be differently defined, satisfying the cryptographic requirements but possibly making for an awkward implementation. For example, Schneier's 32-bit system uses 8-bit input strings because it is convenient [Schneier94c]. To do this his S-boxes have to hold 2^8 or 256 entries. He needs 2^8 entries because he needs all possible permutations of the 8-bit seed to generate the necessary permutations for each of his S-boxes. As his S-boxes have 8-bit input and these 8 bits are a breakdown of a 32-bit number, he needs to have four S-boxes to operate on all of his 32-bit input.

What if he had decided to use 4-bit input? He would then have had to use 2^4 or 16 entries (all permutations of a 4-bit seed for each S-box) in eight S-boxes in order to keep his symmetric implementation. This would need eight S-boxes each with 16 entries. Note that these constructs and number of table entries differ yet both are cryptographically correct [Schneier94c]. Needless to say, a change in word size would affect the S-boxes, but just using a different scheme to implement the S-boxes will give different constructs. Any change in the number of bits going into an S-box which does so in an asymmetric fashion would drastically change the algorithm, so we will retain the symmetry but not necessarily the same breakdown of the original data input to the S-boxes.

Any implementation which uses both a symmetric and 2^n-based scheme is also still able to take advantage of shift operations.

Section 5.2 The Implementation-Independent Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 5.2.1 Removing the Dependencies

We have assumed that this algorithm will be implemented on a binary digital computer. We have assumed that instructions will operate as the size of a single word on that computer. We have assumed that we will break up the input into the S-boxes of the Blowfish function F in some symmetric pattern. We have also assumed the framework of the Feistel network and the general idea of the Blowfish function F in order to keep as close to the algorithm's intent as possible.

We need to redefine some of these dependencies. For instance, we can substitute a variable "wordsize" for the places where the algorithm uses 32-bit dependencies. We can still retain the parts of the algorithm which act upon one word, such as the XORs and the addition mod $2^{wordsize}$. The assumption based on the 32-bit system which gives us four 8-bit by 32-bit S-boxes with 256 entries can be changed to any convenient number of S-boxes with symmetric input-size and wordsize entries so long as the cryptographic requirements for the S-boxes are maintained. This latter includes the number of entries in an S-box, the size of the input to an S-box, and the contents of each S-box itself.

Section 5.2.2 The Implementation-Independent Algorithm

The Feistel network portion of the algorithm is implementation-dependent in three areas. The first is the number of rounds in the Feistel network; the second is the addition of the extra keys P_{17} and P_{18}; the third is the 64-bit block of data input to the Feistel network as it is a twice-wordsize sized block.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The reversal of the swap of x_L and x_R at the end of the 16th round of the Feistel network and the addition of the two extra subkeys allows the Feistel network to be symmetric so that encryption and decryption can use the same structure. If these keys are removed then the encryption will run slightly faster but we will not be able to decrypt using the same structure.

Adding or removing rounds in the Feistel network will impact the algorithm and its analysis primarily by adding or removing executions of the Blowfish function F. As the Blowfish function F is the dominating factor in the performance of the Blowfish Encryption Algorithm, we will concentrate on it and leave the Feistel network alone. We must remember that in leaving the Feistel network unchanged we are accepting the *assumption* of its structure.

However, there are several changes in the Blowfish function F. First, we do not know the number of S-boxes we need because we do not know the number of bits in the input to an S-box. We do not know the number of entries in an S-box because we do not know either the size of the input to an S-box or the partitioning of the data to be input to the S-boxes. It is only because we have assumed that we will keep the property of symmetry that we know that the entries to each S-box will have the same number of bits.

The Blowfish function F now looks something like the following. Note that the Greek lower-case letters indicate the number of the S-box and the lower case Roman alphabet letters denote the location of the entry within the S-box.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$F(x_L) = \quad ...((S_{\alpha,a} + S_{\beta,b} \text{ mod } 2^{wordsize})$

$\text{XOR } S_{\gamma,c}) \text{ ...}$

alternating addition mod $2^{wordsize}$ followed by XORing that output and the next S-box's result until all entries have been used in the computation.

Figure 5.1

Pseudocode For the

Implementation-Independent Blowfish Function F

In Chapter 6 we will analyze this implementation-independent Blowfish function F, tying it in with the Feistel network, and thus look at the entire algorithm in an implementation-independent fashion.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 6

ALGORITHMIC ANALYSIS OF THE IMPLEMENTATION-INDEPENDENT BLOWFISH ALGORITHM

Section 6.1 Sequential Implementation-Independent Blowfish Function F

Looking at Figure B.3 we can see that in the sequential version of the Blowfish function F some things must be done in a particular sequence. Some of these functions can be performed interchangeably. The functions to be executed comprise Table 6.1. The logical actions which may be performed interchangeably are listed after the table itself.

From Chapter 5 we have the pseudocode for the implementation-independent Blowfish function F as Figure 5.1. It is repeated here.

$$F(x_L) = ((S_{\alpha,a} + S_{\beta,b} \text{ mod } 2^{wordsize}) \text{ XOR } S_{\gamma,c}) \ldots$$

alternating addition mod $2^{wordsize}$ followed by XORing that output and the next S-box's result until all entries have been used in the computation.

Figure 5.1

Pseudocode For the Implementation-Independent Blowfish Function F

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Implementing this sequentially we have the following tasks or pieces of work to do and we can use the following order of execution. Note that all of the table-lookups are considered to be equivalent tasks and require the same kind and amount of work and the same kind and amount of time to execute. The additions mod $2^{wordsize}$, under the assumptions made in Chapter 3 and with the additional assumption that all shift instructions take the same time for any small number of places to be shifted, are considered to do the same amount and kind of work and take the same amount of time to execute.

We will use W_i for the different kinds of tasks and T_i for the different units of time taken to do each corresponding task.

Step #	Action / Kind of Work (W_i)	Time (T_i) for This Kind of Task
1	Take off the first substring (W_1)	T_1
2	Take off second substring (W_1)	T_1
3	Fetch the first S-box element (W_2)	T_2
4	Fetch the second S-box element (W_2)	T_2
5	Add these entries mod $2^{wordsize}$ (W_1)	T_1
6	Take off the next substring (W_1)	T_1
7	Fetch the appropriate S-box element (W_2)	T_2

Table 6.1 (1 of 2)

Sequential Implementation-Independent Blowfish Function F

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8	XOR the results of the most recent add mod $2^{wordsize}$ (step 5 the first time) with the result of step 7 (W_1)	T_1
9	Repeat steps 6, 7, and 8 for each "add mod $2^{wordsize}$ " loop [6 (W_1) ,7 (W_2) and 8 (W_1)]	zero or more ($T_1 + T_2 + T_1$)
10	Repeat steps 6, 7, and 8 for each "XOR" loop [6 (W_1), 7 (W_2), and 8 (W_1)]	zero or more ($T_1 + T_2 + T_1$)
last	Repeat steps 9 ($W_1 + W_2 + W_1$) and 10 ($W_1 + W_2 + W_1$) as appropriate (alternating add mod $2^{wordsize}$ loops and XOR loops	zero or more ($T_1 + T_2 + T_1$) and zero or more ($T_1 + T_2 + T_1$)

Table 6.1 (2 of 2)

Sequential Implementation-Independent Blowfish Function F

Within Table 6.1 the following steps may occur in either order:

-- steps 1 and 2

-- steps 3 and 4

or

-- steps 1 and 3 together and steps 2 and 4 together

The rest of the steps must appear in the given sequence.

Note that this is not the usual sequential analysis of an algorithm. This is deliberate. The intent is to look at the different kinds of work which must be done as well as to classify these kinds of work by type. The type of work is meant to represent almost identical amounts of work. By "almost identical" we mean

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

something similar to the epsilon-delta method of The Calculus. That is, given a detailed description of some unit of work (the epsilon), these various units of work can be defined to fit into some discrete class of work , W_i (the delta). Similarly, units of work have corresponding units of time T_i. If two units of work do not have the same units of time then the units of work are considered to differ. The notation W_i and T_i -- as separate terms -- is from the parallel analysis of algorithms and will be looked at when we look at the parallel analysis of this algorithm.

The idea is to be able to classify a unit of work with its corresponding unit of time so that the unit of work can be dealt with instead of the trappings of some particular, possibly assumed, implementation of an algorithm. It is hoped that by concentrating on the kinds of work to be done it may be easier both to analyze an algorithm as well as to find an appropriate implementation once more details enter the picture.

For the sequential implementation-independent Blowfish function F we can now say that this algorithm runs in 5 W_1 + 3 W_2 units of work for three S-boxes. As we add S-boxes and input strings to them, we add ($2W_1 + W_2$) units of work and (2 $T_1 + T_2$) units of time associated with these units of work.

We now have additional information which we are able to use for any number of symmetric S-boxes with any fixed length input string going either to the Blowfish function F or to an S-box. This extra information allows us to know the extra work and time involved in changing the basic implementation with some degree of reliability.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 6.2 Parallel Blowfish Function F

Here we are assuming that we have enough processors (or the capability to multitask without degradation) as we require. We use T_i for the units of parallel time and W_i for the different kinds of units of work to be done in each parallel time-step.

For the parallel implementation-independent Blowfish function F, shown in Table 6.2 below, we have some differences from the sequential version. There are two major differences which matter. The first major change is that we will now go from looking at time associated with units of work to the parallel Work-Time framework perspective. This means that each unit of parallel time, T_i, is the unit of measure; the amount and kind of work which is done in a T_i unit of time is secondary. Secondly, we are no longer constrained to having the input to each S-box the same length for all S-box inputs. The parallel step in which we remove all the substrings of the input to the Blowfish function F need not necessarily correspond to the same kind or number of units of work for each substring. If we wish to use a single unit of work to classify the type of work done, we must choose a new variable, W_3, to represent this new class of work. This new unit of work, W_3, will require a new corresponding unit of time, T_3, when we later generalize the algorithm. It is worth noting that the associated units of work and time, (W_3, T_3), might reduce to (W_1, T_1) or (W_2, T_2), but until the specifics are known, we must assume that (W_3, T_3) comprises a new unit of work and time. In the following tables we will also keep the various T_i time units as associated with their W_i units of work since we will want to look at them again when we are working with the generalized form of the algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In Table 6.2 below, each "step #" is a Work-Time framework unit of parallel time. The entries in the rightmost column are for later reference. Similarly, the differing W_i units of work are only needed to determine the "greatest" amount of work which will be done in each step. (The greatest subscript indicates the greatest amount of work.)

Step #	Action / Kind of Work (W_i)	Time Unit, T_i for This Kind of Task
1	Take off all substrings from the input data (W_3)	T_3
2	Fetch all S-box entries (W_2)	T_2
3	Add the two entries mod $2^{wordsize}$ (W_1)	T_1
4	XOR this result with the output of the next S-box (W_1)	T_1
Do steps 3 and 4 as necessary	W_1	T_1

Table 6.2

Parallel Implementation-Independent Blowfish Function F

If we modify the above table by fetching the S-boxes so that we may always act upon two pieces of data at a time, we come up with the following table. Table 6.3 gives a generalized implementation-independent Blowfish function F. The information within Table 6.3 assumes a minimum of three S-boxes. As this table

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

will become the generalized form of the algorithm, the W_i's and T_i's have been left in.

Step #	Action / Kind of Work (W_i)	Time Unit, T_i for This Kind of Task
1	Take off all substrings from the input data (W_3)	T_3
2	Fetch two (or more) S-box entries (W_2)	T_2
3	Add (the) two entries mod $2^{wordsize}$ (W_1)	T_1
4	XOR this result with the output of the next S-box (W_1) (A third S-box must have been fetched in order to get this far)	T_1
Do steps 3 and 4 as necessary	W_1 and W_2 per additional S-box and operation on an S-box. If we get all the S-boxes in parallel, we lose our (W_2, T_2) cost for this step.	T_1 and T_2 as appropriate to the W_1 and W_2 costs

Table 6.3

General Implementation-Independent Blowfish Function F

The parallel algorithm for the implementation-independent version of the Blowfish function F, now generalized, subsumes the earlier ones. We can use this parallel algorithm to describe a general implementation-independent

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithm for the Blowfish function F. This now includes removing the decision of whether to implement the algorithm sequentially or in parallel. Thus, Table 4.3 has now become the general form of the implementation-independent Blowfish function F.

Section 6.3 Sequential Implementation-Independent Feistel Network

In Chapter 5 we mentioned that the only real dependencies in the Feistel network portion of the Blowfish Encryption Algorithm have to do with the number of rounds, the two extra keys, P_{17} and P_{18}, and the 64-bit input. This latter is the easiest to generalize. We term it the "twice-wordsize" length of the input block of plaintext. As we want to keep the structure of the Feistel network yet remove some of the other implementation-dependent portions of it, let us just look at the number of rounds within the Feistel network.

Let us retain the two extra subkeys as they are convenient and do not really change the algorithmic analysis of this part of Blowfish. Instead, let us choose some small number of rounds, say four rounds, to comprise the basic Feistel network and see what we get as a generalized basic algorithm while allowing the number of rounds to vary. While we have chosen four as a basic number of rounds in the Feistel network, the number of rounds in a Feistel network is primarily subject to cryptographic requirements, not the assumptions made for the purposes of this algorithmic analysis.

Following is Table 6.4 giving the steps to be executed to logically implement this partially implementation-independent Feistel network in Blowfish. As in Chapter

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4, the Blowfish function F has its input passed to it as part of the Feistel network's logic. The logic to receive the output from the Blowfish function F is similarly reflected in the Feistel network's operations. Within this table, the following operations must be performed. Some must be in sequence; some can be done in a less strict fashion.

Referring to Figure B.1 will make it easier to follow this description. Note that we are using the general implementation-independent version of the Blowfish function F from Table 6.3. As a result of this, we no longer have knowledge of the amount of work to be done or the time needed to do this work while the Blowfish function F is executing. From within the Feistel network we know at what point the Blowfish function F is called and when it returns a value. We have a rough estimate of (4 + n) units of some kind of work and time. We get this by looking at Table 6.3 in which we have four steps to execute before we reach work which takes either one unit of W_1 or one unit of $(W_1 + W_2)$ to execute (for step 4). Theoretically, there is no reason why n has to have any particular value, other than it has to be an integer.

Step #	Kind of Task / Unit of Work W_i. The corresponding T_i is assumed
1	Twice-wordsize bits of data enter the Feistel network (W_1, T_1)
2	Wordsize bits go to the left to be x_L in the current round (W_1, T_1)

Table 6.4 (1 of 4)

Sequential Implementation-Independent Feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3	Wordsize bits go to the right (x_R), This becomes x_R in the current round (W_1, T_1)
4	Fetch P_1 (later P_2 through P_4) (W_2, T_2)
5	x_L is XORed with P_1. The result of this step becomes x_R in the next round (W_1, T_1)
6	The result of step 5 is input to the general implementation-independent Blowfish function F, This will take (4 + n) units of ((W_1, T_1) or ((W_1, T_1) and (W_2, T_2)) as appropriate). Looking back to Table 6.3, the 4 is from the first 4 steps and the n is from the choice of steps listed in the last step. If the S-boxes are fetched in parallel, we lose the step which fetches the next S-box and its associated overhead, (W_2, T_2). The first of the four steps needs (W_3 , T_3); the rest are as listed above.
after the 4 + n steps	...
a1	The output from the execution of the Blowfish function F (step 6) is XORed with x_R in the current round. This result becomes x_L for the next round (x_R ---> x_L) (W_1, T_1)
a2	Take the output of the last XOR with P_i and x_L (from step 5) and have it become x_R in the next round (x_L ---> x_R) (at most: (W_1, T_1))

Table 6.4 (2 of 4)

Sequential Implementation-Independent Feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a3	Take the output of the XOR with the last x_R (from step a1) and bring it left to be x_L in the next round (x_R ---> x_L) (at most: (W_1, T_1))
a4	Steps 4 - through a3 are repeated for each full round until the last one. In the last round, we skip step a3 which swaps x_L and x_R (as we will otherwise have to swap them back again) in the last execution. Extra rounds which are to be added must be added in the logic before the beginning of the fourth round (only full rounds are permitted to be added). The overhead for adding a new round is: ([(2 - 4) (W_1, T_1)] + 1 (W_2, T_2) + the (4 + n) steps for each execution of the Blowfish function F)
a5	Repeat steps 4 - a1 (2 (W_1, T_1) + 1 (W_2, T_2)) (the last round)
a6	Fetch P_{last} (P_{last} is equivalent to the P_{18} of the original algorithm) (W_2, T_2)
a7	XOR x_L with P_{last} (x_L is from the last step a3) (W_1, T_1)
a8	Take the output of the Blowfish function F and XOR it with the last x_R (from step 6) (W_1, T_1)
a9	Fetch $P_{next-to-last}$ ($P_{next-to-last}$ is equivalent to P_{17} in the original algorithm) (W_2, T_2)
a10	Take the output of step a8 and XOR it with $P_{next-to-last}$ (W_1, T_1)
a11	Take x_L to become the left wordsize part of the twice-wordsize output (from step a7) (W_1, T_1)

Table 6.4 (3 of 4)

Sequential Implementation-Independent Feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a12	Take x_R to become the wordsize right part of the twice-wordsize output (from step a8) and put the output 64-bit block together (W_1, T_1)
a13	Output the twice-wordsize-bit block of ciphertext and exit the Feistel network. (W_1, T_1)

Table 6.4 (4 of 4)

Sequential Implementation-Independent Feistel Network

Note that steps a2 and a3 in Table 6.4 are listed as "(at most: (W_1 and T_1))". The "at most" is there to keep things generalized and implementation-independent. If this is being executed on a computer, the passings-on of these variables will fall out of the computations which instantiate them. However, if we have a chain of baboons [idea from Tanenbaum87, p. 108], for instance, performing this work, then these steps do have both work and time associated with them. This same notation is also used later on.

We now have a good estimate in terms of units of work and time for the sequential execution of the Feistel network using the general implementation-independent Blowfish function F. From Table 6.4 we get the following measurements:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Number of Units/ Overhead	What Is Being Measured
19	Units of (W_1, T_1)
5	Units of (W_2, T_2)
4 + n	The overhead for each execution of the Blowfish function F. This does not include step a1 from Table 6.4 as this overhead, in terms of steps, is unknown. Therefore, we are dealing only in terms of units of (W_1, T_1) and units of (W_2, T_2) and units of (W_3, T_3)
Overhead of the sequential Feistel network with 4 rounds	The step above this one ⇑
16 (W_1, T_1) + 4(W_2, T_2)	The overhead for each additional round -- when added in a sequential implementation-independent Feistel network from steps 4 - a3 in Table 6.4. We are discounting steps of (W_3 and T_3) as they will become steps of (W_1, T_1) and of (W_2, T_2) - but we don't know how many.

Table 6.5

Measurements in Terms of Work and Time

Using the Sequential Implementation-Independent

Feistel Network With Four or More Rounds

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 6.4 Parallel Implementation-Independent Feistel Network

We are making the same assumptions as were made in Section 6.2. Again we are using the parallel Work-Time framework, so each unit of parallel time does some unit or units of work. As before, the units of work and time, as broken down in the sequential case, are being kept in the table for later reference. Within the table, the number of each step is the measure of parallel time.

Step # / Parallel Time Unit	Kind of Task / Unit of Work W_i. [The corresponding T_i is assumed]
1	Take the twice-wordsize-bit input and split it into two wordsize-bit strings, x_L and x_R (W_1, T_1)
	Fetch all P_i subkeys -- a minimum of 6 for 4 rounds. (W_2, T_2)
2	XOR x_L with P_1 (W_1, T_1)
	Pass x_R to be XORed with the output of the Blowfish function F in the current round (the same (W_1, T_1))
3	Pass the output of the XOR in step 2 to the first Blowfish function F as input (W_1, T_1)

Table 6.6 (1 of 2)

Parallel Implementation-Independent feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

.	Pass this same information to the right side of the Feistel network to be used as x_R in the next round (the same (W_1,T_1))
4	Execute the Blowfish function F. This takes (4 + n) steps measured in units of (W_1,T_1), (W_2,T_2), and (W_3,T_3) as appropriate. (refer to Table 6.3, look at the unit of time in the third column and go to the first column for the corresponding step #)
5	Pass the output of step 4 to the right to be XORed with the waiting data, x_R (from step 3) in the same round (at most: (W_1,T_1))
6	Take the output of this XOR and bring it left as x_L to be XORed with the next P_i subkey (at most (W_1,T_1))
7	Repeat steps 2 - 6 for each full round to be executed in the Feistel network. This has an overhead of $5(W_1,T_1)$ excluding the execution of the Blowfish function F
------	-------
	(After the last full round is executed)
b1	Execute steps 2 - 5 with overhead of $4(W_1,T_1)$ excluding the execution of the Blowfish function F
b2	Execute the two last XORs (the equivalents of P_{17} and P_{18} in Schneier's original Feistel network) in parallel (W_1,T_1)
b3	Combine the output to get the ciphertext (W_1,T_1)

Table 6.6 (2 of 2)

Parallel Implementation-Independent Feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Unlike the redefined general implementation-independent Blowfish function F which was arrived at easily from Tables 6.1 and 6.2, giving Table 6.3, getting the general form of the implementation-independent Feistel network from Tables 6.4 and 6.6 is not laid out as easily. Yes, the general implementation-independent Feistel network exists, but it exists as equating certain steps from Table 6.4 (sequential) with certain steps from Table 6.6 (parallel) and, when looking at the general implementation-independent Feistel network, taking the steps from the set of equivalent steps. A table of these equivalent steps follows on the next page.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Concept done in these steps	Steps from Table 6.4	Steps from Table 6.6
Get input; split into x_L and x_R	1 - 3	1
Fetch the P_i subkeys	4, a6, a9	1
XOR x_L with P_i	5	2
Pass data to Blowfish function F and to be x_R in the next round	6	3 - 4
Blowfish function F is executing	----	----
XOR x_R with the output of the Blowfish function F	a1	5
x_L ---> x_R	a2 - a3	6
repeat for all full rounds	a4	7
	---	---
do the last round	a5	b1
get the last two P_i subkeys	a6, a9	0 - (no step need be done)
do the last two XORs	a7, a8, a10	b2
Put the output data together and output it	a11, a12, a13	b3

Table 6.7

Obtaining a

General Implementation-Independent Feistel Network

From the Sequential and Parallel Feistel networks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 6.5 Where Do We Go From Here?

We have derived a general implementation-independent form for the Blowfish function F (Table 6.3) and a general implementation-independent form for the Feistel network used in Blowfish (Table 6.7). What might we do with the analysis done so far? In particular, can we use our general forms to allow us to model changes in Blowfish's behavior by either modifying the algorithm or its implementation? I conjecture that with these tools we can do so.

Can we determine what the change in performance of Blowfish would be if we were to change the number of S-boxes? Yes. An example doing just this is in Chapter 8. What if we decide that it is too expensive to dedicate a Pentium-class system to just encryption of one 64-bit block of plaintext at a time? Recall that Blowfish was designed and its implementation done in such a way to be optimal for this class of microcomputers. In order to maintain data cache coherence we would need to have either a dedicated system or a system which will run Blowfish and only other programs which must not use the data cache. Might we want to see if we can get a more economical way to run Blowfish? The use of transputers immediately comes to mind.

In Chapter 7, I will give some background on both the transputer and occam, the transputer's native language. Then, in Chapter 8, I will give two different transputer implementations along with a variation of the 8-in 32-out S-boxes which will become 4-in 32-out S-boxes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 7

OCCAM AND THE TRANSPUTER

Section 7.1 The Transputer

The transputer is a computer on a chip. It was developed by INMOS[1] to implement the computer language occam in hardware as a RISC-type chip [CSA94]. We are concerned with the properties of transputers which will enable us to decide whether or not to use them and, if so, in what configuration.

Transputers come in several models. The ones we are interested in have 32-bit CPUs, have 2K-4K bytes of on-chip "fast RAM" [CSA94] , and are individually placed on a single board. While there may be more than one transputer on a physical board (giving a bus-linked connection among the transputers), there is not much temporal difference between having all the transputers on a single board or in a four-transputer network. The time taken to pass data across the links between two transputers is negligible as the links are coprocessors on the transputer chip and the transputer's CPU can be busy doing something else while the information is passed along these links.

[1]INMOS is a member of the SGS-Thomson Microelectronics Group

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Transputers can be on a board on which additional memory can reside. The extra on-board memory has been known to be as great as 64 megabytes and could potentially be as large as four gigabytes. However, the usual on-board memory is 2K - 4K for the smaller, inexpensive systems such as CSA's Transputer Education Kit boards. The layout of a CSA T425 board with 1 megabyte of on-board memory and 4K bytes of on-chip RAM appears as Figure C.1

There are transputer boards with 32 megabytes of on-board memory. These are common for the usual large transputers. Because the transputer has a flat address space it can use all of the 2^{32} bits on a chip to generate this large address space.

Transputer boards can have an interface to a host computer or, as add-on boards, need only have power supplied to them by a passive backplane. Transputers which have an active interface to a computer are able to be programmed and can thus use the memory of the host computer. Moreover, if more than one transputer is actively connected to the same host computer, these transputers and their host are capable of interacting to an extent limited only by the hardware.

In and of themselves, transputers do not have shared memory. Direct communication between transputers is via cables connecting the links from one transputer to another. A link is a physical part of the transputer chip which allows for connection from one transputer to another. A link is a physical piece of hardware which allows for bidirectional synchronous communication. Its software

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

counterpart in occam is called a channel. See Figure C.2 for a graphical depiction of this correspondence.

Transputers have either two or four links. The ones of interest to us have four links. Each link is controlled by its own coprocessor, so the four synchronous I/O links can be working in an asynchronous fashion with respect to each other and to the transputer's processor itself.

Instructions which run on the transputer chip itself have a 1-cycle instruction fetch time. In addition, on-chip there is zero-wait instruction execution. Once we go off-chip and are using the memory on the board on which the transputer resides, we jump to a 3-cycle instruction fetch time.

Section 7.2 Occam

Most of the occam language's instruction set is implemented directly by the transputer. While several enhancements have been made to both the transputer chip and the occam language (officially "occam 2" but commonly referred to as just "occam") they have not changed the basic functionality of either the language or the chip. In particular, the ability to write an occam program to execute on a single chip can be very easily changed to have the occam program run on a network of transputers. This feature allows for multiprocessing on a single chip or on a network of transputers with very little in the way of modification of the occam code.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The transputer has two on-chip clocks. One of these is programmable and can be used to implement a scheduler. The capacity for scheduling is on the chip. By default the microcoded scheduler [INMOS89] runs in a round robin schedule. This can be changed programmatically to force one or more processes to have a high priority while all other processes have a low priority. Figure C.3 has a diagram of the logic of a transputer scheduler. This on-chip clock can also be used to change the default time-slice for the low priority processes.

It is not advisable to have more than one high priority process as these processes run until completion or until a communication interrupt occurs. It is the low priority processes which run using the round robin scheduling algorithm, but they run only when the high priority process(es) is not running. It should be noted that what would be called multitasking on other hardware is called multiprocessing on a transputer. This means that when we speak of multiprocessing in the context of transputers we may be talking about one or more than one transputer.

The kind of process which would do well as a single high priority process in a system with other processes which are low priority might be one which controls and coordinates interdependent actions done by the other processes. This might be an intelligent multprocessing coordinator. Note that multiprocessing on a transputer does not need to be coordinated unless the processes require coordination.

All of the above capabilities of occam and transputers allow the design of a transputer system to be initially implemented on a single chip and later be put onto a network of transputers with very little change in the occam code.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 8

EXAMPLES

Section 8.1 A Variant Blowfish Function F

Schneier's algorithm is designed to be optimal for 32-bit systems with significant on-chip cache and symmetric S-boxes; in particular, with four 8-in 32-out S-boxes. Let us look at a variation of the Blowfish function F for the purpose of contrasting the two. (For our purposes, this variant Blowfish function is assumed to be cryptographically correct.) Insofar as the Feistel network is concerned, either of these functions could be used. As the Blowfish function F is the most frequently executed logic in the algorithm, studying its performance serves as a benchmark for the performance of the entire algorithm.

This variant implementation has eight 4-in 32-out S-boxes. Each S-box has 2^4 or 16 entries. There are eight or 2^3 S-boxes. Each entry, as in the original Blowfish function F, is a 32-bit binary string. This gives us 8 or 2^3 S-boxes each with 2^4 entries, each entry being 32 or 2^5 bits. That is, we have $2^3 \times 2^4 \times 2^5 = 2^{12}$ bits of storage required to store all of the S-box data. This is less storage than required for the original algorithm as given in section 3.1.1.1.2. This variant implementation appears graphically as Figure B.4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A pseudocode description of the variant function follows as Figure 8.1.

$$
\begin{aligned}
F(x_L) = (((((((S_{1,a} + S_{2,b} \bmod 2^{32})\\
&\text{XOR } S_{3,c})\\
&+ S_{4,d} \bmod 2^{32})\\
&\text{XOR } S_{5,e})\\
&+ S_{6,f} \bmod 2^{32})\\
&\text{XOR } S_{7,g})\\
&+ S_{8,h} \bmod 2^{32}
\end{aligned}
$$

Figure 8.1

A Variant Blowfish Function F

When we compare this variant algorithm with the original, we see that many of the similarities remain. We still have symmetric S-boxes. The size of the input to each S-box is still a power of two -- which allows us to retain the ability to use shift instructions in the implementation. The instructions needed logically are still XORs, ADD mod 2^{32}, and table lookups. In fact, from the point of view of the Feistel network -- which calls the Blowfish function F and receives a value back from it -- these two versions of the Blowfish function F appear to be the same.

If we look at Schneier's C code [Schneier94c] in Figure B.5 we can see the effects of this change in the function's implementation. The gist of Schneier's C code is that one 32-bit piece of data is input to the Blowfish function F. An 8-bit piece is taken from the right-hand end (the variables which will become the subscripts come in reverse order due to the use of C) and the 32-bit unsigned

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

integer is shifted until the next 8-bit segment is ready to be removed from it. This is done repeatedly until all four 8-bit segments are removed and stored.

In the case of this variant algorithm, there are eight 4-bit variables which must be removed from the 32-bit input. Also, eight shifts are required to position the data properly for removal from the input string. If the same data types are used, extra work must also be done so that each 4-bit variable becomes an 8-bit variable allowing for the table lookup to be done with unsigned short integers containing correct data.

In Schneier's C code we can do our table lookups along with our ADDs mod 2^{32} and our XORs in the same line of code. We need four table lookups and three RR operations. Schneier's code is simple enough that it can be executed in either one slightly complicated C instruction or in three simple ones.

In the variant implementation, we cannot do things this simply. In addition to having to convert our eight variables from 4-bit unsigned integers to 8-bit ones, we now have to do eight table lookups and seven RR instructions: four ADDs mod 2^{32} and three XORs. There is no way in which this can be written in one instruction such that it is not obfuscatory. In order to write maintainable code, several instructions, at minimum, must be used.

Section 8.1.1 Analysis

As we are examining a sequential implementation, we are looking to have the units of work demarcate the units of time. Looking at Table 6.3, we see that we

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

can obtain the number of units of work and the associated units of time for this implementation.

Our W_3 unit of work now measures both the removal of each substring and any repositioning of this data to fit into the proper data type. Depending on how this is done, a unit of W_3 work may be just a unit of W_1 work. Now, we have 8 units of W_3 work to remove and reposition the input strings, 8 units of W_2 work to fetch the eight S-box entries, and 7 units of W_1 work for the 4 ADDs mod 2^{32} and the 3 XORs. The units of W_3 work may be the same kind of work as the W_1 work depending on how it is implemented.

Section 8.2 Parallel Blowfish Implemented on Transputers

The entire Blowfish Encryption Algorithm can be run on a transputer network configured as a tree or on a single transputer. This dissertation deals only with the encryption portion of Blowfish, so only the Blowfish function F and the Feistel network are being shown with transputer implementations. We will address ourselves to the Blowfish function F first.

There are two kinds of transputer implementations which are appropriate to look at with regard to Blowfish. The more obvious one is a 4-transputer network configured as a tree; the less obvious is the multiprocessed single transputer implementation. Because of the capability to schedule each step in a parallel implementation, it makes sense to look at multiprocessing on both configurations of transputers. The capabilities of occam and the transputer allow us to use very similar code for both implementations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Section 8.2.1 Blowfish On a 4-Transputer Network

If we have four or more transputers we can configure a small network as shown below. The transputers in this diagram are labeled solely for convenience in referencing them. Transputer 1 is the coordinating transputer, transputers 2, 3, and 4 are satellite transputers which get their input passed to them from transputer 1. When the Blowfish function F needs an S-box entry, it gets one from one of the four transputers.

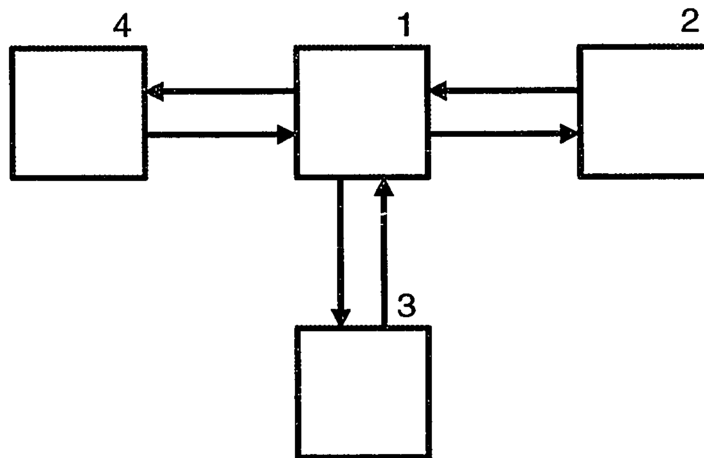


Figure 8.2

A Network of Four Transputers

By assumption, transputers 2, 3, and 4 are connected to transputer 1 by physically close, short cables. Transputer 1 is also connected to a PC host. Figure C.1 shows a transputer board with these same links which is also similar to our transputer 1. Remember that the passing of information across transputer links is implemented by the links' coprocessors, so, as these links are cabled physically close to each other, we are able to ignore communication time

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

between transputers. When we want an S-box entry from each of the satellite transputers, only a small amount of information is being passed across these links. Because of this, we may also ignore data propagation time. (With older transputer chips such as the T414 we must consider propagation recognition time, so let us restrict ourselves to the T425 chip class.)

Referring to Figure 8.2, transputer 1 coordinates the actions among all of the transputers. At the beginning of the execution of the entire Blowfish Encryption Algorithm, when the S-boxes are initially generated, the data making up the contents of each S-box must be passed along to the transputer which will eventually store that particular S-box. This can be done partially in parallel using all four transputers.

Transputer 1 gets the data making up each S-box sequentially from the host PC via its link 0. (The S-boxes could also have been generated on transputer 1.) The coprocessor for each link between two transputers can transfer each S-box's data from transputer 1 to each of transputers 2, 3, or 4 while the rest of the S-box data is being read into transputer 1. The three satellite transputers will each end up with the contents of a different one of the four S-boxes. These S-boxes will each be stored in the on-chip RAM of the particular satellite transputer. The remaining S-box will be stored in the on-board DRAM of the board containing transputer 1. Each S-box will take up 4K bytes of fast RAM memory on each of the satellite transputers. On the board containing transputer 1, the 4K bytes taken up by the S-box will be in the on-board DRAM.

When executing, the body of the Blowfish function F has its code and working data located in the fast RAM of transputer 1. When the Blowfish function F

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

executes it needs an entry from each S-box. The S-box associated with transputer 1 will have its S-box data in the on-board DRAM. The other S-boxes are in the fast RAM of transputers 2, 3, and 4. The request for data from transputers 2, 3, and 4 initiates a synchronous, connection-oriented communication.

Each of the requests for an S-box entry from one of the satellite transputers is done in parallel using almost the same code as that needed to get the S-box entry on transputer 1. The difference in the occam code is in the configuration of the channels. As the requests are able to be implemented in parallel -- one being handled by the CPU of transputer 1 and the other three being handled by the coprocessors controlling the three links to transputers 2, 3, and 4 -- we really do have the ability to get our S-box entries in parallel. There is a very small amount of overhead using these satellite transputers.

The request from transputer 1 to a satellite transputer has negligible overhead. Each request should pass a pointer to the S-box entry on the satellite transputer. The satellite transputer's S-box entries will have a fixed address within the fast RAM, established when the S-boxes were put onto these transputers. Passing a pointer allows each satellite transputers to fetch its S-box entry using a direct address. This preserves the equivalent of cache coherence for the on-chip fast RAM on these transputers.

Fetching an S-box entry from the on-board RAM associated with transputer 1 is not negligible. While the instruction fetch for requesting the S-box entry has a zero wait state and requires only one clock cycle to execute, the data fetch for the S-box entry in the on-board DRAM will take 3 clock cycles. This extra work

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and time will dwarf the overhead incurred in fetching an S-box entry from one of the satellite transputers.

The S-box fetches have been continually associated with the T_2 unit of parallel time and the class of W_2 work. Clearly, this implementation changes the meaning of these units of measure.

Once we are doing the actual encryption of the modified plaintext, the Blowfish function F is called. Transputer 1 takes the 32-bit input and breaks it down into four 8-bit strings of data. Each of these strings is passed to the appropriate transputer in order to get the appropriate S-box entry. We can structure these requests to execute in parallel. The S-box entry for transputer 1 is obtained differently as it is in the on-board DRAM while the other S-boxes are on different transputers but are on-chip for those transputers. The only difference in the occam code for obtaining the data from the S-box on transputer 1 is that the configuration of this channel must be modified.

In occam the equivalent of a transputer link is called a channel. As the transputer links are each controlled by a dedicated coprocessor, while the actual transfer of information from one transputer link to a (particular) second transputer link is synchronous, the four links of a single transputer can be handling the transfer of data in an asynchronous manner.

Once each of the three satellite transputers has both the address of the data it is able to fetch the appropriate entry from its particular S-box. Once it has done this, each of the three satellite transputers will signal transputer 1 letting it know

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that there is input for it. The data is then passed back to transputer 1 in a manner analogous to its receipt.

The program running on transputer 1 will know the order of receipt of the S-box entries. As it gets the data in the order needed by the Blowfish function F (S-box 1 and S-box 2, then S-box 3, then S-box 4) it will do the necessary ADD mod 2^{32} , the XOR, and then the last ADD mod 2^{32} . The result of this last step is then ready to be passed back to the Feistel network. Note that in order to pass this data back to the Feistel network transputer 1 must be both the entry and exit point for the Feistel network.

Analysis

Looking back at Table 6.3, we see that our units of W_3 work remain the same; it may be the case that these units of W_3 work might become units of W_1 work depending on how the implementation is done. However, our four units of W_3 work will be sequential and will be done in four units of T_3 time. Our units of W_2 work, however, will be done in parallel and will require one unit of T_2 parallel time which must be sufficiently long enough to do the "longest" of the W_2 units of parallel work. Our ADDs mod 2^{32} and our XORs require three units of W_1 sequential work to be done in three units of sequential time, T_1. This latter is so as it is more efficient to execute these instructions in the on-chip memory of transputer 1 than it is to do it elsewhere.

Section 8.2.2 Parallel Blowfish on a Single Transputer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As was mentioned in section 8.2.1, occam can be configured to run both using links and using local channels concurrently. Let us assume that the code to use both internal and external channels has been configured and that we are running Blowfish on a single transputer. Occam has the capability to run multiple processes on a single transputer. This is termed multiprocessing rather than multitasking even though only one processor is being used.

The main difference between this implementation and the 4-transputer network is that all of the S-boxes are on the on-board DRAM. Figure C.1 shows this kind of transputer board layout. Note that the only link in use is the one between the host PC and the transputer, link 0.

We will assume that the S-boxes are initialized in a manner similar to that described in section 8.2.1. One S-box is input to the transputer at a time and only one S-box can be stored at a time. Instructions which run on the transputer chip itself have a 1-cycle instruction fetch time. In addition, on-chip there is zero-wait instruction execution. Any operations on the transputer's on-board memory take about three times as long to do as on the transputer itself.

The code to implement Blowfish runs on the transputer itself. Working data and instructions are stored in the on-chip fast RAM. The multiprocessing done in the Blowfish function F could be defined to run in parallel (using the appropriate occam constructs) or it could be scheduled in detail using the on-chip scheduler. As scheduling a parallel implementation is a method used in the parallel design of algorithms, this latter looks to be appealing.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The on-chip scheduler is actually two lists of scheduled processes. One list is for high priority processes and the other is for low priority processes. The default scheduling algorithm is round-robin for the low level processes. High priority processes will run to completion unless a communication interrupt occurs. While the communication interrupt is being serviced a low priority process may run or continue to run in its round-robin timeslice until the high priority process resumes. While more than one high priority process may be scheduled, this is not a good idea as high priority processes run to completion before allowing low priority process to execute. There are two on-chip clock registers which may be used to alter the timeslice for the round-robin scheduler.

While at first glance it appears that by using the schedulers to overlap portions of the execution of the Blowfish function F (i.e., pipelining) is the more efficient method, looking at it more carefully we find that this is not so. As all of the S-boxes are located in the on-board DRAM of the transputer board, no interruptions to the high priority scheduler will ever occur. This is due to the fact that "communication" comes via the transputer's links, and nothing but the link to the host computer (link 0) can generate a communications interrupt while this code is running. Thus, whether you schedule the instructions in Blowfish to run sequentially or in parallel, the only necessary concern is to choose the code which will take up the least amount of on-chip space (lest we need more than 4K bytes of on-chip fast RAM). So, we will end up running sequentially no matter how we design the code.

Analysis

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Looking back at Table 6.3 we see that since we have a sequential implementation we must do our units of W_i work sequentially, doing our work in the appropriate T_i time. We have four units of W_3 work which might become units of W_1 work depending on how the implementation is done. We have four units of W_2 work to get the S-box entries. Finally, we need three units of W_1 work to execute the two ADDs mod 2^{32} and the XORs.

Section 8.2.3 The Feistel Network Implemented on Transputers

For the four-transputer network in Section 8.2.1 we can utilize some of the parallelism of the network to implement the Feistel network in Blowfish. It will not do much for our performance, however, as the Blowfish function F dominates the algorithm's performance.

For the Feistel network, we can store the 18 P_i subkeys on transputer 1. This will lead to sequential fetching of the P_i subkeys when we actually need them. It is not a good idea to store anything other than an S-box on a satellite transputer as the same problem described in section 8.2.1 (akin to violating cache coherence) will result.

Similarly, the swapping of x_L and x_R, done in parallel allows these swaps, in both directions, to be done transparently. Depending on how the coding for Blowfish is done, this may or may not buy us anything. Either assignment statements will take care of these swaps or we will get 15 x_L to x_R swaps and 15 x_R to x_L swaps for one fourth of the sequential overhead. As neither the keys nor the instructions can be put on a satellite transputer without corrupting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the on-chip fast RAM's contents, it makes no sense to consider putting either this code or its data anywhere else but on transputer 1.

The only other realizable parallelism which we can get from the network configuration implementation is to split the input in parallel and to merge the output in parallel. This will reduce these two set of steps to two single steps. We can also do the last two XORs in parallel. It will take as much time to split and rejoin this data than we can get from what the parallelization might give us. This can be done easily on transputer 1.

In essence, we can get only a small amount of savings by using parallelism for the Feistel network using the 4-transputer network as described above.

What happens in the case of the single transputer implementation of Blowfish? In this case, the storage requirements dominate our concern. As we want as much of our code and frequently used data to be in the on-chip fast RAM, the code for the Feistel network should be in the on-chip fast RAM. All four of the S-boxes will need to be stored in the on-board DRAM. If there is still enough room for the code and the working storage requirements, we can consider putting the 18 P_i subkeys on-chip. If we run out of room in the on-chip fast RAM, all other storage will need to be put in the on-board DRAM.

In this case, we get no improvement over the sequential implementation of Blowfish's Feistel network. In fact, every instruction which must be done involving the on-board memory will again take approximately three times as long as the on-chip instructions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If the entire Blowfish Encryption Algorithm, not just the encryption portion of it, were to be run on either a transputer network such as described above, or on a single transputer, again described as above, the code and intermediate data storage are of greater concern than S-box storage. The size of the code and intermediate storage will determine what stays in the on-chip fast RAM and what goes either in one or more satellite transputers (in the network case) or in the on-board memory (in the single transputer case).

Section 8.3 What Have We Learned From These Examples?

These examples have illustrated a change in the algorithm and two different changes in its implementation. In each case we have been able to look back to Table 6.3, the "Generalized Implementation-Independent Blowfish Function F," to determine most or all of the effects of each change. I conjecture that using Table 6.3 we can determine most or all of the effects of any kind of change to Blowfish.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 9

OBSERVATIONS AND CONCLUSIONS OF THE ANALYSIS

Section 9.1 Results

We came up with a general form of both algorithms comprising the encryption portion of the Blowfish Encryption Algorithm: the Blowfish function F and the Feistel network. To do this, we started from the original algorithms for these substructures and generalized as we went. Eventually, this led to a general implementation-independent version of each algorithm. In going from the specifics to the ultimate general form, we used the Work-Time framework of parallel analysis. We did this to give us the greatest flexibility in generalizing the algorithms. The sequential and parallel analyses of both the Blowfish function F and the Feistel network fell out as a byproduct.

The purpose of generalizing the algorithms is so that we can use them as templates. In the case of Blowfish, where the measures are in terms of clock cycles and about 5K of memory, this doesn't give us too much more useful information. But, suppose that we are dealing with costing the construction of a shopping mall. In this kind of situation, where the expenditures are substantial, if we could find a generalized form for all the kinds of work being done and could express it in an algorithmic fashion, we should be able to come up with an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implementation-independent model for the execution of the job. This model could then be used to estimate the effects of changing either the way the work is done or kind of work to be done.

Currently, something such as linear programming could give us this kind of result. However, in setting up a set of linear equations we need to know not only what the variables are but also what their relative weight is. If we could work from the opposite direction, we can then limit our decisions based on introducing the dependencies.

Section 9.2 Limitations and Questions for Further Study

While we were able to come up with a general algorithmic form working from a specific instantiation of our problem, how much of this is related to the particular algorithms? Even if this kind of modeling can be done for other algorithms, is there some systematic way in which we might deduce the general case from the specific? This must be left for future research.

Section 9.3 Summary

We have shown an instance where we can derive a useful generalization from a specific instance of an algorithm. We traced the encryption portion of Bruce Schneier's Blowfish Encryption Algorithm analyzing both the Feistel network and the Blowfish function F. We started with his sequential algorithms, parallelized them, and then removed the implementation-dependent portions of these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithms, substituting more general algorithms. We finished with one generalized algorithm covering both the Blowfish function F and its surrounding Feistel network. We derived what is essentially a template or model which can be used to analyze any changes which might be made to these algorithms. In the case of the Blowfish Encryption Algorithm, the changes are limited to only a few kinds and the effects of changing any part of the algorithm are either obvious or negligible. However, there are other instances where this ability to model in this fashion could make major differences in how and why something with significant expenses is analyzed and implemented.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A

The Original Blowfish Algorithm 80

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The Blowfish Encryption Algorithm[2]

A fast, new algorithm for 32-bit CPUs

Bruce Schneier

Blowfish is a block-encryption algorithm designed to be fast (it encrypts data on large 32-bit microprocessors at a rate of 26 clock cycles per byte), compact (it can run in less than 5K of memory), simple (the only operations it uses are addition, XOR, and table lookup on 32-bit operands), secure (Blowfish's key length is variable and can be as long as 448 bits), and robust (unlike DES, Blowfish's security is not diminished by simple programming errors.)

The Blowfish block-cipher algorithm, which encrypts data one 64-bit block at a time, is divided into key-expansion and a data-encryption parts. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes. Data encryption consists of a simple function iterated 16 times. Each iteration, called a "round," consists of a key-dependent permutation and a key- and data-dependent substitution.

---

[2]This article was originally published in *Dr. Dobb's Journal*, April, 1994. There are slight modifications in this copy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Subkeys

Blowfish uses a large number of subkeys that must be precomputed before any data encryption or decryption. The P-array consists of 18 32-bit subkeys, $P_1, P_2, \ldots P_{18}$, and there are four 32-bit S-boxes with 256 entries each: $S_{1,0}, S_{1,1}, \ldots, S_{1,255}$; $S_{2,0}, S_{2,1}, \ldots, S_{2,255}$; $S_{3,0}, S_{3,1}, \ldots, S_{3,255}$; $S_{4,0}, S_{4,1}, \ldots, S_{4,255}$.

Encryption

Blowfish is a Feistel network consisting of 16 rounds; see Figure 1 in Appendix B. The input is a 64-bit data element, x. Divide x into two 32-bit halves: x_L and x_R.

Then, for I=1 to 16:

$x_L = x_R \ XOR \ P_i$
$x_R = F(x_L) \ XOR \ x_R$
Swap x_L and x_R

After the sixteenth iteration, swap x_L and x_R to undo the last swap. Then $x_R = x_R \ XOR \ P_{17}$ and $x_L = x_L \ XOR \ P_{18}$. Finally, recombine x_L and x_R to get the ciphertext.

Function F looks like this: Divide x_L into four eight-bit quarters: a, b, c, and d.

$F(x_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) XOR \ S_{3,c}) + S_{4,d} \bmod 2^{32}$; see Figure 2 [my Figure B.3].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Decryption is exactly the same as encryption, except that $P_1, P_2, \cdots, P_{18}$ are used in the reverse order.

Implementations of Blowfish that require the fastest speeds should unroll the loop and ensure that all subkeys are stored in cache. For the purposes of illustration, I've implemented Blowfish in C; Listing One (page 98) is blowfish.h and Listing Two is blowfish.c[3]. A required data file is available electronically; see "Availability," page 3. [4]

Generating the Subkeys

The subkeys are calculated using the Blowfish algorithm, as follows:

1. Initialize first the P-array and then the four S-boxes, in order, with a fixed random string. This string consists of the hexadecimal digits of π.

2. XOR P_1 with the first 32 bits of the key, XOR P_2 with the second 32 bits of the key, and so on for all bits of the key (up to P_{18}). Cycle through the key bits repeatedly until the entire P-array has been XORed.

3. Encrypt the all-zero string with the Blowfish algorithm, using the subkeys described in steps #1 and #2.

---

[3]Schneier's revised C code is in Figure B.5.
[4]These pages refer to the original article in *Dr. Dobb's Journal.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Replace P_1 and P_2 with the output of step #3.

5. Encrypt the all-zero string using the Blowfish algorithm with the modified subkeys.

6. Replace P_3 and P_4 with the output of step #4.

7. Continue the process, replacing all elements of the P-array and then all four S-boxes in order, with the output of the continuously changing Blowfish algorithm.

In total, 521 iterations are required to generate all required subkeys. Applications can store the subkeys rather than re-executing this derivation process.

Design Decisions

The underlying philosophy behind Blowfish is that simplicity of design yields an algorithm that is easier both to understand and to implement. Hopefully, the use of a streamlined Feistel network (the same structure used in DES, IDEA, and many other algorithms), a simple S-box substitution, and a simple P-box substitution, will minimize design flaws.

For details about the design decisions affecting the security of Blowfish, see "Requirements for a New Encryption Algorithm" (by B. Schneier and N. Fergusen) and "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)" (by B. Schneier), both to be included in *Fast Software Encryption*, to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be published by Springer-Verlag later this year as part of their *Lecture Notes on Computer Science* series. The algorithm is designed to be very fast on 32-bit microprocessors. Operations are all based on a 32-bit word and are one-instruction XORs, ADDs, and MOVs. There are no branches (assuming you unravel the main loop). The subkey arrays and the instructions can fit in the on-chip caches of both the Pentium and the PowerPC. Furthermore, the algorithm is designed to be resistant to poor implementation and programmer errors.

I'm considering several simplifications to the algorithm, including fewer and smaller S-boxes, fewer rounds, and on-the-fly subkey calculation.

Conclusions

At this early stage, I don't recommend implementing Blowfish in security systems. More analysis is needed. I conjecture that the most efficient way to break Blowfish is through exhaustive search of the keyspace. I encourage all cryptanalytic attacks, modifications, and improvements to the algorithm.

However, remember one of the basic rules of cryptography: The inventor of an algorithm is the worst person to judge its security. I am publishing the details of Blowfish so that others may have a chance to analyze it.

Blowfish is unpatented and will remain so in all countries. The algorithm is hereby placed in the public domain and can be freely used by anyone.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix B

Figures Relating to the Blowfish Encryption Algorithm

Figure B.1	Blowfish is a Feistel Network Consisting of 16 Rounds	86
Figure B.2	A Round in the Feistel Network	87
Figure B.3	Blowfish Function F.	88
Figure B.4	Blowfish Function F as a 4-in 32-out Function	89
Figure B.5	Schneier's Revised C Code for the Blowfish Function F	90

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

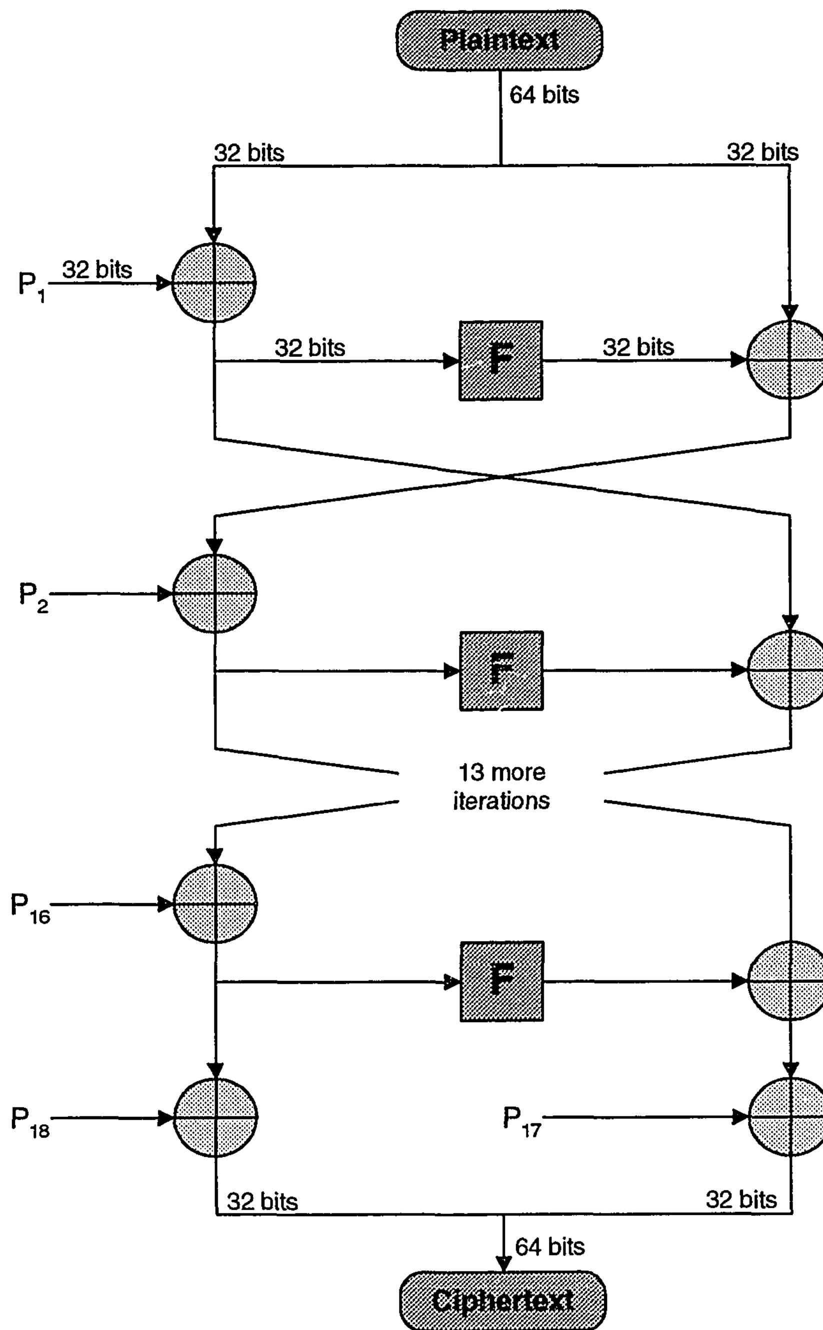
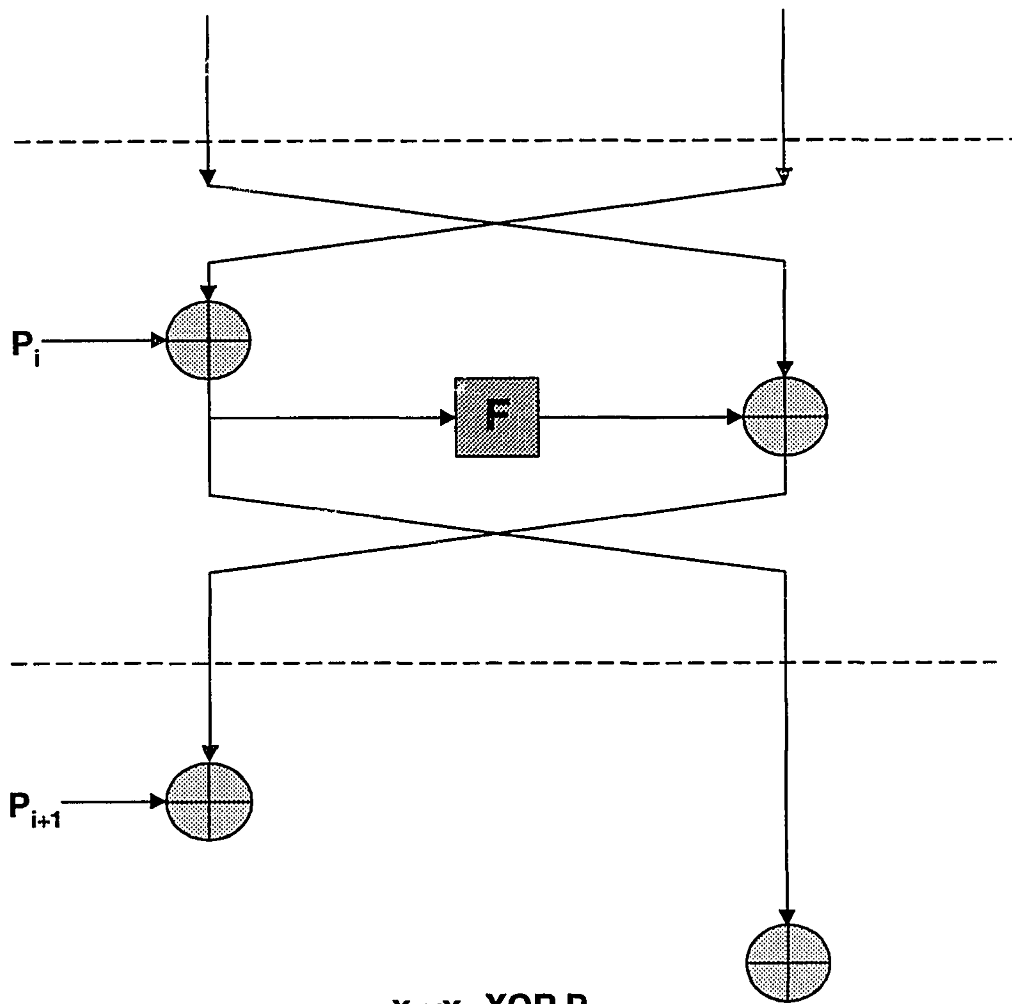


Figure B.1
Blowfish is a Feistel Network Consisting of 16 Rounds[5]

[5] [SCHNEIER94b, p.40]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



$x_L = x_L$ XOR P_i

$x_R = F(x_L)$ XOR x_R

Swap x_L and x_R

Figure B.2
A Round in the Feistel Network

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

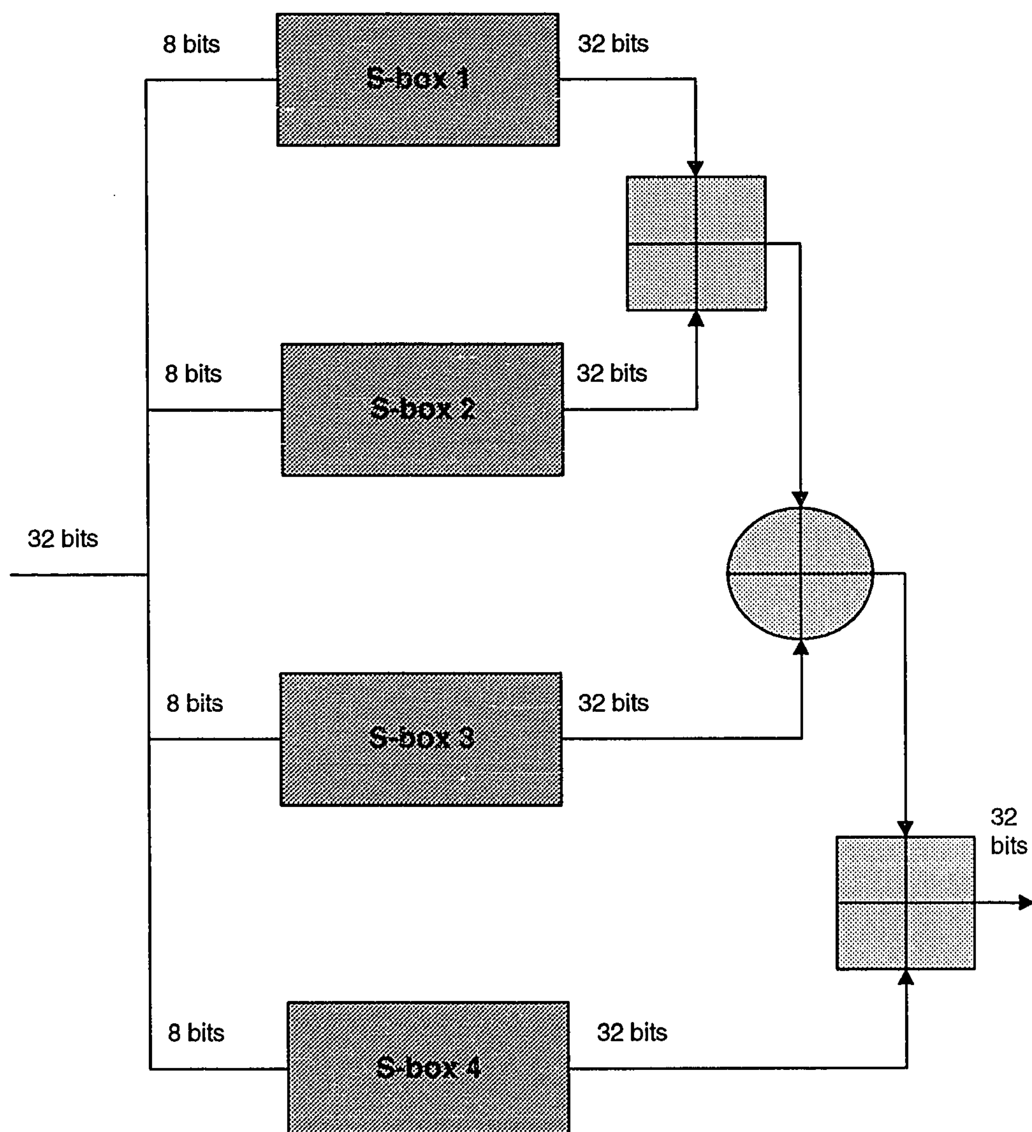


Figure B.3

Blowfish Function F[6]

---

[6] [SCHNEIER94b, p.40]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

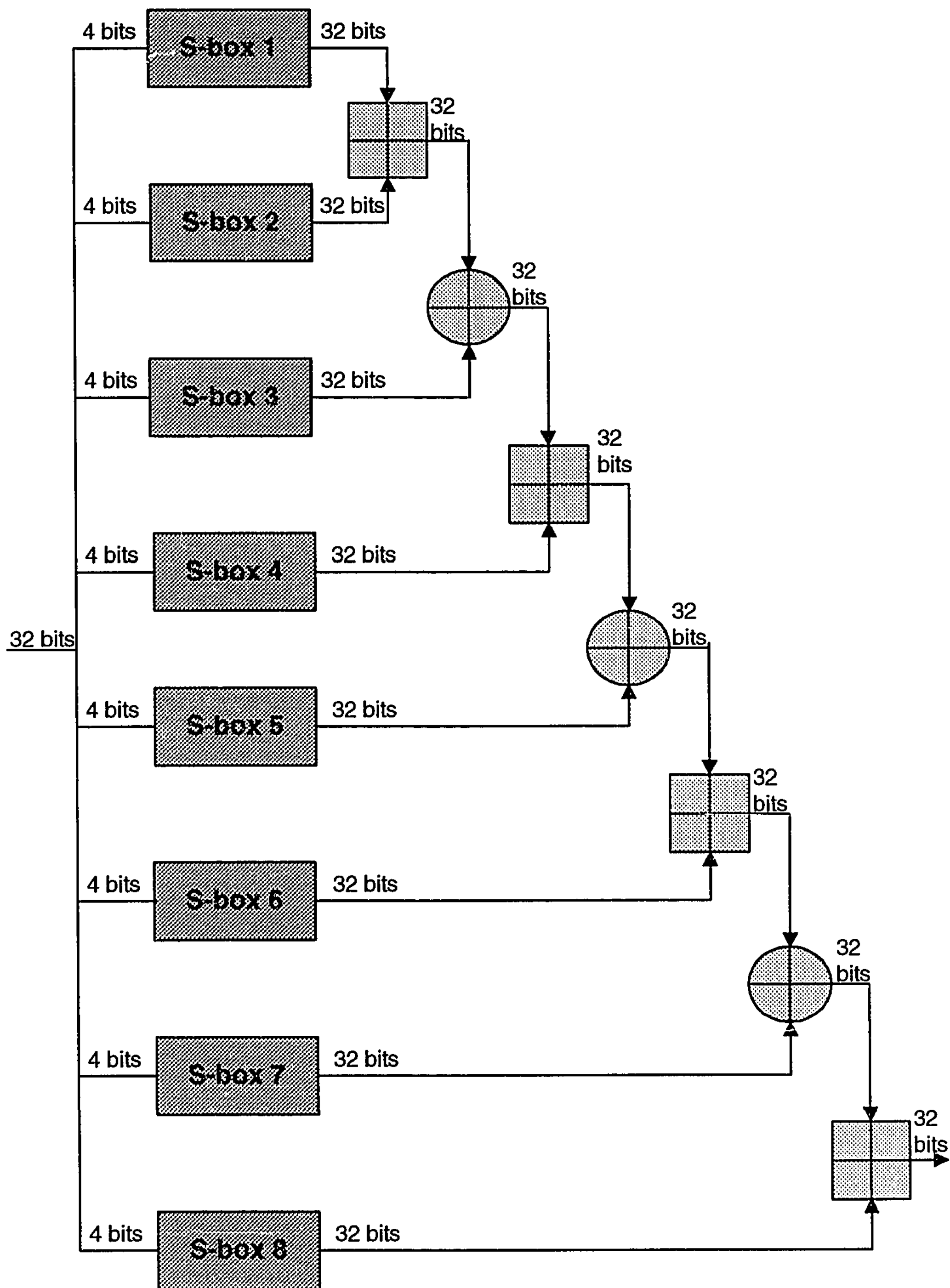


Figure B.4

A Variant Blowfish Function F as a 4-in 32-out Function

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
unsigned long F(unsigned long x)
{
    unsigned short a;
    unsigned short b;
    unsigned short c;
    unsigned short d;
    unsigned long y;
    d = x & 0x00FF;
    x >>= 8;
    c = x & 0x00FF;
    x >>= 8;
    b = x & 0x00FF;
    x >>= 8;
    a = x & 0x00FF;
    y = S[0][a] + S[1][b];
    y = y ^ S[2][c];
    y = y + S[3][d];
    return y;
}
```

Figure B.5

Schneier's Revised[7] C Code

For the Blowfish Function F

[7]As of May, 1994

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix C

Figures Relating to Occam and the Transputer

Figure C.1	Layout of CSA transputer kit board	92
Figure C.2	Transputer Links and the Corresponding Occam Channels	93
Figure C.3	On Chip Scheduler	94

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

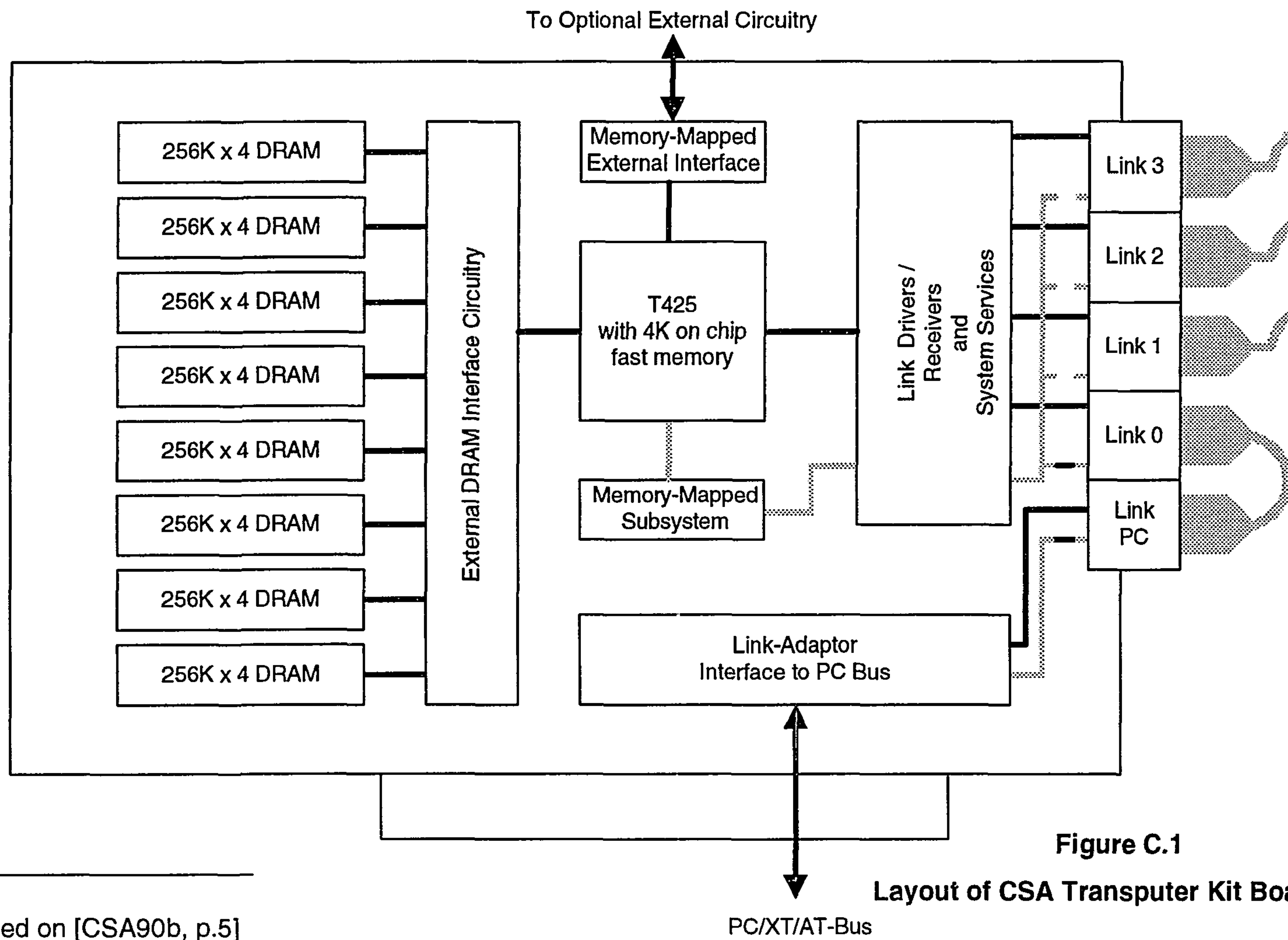
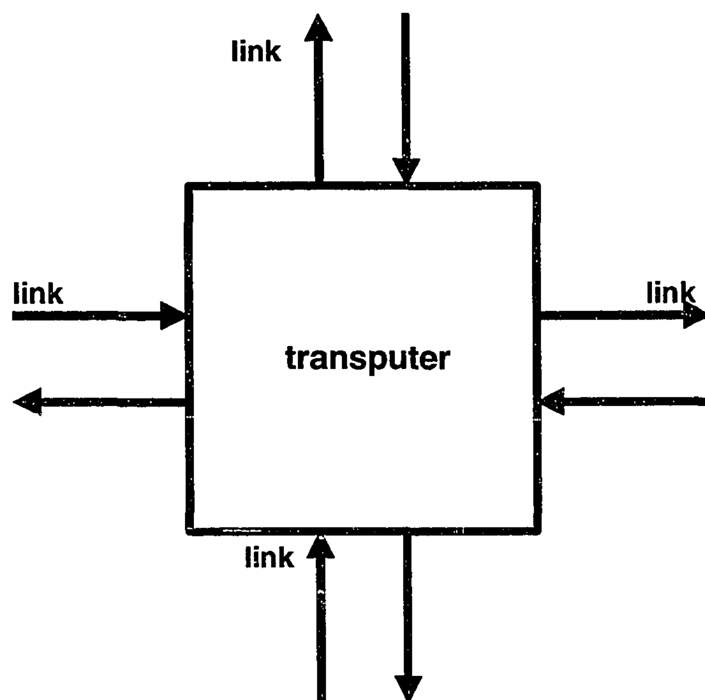


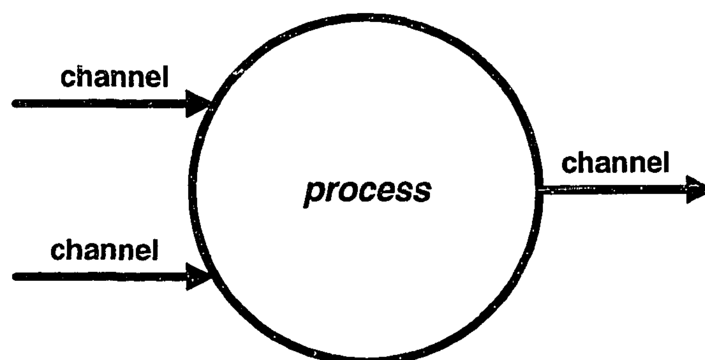
Figure C.1

Layout of CSA Transputer Kit Board[8]

[8] Based on [CSA90b, p.5]



a: Transputer with Links



b: Process with Channels

Figure C.2

Transputer Links and the Corresponding Occam Channels[9]

---

[9] Based on [CSA90d, p.83]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

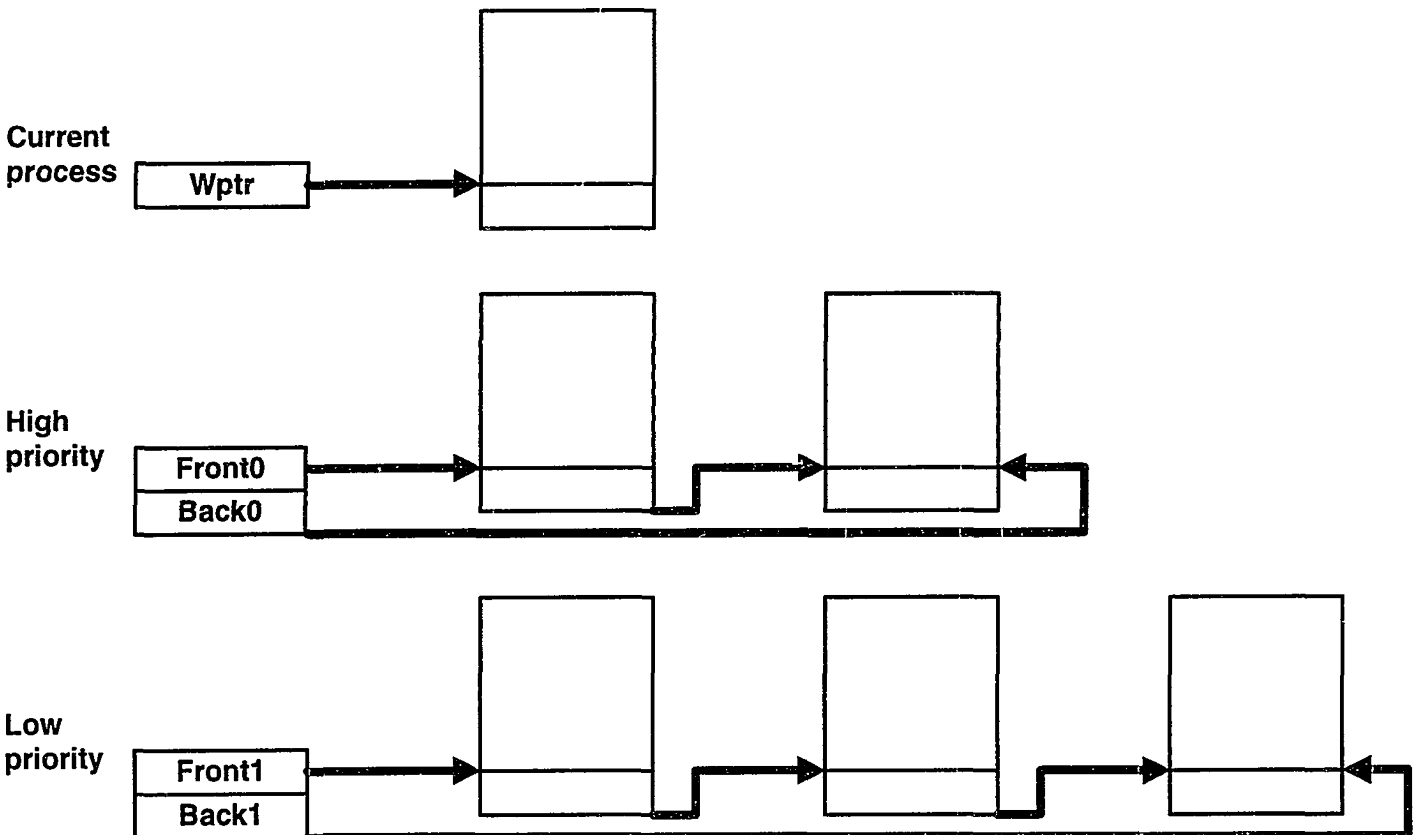


Figure C.3

On Chip Scheduler[10]

[10] Based on [CSA90d, p.110]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

AHO74 Aho, A., J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1974.

CSA90a[11] *A Tutorial Introduction to Occam Programming.* INMOS Limited, 1990. INMOS is a member of the SGS-Thomson Microelectronics Group.

CSA90b *User Guide: Theory of Operations; Installation; Schematics.* Computer System Architects, 1990.

CSA90c *Occam 2 Toolset User Manual.* INMOS Limited, 1990. INMOS is a member of the SGS-Thomson Microelectronics Group.

CSA90d *Occam and the Transputer: A Workbook.* Text SGS-Thomson Microelectronics, 1990; Format Computer Systems Architects, 1990.

CSA90e *The Transputer Instruction Set: A Compiler Writer's Guide.* INMOS Limited, 1990. INMOS is a member of the SGS-Thomson Microelectronics Group.

[11]Computer System Architects (manufacturers of the Transputer Education Kit); hereafter abbreviated as CSA.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CSA90f *Transputer Technical Specifications.* INMOS Limited, 1990. INMOS is a member of the SGS-Thomson Microelectronics Group.

CSA94 Private Correspondence, 1994.

INMOS88 INMOS Limited. *Occam 2 Reference Manual.* New York: Prentice Hall, 1988.

INMOS89 *TRANSPUTER APPLICATIONS NOTEBOOK Architecture and Software.* INMOS Limited, 1989. INMOS is a member of the SGS-Thomson Microelectronics Group.

JONES88 Jones, G. and M. Goldsmith. *Programming in Occam 2.* New York: Prentice Hall, 1988

JAJA92 JaJa, J. *An Introduction to Parallel Algorithms.* Reading, MA: Addison-Wesley, 1992.

LEIGHTON92 Leighton, F. T. *An Introduction to Parallel Algorithms and Architecture: Arrays, Trees, Hypercubes.* San Mateo, CA: Morgan Kaufmann, 1992.

SCHNEIER94a Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* New York: John Wiley & Sons, 1994.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SCHNEIER94b Schneier, B. "The Blowfish Encryption Algorithm." *Dr. Dobb's Journal*, April, 1994, pp. 38-40, 98-99.

SCHNEIER94c Schneier, B. Personal communications.

TANENBAUM87 Tanenbaum, A. *Operating Systems: Design and Implementation*. Englewood, NJ: Prentice Hall, 1987.

WELSH88 Welsh, D. *Codes and Cryptography*. Oxford: Oxford University Press, 1988.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.