1989

Computer construction of (4,4,v)-threshold schemes from Steiner Quadruple Systems

W. Monroe John

Rochester Institute of Technology
Computer Science Department

Computer Construction of $(4,4,v)$-Threshold Schemes from Steiner Quadruple Systems

by

W. John Monroe

17 July 1989

A thesis, submitted to
The Faculty of the Computer Science Department
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Dr. Donald L. Kreher (Advisor)

Dr. Stanislaw P. Radziszowski

Dr. Peter G. Anderson

Rochester Institute of Technology
Computer Science Department
Masters Thesis

Computer Construction of $(4,4,v)$-Threshold Schemes from Steiner Quadruple Systems

W. John Monroe

24 July 1989
Date

Computer Construction of (4,4,v)-Threshold Schemes from Steiner Quadruple Systems

W. John Monroe

ABSTRACT

A construction for $3v/4$ pairwise disjoint quadruple systems on v points has been given by Lindner. This thesis looks at an implementation of nearly optimal $(4,4,v)$–threshold schemes based on his construction. These threshold schemes will have $3v/4$ keys, whereas the best implementation known to date is based on a construction given by Shamir and yields only $v/4$ keys. Lindner's construction depends heavily on the existence of an N_2–latin square of order $v/4$, thus several constructions for them have also been implemented. Unfortunately, due to the combinatorial nature of the problem, the limitations of this implementation are an important issue and will be discussed.

TABLE OF CONTENTS

Computer Construction of $(4,4,v)$-Threshold Schemes from Steiner Quadruple Systems

1. Introduction

A threshold scheme is a method of distributing partial information about a key (a data encryption code, a trade secret, the combination to a bank vault, etc.) to each of a group of participants in such a way that any t or more of them acting in unison can determine the correct key, but any fewer than t participants cannot determine the correct key. The value t is called the *threshold.* A more formal definition follows.

DEFINITION 1. Given X, $|X| = v$, a set of shadows (pieces of partial information), and K, $|K| = m$, a set of keys, a (t,w,v)–*threshold scheme* is a pair $(\mathcal{B},\phi)$, where $\mathcal{B}$ is a set of distinct w–subsets (blocks) of X, $|\mathcal{B}| = b$, and $\phi\colon \mathcal{B}\rightarrow K$, such that:

(1) for *all* $S \subseteq X$ *such that* $|S| = t$, $|\{\phi(B) : S \subseteq B \in \mathcal{B}\}| \leq 1$, and
(2) for *all* $S \subseteq X$ *such that* $|S| < t$, $|\{\phi(B) : S \subseteq B \in \mathcal{B}\}| \neq 1$.

In other words, any t shadows determine *at most* one key while any less than t shadows do not determine a *unique* key.

1.1. Background

The idea of threshold schemes was first introduced in 1979 by Shamir [Sh] and Blakely [B]. Since that time a variety of threshold scheme constructions have been given. The majority of these constructions are based on linear algebra (e.g. Kothari [K]). Recently, however, Stinson and Vanstone [SV] have given some constructions

for threshold schemes based on combinatorial designs. In this thesis we consider a construction for threshold schemes that, while not optimal, beats the threshold value of Stinson and Vanstone's constructions; and that, while accommodating only four participants, contains three times as many keys as Shamir's schemes. Then we discuss the major issues of an implementation of these schemes. The issue of greatest concern will be the space complexity of the system.

In order to demonstrate the rationale behind the choice of combinatorial designs to derive constructions of threshold schemes, it is necessary to first discuss the notion of security in threshold schemes and give some definitions and theorems. To begin with, consider the key space K containing m keys $k_1, k_2, \ldots, k_m$ such that $k_i \neq k_j$ for $0 < i \neq j \leq m$. The probability distribution on K is a discrete uniform distribution, and for all $0 < i \leq m$, $p(k_i) = 1/m$. That is to say, in the absence of any other information, the probability of choosing the correct key is simply the reciprocal of the number of keys. Now consider the set X of pieces of partial information. Every subset $S \subseteq X$ defines a conditional probability distribution on K, where the odds of "guessing" the key $k \in K$ are equal to $p(k \mid S)$. With this in mind, the following concept is introduced.

DEFINITION 2. Given a (t,w,v)–threshold scheme and a non–negative integer $t' < t$, the threshold scheme is said to be *perfectly t'–secure* if for every subset $S \subseteq X$ such that $|S| = t'$ and $|\{B \in \mathcal{B} : S \subseteq B\}| \geq 1$, and for every key $k \in K$, $p(k \mid S) = p(k)$.

That is, any subset of fewer than t pieces of partial information (elements of X)

gives absolutely no information about any key.

Considering the above definition, one could observe that a threshold scheme with some sort of uniform structure would be more likely to be *perfectly* t'*−secure.* Before going on then, consider the following definition of one sort of uniformity for a threshold scheme.

DEFINITION 3. A threshold scheme is said to be *regular* if

(1) for every key $k \in K$, $|\{B \in \phi^{-1}(k)\}| = b/m$, and

(2) each block $B \in \phi^{-1}(k)$ is chosen with equal probability m/b.

So, not only does a *regular* threshold scheme have a uniform structure, but the method of choosing which block of partial information to distribute must not favor any single block over any other block.

The two preceding definitions are now combined in the following result.

LEMMA 1. *(Stinson and Vanstone [SV]) A regular (t,w,v)–threshold scheme $(\mathcal{B},\phi)$ is perfectly t'−secure if and only if for every $S \subseteq X$ such that $|S| = t'$, there exists a non-negative integer $\lambda(S)$ such that for every key $k \in K$,*

$$|\{B \in \phi^{-1}(k) : S \subseteq B\}| = \lambda(S).$$

From this result, a simple counting argument allows one to immediately obtain the following Lemma due to Stinson and Vanstone [SV].

LEMMA 2. *If a regular threshold scheme is perfectly t'–secure, then it is perfectly t''–secure for all t'', $1 \le t'' \le t'$.*

In order to help clarify these ideas, a tabular representation of a perfectly t'–secure regular threshold scheme is presented in Table I. In the table an entry of 1 in the i^{th} row and j^{th} column indicates that $S_i \subseteq B_j$ ($|S_i| = t'$) and an entry of 0 indicates that $S_i \not\subseteq B_j$. Note that every key is associated with the same number of blocks, and in any given row the number of 1 entries is the same for each section (i.e. for each key) of the table; in fact, that number is $\lambda(S)$.

TABLE I

	K_1				K_2				$\cdots$	K_m			
	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	$\cdots$	B_{4m-3}	B_{4m-2}	B_{4m-1}	B_{4m}
S_1	1	0	1	0	1	1	0	0		0	1	1	0
S_2	0	0	0	0	0	0	0	0		0	0	0	0
S_3	1	1	0	0	0	1	0	1		1	0	0	1
S_4	1	1	1	0	1	0	1	1		0	1	1	1
S_5	1	1	1	1	1	1	1	1	$\cdots$	1	1	1	1
.		.	.			.	.				.	.	
.		.	.			.	.				.	.	
.		.	.			.	.				.	.	
$S_{\binom{v}{t'}}$	0	0	1	0	1	0	0	0		0	1	0	0

Finally, to conclude this section, a definition characterizing threshold schemes that are perfectly $(t-1)$–secure is presented.

DEFINITION 4. A regular (t, w, v)–threshold scheme is said to be *perfect* if it is perfectly $(t-1)$–secure.

Given the above background information, it is now possible to move on and give a combinatorial characterization of regular threshold schemes.

1.2. Perfect Threshold Schemes from a Combinatorial Perspective

To begin this section it will be necessary to define several terms. Consider first the definition of a w–uniform hypergraph.

DEFINITION 5. Let $\mathcal{A}$ be a collection of w–subsets (blocks) of a v–set X. Then the pair $(X, \mathcal{A})$ is said to be a *w–uniform hypergraph* on v points.

Note that $\mathcal{A}$ was referred to as a *collection* rather than a *set* of blocks. This is because $\mathcal{A}$ is allowed to contain "repeated" blocks. Such collections are sometimes referred to as *multisets*. When dealing with multisets, one item of interest will be the number of times a particular block (subset) is "repeated". That number is known as the *multiplicity* of that block. In the case that $\mathcal{A}$ is a set each block has multiplicity one and $(X, \mathcal{A})$ is said to be *simple*.

Suppose $(X, \mathcal{A})$ is a w–uniform hypergraph on v points, as discussed above. Then the *t'–induced hypergraph* of $(X, \mathcal{A})$ is defined as follows:

DEFINITION 6. Given any non–negative integer $t' \leq w$ and a w–uniform hypergraph $(X, \mathcal{A})$, then the *t'–induced hypergraph* of $(X, \mathcal{A})$ is the pair $(X, \mathcal{A}(t'))$, where $\mathcal{A}(t') = \bigcup_{A \in \mathcal{A}} \{S \colon |S| = t',\ S \subseteq A\}$.

Notice that even if $(X, \mathcal{A})$ is simple, $(X, \mathcal{A}(t'))$ need not be unless $t' = w$, in which case $\mathcal{A}$ and $\mathcal{A}(t')$ are equivalent.

At this point it is possible to define a relationship between two w–uniform hypergraphs.

DEFINITION 7. Two w–uniform hypergraphs $(X, \mathcal{A}_1)$ and $(X, \mathcal{A}_2)$ are said to be *t–compatible* if (1) $\mathcal{A}_1(t) \bigcap \mathcal{A}_2(t) = \phi$, and (2) $\mathcal{A}_1(t-1) = \mathcal{A}_2(t-1)$.

In other words, two w–uniform hypergraphs are *t–compatible* if their t–induced hypergraphs are disjoint and their $(t-1)$–induced hypergraphs are identical.

It is now possible to characterize perfect (t, w, v)–threshold schemes in terms of t–compatible w–uniform hypergraphs.

THEOREM 3. *(Stinson and Vanstone [SV]) There exists a perfect (t, w, v)–threshold scheme having v shadows and m keys if and only if there exist m pairwise t–compatible w–uniform hypergraphs on v points.*

As in any security scheme, the total number of possible keys is important; there must be enough keys to discourage any attempt by the enemy to systematically check all possible keys. Consequently, one would like a way to determine the maximum number of keys that a perfect (t, w, v)–threshold scheme can handle. From Theorem 1 it can be seen that this maximum number of keys is just the maximum number $m(t, w, v)$ of pairwise t–compatible w–uniform hypergraphs on v points. An upper bound on $m(t, w, v)$ due to Stinson and Vanstone is presented in Theorem 4. Determining the exact number $m(t, w, v)$ is NP–hard [SV].

THEOREM 4. *(Stinson and Vanstone [SV])* $m(t, w, v) \leq (v-t+1) \;/\; (w-t+1)$.

Note that in schemes where $w=t$, $m(t,w,v)$ is simply bounded by $v-t+1$.

The next task is to determine when the upper bound given above can actually be obtained. Before proceeding, however, a brief description of *Steiner systems* is in order. Given integers t, k and v, $1 \le t \le k \le v$, an $S(t,k,v)$ Steiner system, hereafter referred to as an $S(t,k,v)$ system, is a simple (i.e. has no "repeated" blocks) k–uniform hypergraph $(X, \mathcal{A})$ on v points such that every t–subset of those v points occurs in *exactly* one block. Put another way, $\mathcal{A}(t)$ is simply the *set* of all t–subsets of X. Furthermore, a Steiner system is said to be *partitionable* if it is possible to partition $\mathcal{A}$ (the set of blocks) into the subsets $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_j$, $j=(v-t+1)/(k-t+1)$, such that each k–uniform hypergraph $(X, \mathcal{A}_i)$, $1 \le i \le j$, is an $S(t-1,k,v)$ Steiner system. (For a more thorough treatment of Steiner systems and designs in general see [BJL].) With this brief definition of Steiner systems in mind, we present the following result due to Stinson and Vanstone.

THEOREM 5. *(Stinson and Vanstone [SV])* $m(t,w,v) = (v-t+1)/(w-t+1)$ *if and only if there exists an* $S(t,w,v)$ *Steiner system that can be partitioned into* $S(t-1,w,v)$ *Steiner systems.*

Stinson discusses in [St] several situations where this theorem allows one to determine exact values for $m(t,w,v)$. In those cases where $m(t,w,v) = (v-t+1)/(w-t+1)$, the resulting (t,w,v)–threshold scheme is described as *optimal*.

1.3. A Threshold Scheme from Pairwise Disjoint $S(2,3,9)$ Systems

As an example of what we are trying to accomplish, let us consider a scaled–down example. Instead of looking at some $(4,4,v)$–threshold scheme, I shall present a $(3,3,9)$–threshold scheme. This scheme is generated by partitioning the $S(3,3,9)$ system into seven $S(2,3,9)$ systems. In 1974 R. Wilson [W] gave a scheme for partitioning an $S(3,3,v)$ system into $v-2$ disjoint $S(2,3,v)$ systems, whenever $v-2 \equiv 7$ *modulo* 8 is a prime. Stinson and Vanstone [SV] make use of this partitioning scheme to give a general construction for optimal $(3,3,v)$–threshold schemes, again for $v-2 \equiv 7$ *modulo* 8 a prime.

Let $(X, \mathcal{A})$ denote the $S(3,3,9)$ system where $X = \mathbb{Z}_7 \bigcup \{a, \tilde{a}\}$ and $\mathcal{A}$ is the set of all $84 = \binom{9}{3}$ triples (blocks) on the $|X| = v$ points. To partition the 84 triples into seven $S(2,3,9)$ systems (X, A_i), $0 \leq i < 7$, first construct the 12 blocks of A_0 as follows [SV]:

> 1 block: $\{a, \tilde{a}, 0\}$;
>
> 5 blocks: $\{x, y, z\}$, $x \neq y \neq z \neq x$ and $x+y+z=0$, $x, y, z \in \mathbb{Z}_7$;
>
> 3 blocks: $\{a, x, -2x\}$, x a quadratic residue in $\mathbb{Z}_7$; and
>
> 3 blocks: $\{\tilde{a}, x, -2x\}$, x a quadratic non-residue in $\mathbb{Z}_7$.

To complete the partition of the $S(3,3,9)$ system into $S(2,3,9)$ systems it is necessary to extend addition *modulo* 7 to all points of X by defining $a+i=a$ and $\tilde{a}+i=\tilde{a}$, for $i \in \mathbb{Z}_7$. The remaining systems are constructed by simply letting $A_i = \{\{x+i,\ y+i,\ z+i\} : \{x,y,z\} \in A_0\}$, for any $i \in \mathbb{Z}_7$. The partition is given in Table II.

TABLE II

A_0	A_1	A_2	A_3	A_4	A_5	A_6
$a\,\tilde{a}\,0$	$a\,\tilde{a}\,1$	$a\,\tilde{a}\,2$	$a\,\tilde{a}\,3$	$a\,\tilde{a}\,4$	$a\,\tilde{a}\,5$	$a\,\tilde{a}\,6$
016	120	231	342	453	564	605
025	136	240	351	462	503	614
034	145	256	360	401	512	623
124	235	346	450	561	602	013
356	460	501	612	023	134	245
$a\,15$	$a\,26$	$a\,30$	$a\,41$	$a\,52$	$a\,63$	$a\,04$
$a\,23$	$a\,34$	$a\,45$	$a\,56$	$a\,60$	$a\,01$	$a\,12$
$a\,46$	$a\,50$	$a\,61$	$a\,02$	$a\,13$	$a\,24$	$a\,35$
$\tilde{a}\,31$	$\tilde{a}\,42$	$\tilde{a}\,53$	$\tilde{a}\,64$	$\tilde{a}\,05$	$\tilde{a}\,16$	$\tilde{a}\,20$
$\tilde{a}\,62$	$\tilde{a}\,03$	$\tilde{a}\,14$	$\tilde{a}\,25$	$\tilde{a}\,36$	$\tilde{a}\,40$	$\tilde{a}\,51$
$\tilde{a}\,54$	$\tilde{a}\,65$	$\tilde{a}\,06$	$\tilde{a}\,10$	$\tilde{a}\,21$	$\tilde{a}\,32$	$\tilde{a}\,43$

The pair $(X, \mathcal{A})$ thus forms a (3,3,9)–threshold scheme with $K = \{0,1,...,6\}$. If we assume that for any given key the block of shadows to be distributed is chosen at random, then $(X, \mathcal{A})$ is clearly regular. And since each A_i is an $S(2,3,9)$ system, then $(X, \mathcal{A})$ must be perfectly 2–secure. Thus, $(X, \mathcal{A})$ is a perfect (3,3,9)–threshold scheme. In addition, it can be seen directly from Theorem 5 that this scheme is optimal.

The construction given above for this scheme is easily generalized to a scheme with any number of keys, k, where $k \equiv 7$ *modulo* 8 is a prime [SV]. In the case presented here with $k = 7$, all possible blocks of shadows were generated; however, in practice it is not necessary to generate any blocks in advance; nor is it necessary to generate more than two blocks when selecting a block of shadows to distribute. To select a block of shadows to be given out for k_i, simply generate a random block of A_0 and add i to each element of that block to obtain the block of shadows to distribute.

Determining the key corresponding to a group of shadows in this case is also relatively simple and can be done on average in $O(1)$ time [SV]. It is just a matter of determining which case of the construction generated the block and then performing the "inverse construction." So is is possible to easily and efficiently implement optimal $(3,3,v)$–threshold schemes. However, the major practical limitation to such a system is that it only involves three participants and has a threshold of only three. In practice, larger numbers of participants and a greater threshold are generally desirable. If a similar method could be found for partitioning an $S(4,4,v)$ system into Steiner Quadruple Systems, then it would be possible to construct a threshold scheme for four participants with a threshold of four.

Actually, Shamir's scheme [Sh], which is based on interpolation of polynomials in $GF(v)$, for v a prime, can be used to construct both $(3,3,v)$– and $(4,4,v)$–threshold schemes. In fact, it can be used to construct any (t,w,v)–threshold scheme, $0<t\leq w<v$ and v a prime [Sh]. However, such a scheme provides only v/w keys [St]. Stinson and Vanstone's $(3,3,v)$–threshold schemes, by contrast, provide $v-2$ keys; and the $(4,4,v)$ schemes that we are considering provide $3v/4$ keys—three times as many as Shamir's schemes with $t=w=4$.

2. Constructing $3v/4$ Pairwise Disjoint $S(3,4,v)$ Systems

Suppose it is possible to partition an $S(4,4,v)$ system into $S(3,4,v)$ systems in much the same way $S(3,3,v)$ systems can be partitioned into $v-2$ disjoint $S(2,3,v)$ systems, thereby creating a $(4,4,v)$–threshold scheme. Such a scheme with v shadows would be optimal and would have $v-3$ keys (Theorem 2). Unfortunately, no

simple and efficient method of partitioning $S(4,4,v)$ systems is known. In fact, no method is known. However, Lindner has given the following result.

THEOREM 6. *(Lindner [L2]) For all* $v > 16$, $v \equiv 8, 16$ *modulo* 24, *there exist at least* $3v/4$ *pairwise disjoint* $S(3,4,v)$ *systems.*

Combining this result with Theorem 3, we know $m(4,4,v) \geq 3v/4$ for these values of v. Otherwise, the best lower bound to date [St][Sh][T] for $m(4,4,v)$ is

$$m(4,4,v) \geq \begin{cases} v/3 & \text{for } v \equiv 0, 6 \textit{ modulo } 12, \\ v/4 & \text{for } v \equiv 4, 20 \textit{ modulo } 24,\ v/4 \text{ a prime power.} \end{cases}$$

In the paper [L2] where he gives this result, Lindner also gives a construction for the $3v/4$ pairwise disjoint $S(3,4,v)$ systems. We will use this construction as the backbone of a computerized implementation of perfect $(4,4,v)$–threshold schemes. Again, the schemes produced will not be optimal; however, they will have significantly more keys than any other current implementation.

The implementation will consist of two main tasks. The first is to generate a random block of the scheme whose four elements (shadows) are to be distributed among the four participants. The set of shadows will simply consist of the integers $0, 1, ..., (v-1)$, so we could simply select four of those v integers at random; however, it would then be necessary to determine which, if any, of the $3v/4$ keys that block would be associated with. Instead, we will randomly select one of the $3v/4$ keys, use Lindner's construction to generate the $S(3,4,v)$ associated with that key, and then randomly select one of those blocks to distribute. The requirement that each block be chosen with equal probability dictates that these selections be made

randomly.

The second task that must be performed is to determine the appropriate key from a given block. This is significantly more complicated than the first task. The scheme needs to be large enough (contain enough keys) that someone cannot simply try every possible key. In such a system it clearly will be impractical, if not impossible, to simply store all $3v/4$ pairwise disjoint $S(3,4,v)$ systems and then search them to find a matching block. It would thus be very valuable to have a computationally efficient "inverse construction" for decoding a block of shadows, but this appears to be difficult to achieve. As a result, a rather brute force approach is taken. The implementation details of this task, as well as those of the first task, will be discussed in the next chapter; but first we need to look at Lindner's construction of the $3v/4$ pairwise disjoint $S(3,4,v)$ systems. In order to be consistent with Lindner's notation and terminology, we will take $v \equiv 2,4$ *modulo* 6, $v > 4$, for the remainder of this thesis. So we will be constructing $3v$ pairwise disjoint $S(3,4,4v)$ systems. Note that $v \equiv 2,4$ *modulo* 6, $v > 4$, implies $4v \equiv 8,16$ *modulo* 24, $4v > 16$, as required by Theorem 6.

2.1. Supporting Algorithms

The main construction relies on three important supporting algorithms: an algorithm for constructing an $S(3,4,2v)$ system from an $S(3,4,v)$ system, an algorithm for constructing an $S(3,4,4v)$ system from an $S(3,4,v)$ system, and the construction of N_2–latin squares. Both algorithms for constructing larger quadruple systems from any given quadruple system are due to Lindner [L2] who refers to them as the $2v$ *construction* and the $4v$ *construction*, respectively; and we will do

likewise. When constructing any quadruple system we will always choose the v–set of points to be $\mathbb{Z}_v$, the integers *modulo v*. Although any v–element set Q will do in the subsequent constructions, choosing $Q=\mathbb{Z}_v$ makes the programming job easier.

2.1.1. $2v$ Construction (Lindner [L2])

The $2v$ construction is relatively simple. Let (Q,q) be an $S(3,4,v)$ system on the set of points $Q=\mathbb{Z}_v$ and let α be any permutation on Q. Set Q' equal to the collection of ordered pairs $Q\times\{i,j\}$. Now, for each block (quadruple) $\{x,y,z,w\}\in q$, the following eight blocks of ordered pairs should be added to the new $S(3,4,2v)$ system, (Q',q'):

$$\begin{array}{ll}
\{(x,i),(y,i),(z,i),(w^\alpha,j)\}, & \{(x,j),(y,j),(z,j),(w^{\alpha^{-1}},i)\},\\
\{(x,i),(y,i),(z^\alpha,j),(w,i)\}, & \{(x,j),(y,j),(z^{\alpha^{-1}},i),(w,j)\},\\
\{(x,i),(y^\alpha,j),(z,i),(w,i)\}, & \{(x,j),(y^{\alpha^{-1}},i),(z,j),(w,j)\},\\
\{(x^\alpha,j),(y,i),(z,i),(w,i)\}, & \{(x^{\alpha^{-1}},i),(y,j),(z,j),(w,j)\}.
\end{array}$$

To complete the construction of (Q',q'), the block $\{(x,i),(y,i),(x^\alpha,j),(y^\alpha,j)\}$ is added to q' for every subset $\{x,y\}\subseteq Q$. Then (Q',q') will be an $S(3,4,2v)$ system on $Q\times\{i,j\}$.

2.1.2. $4v$ Construction (Lindner [L2])

The $4v$ construction is somewhat more involved than the $2v$ construction, consisting of four parts. In addition to those things needed for the $2v$ construction, another permutation β on Q is necessary; β need not be related in any way to α. Also the set Q' of ordered pairs must be modified so that we have $Q'=Q\times\{i,j,s,t\}$; and we let $\pi=\{(i,j),(s,t)\}$.

The first part of this algorithm employs the $2v$ construction. That construction is performed twice on (Q,q): once using the points $Q \times \{i,j\}$ and once using the points $Q \times \{s,t\}$. All $\binom{2v}{3}/4$ blocks from each of the two $S(3,4,2v)$ systems created are placed in (Q',q'). Part two of this construction adds $8\binom{v}{2}$ blocks to the system by adding the following eight blocks for each $\{x,y\} \subseteq Q$:

$$\begin{array}{ll}
\{(x,i),(y,i),(x,s),(y^{\beta},t)\}, & \{(x,j),(y,j),(x,s),(y^{\beta},t)\}, \\
\{(x,i),(y,i),(y,s),(x^{\beta},t)\}, & \{(x,j),(y,j),(y,s),(x^{\beta},t)\}, \\
\{(x,s),(y,s),(x,i),(y^{\alpha},j)\}, & \{(x,t),(y,t),(x,i),(y^{\alpha},j)\}, \\
\{(x,s),(y,s),(y,i),(x^{\alpha},j)\}, & \{(x,t),(y,t),(y,i),(x^{\alpha},j)\}.
\end{array}$$

The third part of the construction generates $48\binom{v}{3}/4$ blocks. Note that $\binom{v}{3}/4$ is the number of blocks in q. So for each of those blocks we will add 48 new blocks to q' in the following way: for each of the $\binom{4}{2}=6$ pairs $\{x,y\}$ in each block $\{x,y,a,b\} \in q$ add the eight blocks

$$\begin{array}{ll}
\{(x,i),(y,i),(a,s),(b^{\beta},t)\}, & \{(x,j),(y,j),(a,s),(b^{\beta},t)\}, \\
\{(x,i),(y,i),(b,s),(a^{\beta},t)\}, & \{(x,j),(y,j),(b,s),(a^{\beta},t)\}, \\
\{(x,s),(y,s),(a,i),(b^{\beta},j)\}, & \{(x,t),(y,t),(a,i),(b^{\beta},j)\}, \\
\{(x,s),(y,s),(b,i),(a^{\beta},j)\}, & \{(x,t),(y,t),(b,i),(a^{\beta},j)\}.
\end{array}$$

Finally, the fourth part constructs the remaining v^2 blocks needed to complete (Q',q') using the rule: for every ordered pair $(a,b) \in Q \times Q$ place the block $\{(a,i),(a^{\alpha},j),(b,s),(b^{\beta},t)\}$ into q'. Again, for a slightly more formal description of this construction see [L2].

2.1.3. N_2–Latin Square Constructions

The third and final supporting topic we need to discuss is the generation of N_2–latin squares. First of all, a *latin square* of order n is an $n \times n$ table of elements from $x \in \mathbb{Z}_n$, such that each element appears exactly once in each row and exactly once in each column of the table. Consider a latin square L of order n and let $L_{i,j}$ represent the entry in the i^{th} row and j^{th} column of L. For L to be an N_2–latin square there must not exist any $i,j,k_1,k_2 \in \mathbb{Z}_n$ such that $L_{i,j} = L_{i+k_1,j+k_2}$ *and* $L_{i+k_1,j} = L_{i,j+k_2}$. In other words, an N_2–latin square is simply a latin square that does not contain any subsquares of order two.

Unfortunately, constructing N_2–latin squares is not as simple as it might at first seem. There are several cases which must be considered based on the order n of the square to be constructed. The first case, n odd, is rather trivial since the group operation table for the cyclic group of order n is an N_2–latin square. Next, let us consider those orders n that are even and have an odd factor m; this includes all even values that are not powers of two. In this case Kotzig, Lindner, and Rosa [KLR] give an algorithm for generating an N_2–latin square of order n using an N_2–latin square of order m as the basis for a "doubling" scheme. They start with the N_2–latin square of order m and then create an N_2–latin square of order $2m$. This step is repeated to obtain an N_2–latin square of order $4m$, and so on until the N_2–latin square of order $n = 2^k m$ has been generated.

N_2–latin squares of orders that are powers of two are the most difficult to construct. For orders two and four, no N_2–latin square exists [KLR]. There are exactly three non-isomorphic N_2–latin squares of order eight which were found by

Denniston [D] via an exhaustive computer search. McLeish [M] presents constructions for N_2–latin squares of all orders $n = 2^k$, $k \geq 6$. Her constructions rely on the direct product and singular direct product of certain smaller N_2–latin squares. Finally in 1978, Kotzig and Turgeon [KT] settled the problem of N_2–latin square existence by presenting a construction for all even orders n, $n \not\equiv 0$ *modulo* 3 and $n \not\equiv 3$ *modulo* 5, thereby covering the remaining open orders $n = 16$ and $n = 32$.

2.2. The Construction

The $4v$ construction as described above in section 2.1, will generate one $S(3,4,4v)$ system. Now we need to look at some way of making repeated use of it to generate $3v$ of them that are pairwise disjoint. The method we are going to describe is due to Lindner [L2].

To begin, we will need to define a few terms. Recall that $Q = \mathbb{Z}_v$, and let B_{ij} be a collection of quadruples. Then $(Q \times \{i,j\}, B_{ij1})$, $(Q \times \{i,j\}, B_{ij2}), \ldots,$ $(Q \times \{i,j\}, B_{ijv})$, $i \neq j \in \{0,1,2,3\}$, will represent any collection of v pairwise disjoint $S(3,4,2v)$ systems. Such a collection can be obtained by performing the $2v$ construction v times on a given $S(3,4,v)$ system using any collection of v permutations on Q that form an N_2–latin square of order v.

Next we define $B(\pi,\alpha,\beta)$ to be the representation for the collection of blocks generated by parts two, three and four of the $4v$ construction. Also, let $\pi_1 = \{(0,1),(2,3)\}$, $\pi_2 = \{(0,2),(1,3)\}$, $\pi_3 = \{(0,3),(1,2)\}$, and ϕ be a v–cycle on Q. It should be noted that v will always be even since quadruple systems exist only on the orders $v \equiv 2, 4$ *modulo* 6 [H].

Finally, we can construct the $3v$ pairwise disjoint $S(3,4,4v)$ systems on the set of points, $S = Q \times \{0,1,2,3\}$, as follows:

(a) v quadruple systems $(S,\ B_{01k} \cup B_{23k} \cup B(\pi_1, \phi^k, \phi^k))$, $k = 1,2,\ldots,v$;

(b) v quadruple systems $(S,\ B_{02k} \cup B_{13k} \cup B(\pi_2, \phi^k, \phi^{k+2}))$, $k = 1,2,\ldots,v$;

(c) v quadruple systems $(S,\ B_{03k} \cup B_{12k} \cup B(\pi_3, \phi^k, \phi^{k+1}))$, $k = 1,2,\ldots,v$.

Again, while any four–element set would do, the choice of the set $\{0,1,2,3\}$ makes the programming task easier. For a proof that the collection of quadruple systems hereby produced are in fact pairwise disjoint, refer to [L2].

3. Implementing $(4,4,4v)$–Threshold Schemes

In order to create a practical implementation of a $(4,4,4v)$–threshold scheme based on Lindner's construction of $3v$ pairwise disjoint $S(3,4,4v)$ systems there are three basic things that are necessary:

(1) choosing an admissible value for v and constructing the $3v$ pairwise disjoint $S(3,4,4v)$ systems;

(2) for a given key value k, $0 \leq k < 3v$, selecting a random block of the k^{th} quadruple system and distributing the shadows to the participants;

(3) taking a set of four shadows and determining which key, if any, it represents.

The difficulty here is that the number of blocks in each quadruple system increases exponentially with v; thus, we will quickly run out of space if we try to store all the pairwise disjoint $S(3,4,4v)$ systems. Fortunately, it is not necessary to actually store them.

3.1. Generating $3v$ Pairwise Disjoint $S(3,4,4v)$ Systems

Though it is not necessary to actually store any of the $3v$ pairwise disjoint $S(3,4,4v)$ systems, an implementation which will generate them will serve as the basis for the implementation of the threshold scheme. In fact, in the course of development the first program that was written generated each of the $3v$ pairwise disjoint $S(3,4,4v)$ systems and stored them in separate files. While this program has limited practical value, due to storage constraints, it was very valuable for checking the correctness of other programs; it might also be useful in other research where a number of pairwise disjoint quadruple systems are desired. We will therefore proceed to describe this program.

One important consideration is the method used to internally represent quadruples. Recall that in section 2.2 the set S of $4v$ points was represented as the set of ordered pairs $Q \times \{0,1,2,3\}$. Since single integers are more efficient to represent on a machine than are ordered pairs, the implementation relies on a mapping of the ordered pairs of S onto the integers. Let $x \in Q$ and $y \in \{0,1,2,3\}$, then the mapping $(x,y) \rightarrow x + vy$ gives a set of points, $S' = \mathbb{Z}_{4v}$, making it possible to represent a quadruple as a set of $4v$ bits, where the four bits representing the four elements of the quadruple are ON and all other bits are OFF. There are two reasons for this. First, as a result of the mapping just described, the construction often involves the addition of a given multiple of v to each member of a set, which can thus be accomplished by a simple cyclic bit–shift. The second reason is that we can efficiently compare two quadruples using bit operations. Unfortunately, this method of representation limits the maximum value of $4v$ to the wordsize of the machine.

Thus, several routines were developed to perform bit operations on multiple–word sets. Unlike usual bit–shift operations which discard bits as they are shifted off, the new bit–shift operations are in theory cyclic and thus do not discard any shifted bits. This is necessary because we are performing *modulo* $4v$ arithmetic. In practice, however, the situation never arises where the usual type of bit–shift operations cannot be used. It should be noted here that this bit–set representation is only used for the new quadruples being constructed. At first glance, this method of representation may seem to be a waste of memory for large values of v; but since only a small constant number of these bit–sets are used, the waste is negligible.

As we saw in the previous chapter, a quadruple system and an N_2–latin square, both of order v, are needed in order to construct the $3v$ pairwise disjoint $S(3,4,4v)$ systems of order $4v$. Additionally, a v–cycle is necessary. Consequently, our implementation expects two ASCII input files––one containing the quadruple system and the other containing both the v–cycle and the N_2–latin square. The quadruple system file should have one quadruple per line, with the four points of each quadruple in any order, separated by spaces. The second file will contain $v+1$ lines, the first containing the v–cycle and the remaining v lines containing the N_2–latin square. Although it is necessary to read through the quadruple file twice for each $S(3,4,4v)$ system constructed, the file will not be read and stored internally due to its size––$c\binom{v}{3} = O(v^3)$ bytes. Instead, each quadruple will be read from the file when needed, stored temporarily in a $4{\times}1$ array, and then discarded. The v–cycle on the first line of the second file is, however, read and stored internally in a $v \times 1$ array before the actual construction begins; but the N_2–latin square

will be read one line at a time during the construction, each line being temporarily stored in a $v \times 1$ array, used, and then discarded when the next is read. In order to keep the system flexible, all of these arrays are dynamically allocated based on the value of v entered by the user.

Once the two files are read in, the construction can begin. The $3v$ pairwise disjoint $S(3,4,4v)$ systems are constructed one at a time, each being written out to a separate file. In fact, each quadruple is written out as it is generated. In this way a great savings of internal memory is realized and the maximum number of open files is not exceeded. As an aside, it should be noted that the $3v$ output files will each have the same format as the input quadruple system file.

For the actual construction, a loop of v iterations is used. The i^{th} time through this loop, a new permutation, the i^{th} row of the N_2–latin square, is read in and three pairwise disjoint $S(3,4,4v)$ systems are constructed—one of each of the three types, (a), (b) and (c) of the construction in section 2.2. In each quadruple system constructed, the first $2\binom{2v}{3}/4$ quadruples are those generated by part one of the $4v$ construction, the next $8\binom{v}{2}$ blocks are those generated by part two, the next $48\binom{v}{3}/4$ blocks are generated by the third part, and the final v^2 quadruples are those generated by part four of the $4v$ construction.

3.2. Constructing a Block of Shadows for a Given Key

As we noted above, it is not necessary to actually construct and store all $3v$ pairwise disjoint $S(3,4,4v)$ systems in order to implement the threshold scheme. Instead, we simply generate a random block of shadows associated with the user-

specified key, $k \in \{0,1,...,3v\}$. To accomplish this we simply modify the implementation discussed above. As above, the same two files are necessary and must be read in the same way. Additionally, the user must specify the key k; and a random integer n, $0 \leq n < \binom{v}{3}/4$, is generated. It is then possible, from the values of v and k, to directly compute the values of all indices of the various loops in the above implementation when the n^{th} block for the key k would be generated. Thus the loops can be eliminated from this version of the above implementation, allowing us to generate a random block of shadows in $O(v)$ time. The reason we cannot do it in $O(1)$ time is because it is necessary to read in the v–cycle and one line of the N_2–latin square. Note: this assumes direct file access capability on the quadruples file, without which the time to generate a random block of shadows is $O(v^3)$.

3.3. Decoding a Block of Shadows

Since no direct method is known for decoding a set of shadows, it is necessary to begin generating all quadruples until one is found that matches the given set of shadows. To do this requires very little modification to the implementation discussed in section 2 above. Instead of writing out the quadruples as they are generated, we simply compare each one to the set of shadows entered by the user and then discard it, until a match is found. When a quadruple is generated that matches the given set of shadows, we simply output the number of the quadruple system currently being "constructed" as the key. If no match is found then the set of shadows is not valid. Clearly, decoding a set of shadows is much less efficient than generating a set of shadows. The worst case occurs when the block of shadows is not associated with any key; all $3v\binom{4v}{3}/4$ must be generated and checked, thus

taking $O(v^4)$ time.

3.4. Generating Needed Input Files

In our discussions above, we simply assumed that the necessary input files were available; unfortunately, that is not the case. They must be generated also. In order to generate them, an implementation of Lindner's [L2] $2v$ construction has been developed and may be repeatedly applied to any of the quadruple systems listed in Appendix B. To obtain other quadruple systems with orders that are not even multiples of the order of one of the quadruples listed in Appendix B, see Lindner and Rosa [LR] for the necessary construction.

Generating the other file is somewhat easier. A UNIX† Korn shell script has been provided to create it. The user simply enters the name of the scheme for which the file is needed and the order n. The shell–script first calls a program to generate an n–cycle and place it in the file, and then calls the program containing implementations of all the N_2–latin square constructions mentioned in Section 2.1.3, appending the square it generates onto the file. Since squares of some orders can be generated by more than one of the constructions cited, it should be mentioned that the construction given in [KT] is only used for orders $n = 2^h$, $h \not\equiv 3$ *modulo* 4, $h > 3$; and, the direct product construction given in [M] is therefore used only when $n = 2^h$ and $h \equiv 3$ *modulo* 4, $h > 3$. This situation arises precisely when $2^h \equiv 3$ *modulo* 5.

† UNIX is a Trademark of Bell Laboratories.

4. Concluding Remarks

Having successfully developed a software system which implements $(4,4,4v)$–threshold schemes, an important question that must be asked is: *Is the system of practical use, and what are its limitations?* Unfortunately, in this case the limitations are such that the system described in section 3, with a large enough number of keys to be of practical use, is impossible on most current hardware configurations. We shall first look at exactly what the limitations are and then at possible improvements to the system.

4.1. Limitations

Often when trying to implement algorithms to solve combinatorial problems, the limiting factor is time. The machine might run for days, months or even years without finding a solution. In our system the decoding operation is the most expensive, taking worst-case time $O(v^4)$. While this may present a problem for large enough values of v, the speed of the machine is of secondary concern. Space is a much greater concern. On a typical system available to a user of this system, both external storage and primary memory capacities will limit the value of v long before the machine's speed does. The two input files—the file containing the $S(3,4,v)$ system and the file containing the v–cycle and the N_2–latin square—are the reason for this.

Consider first of all the file containing the $S(3,4,v)$ system. It contains $\binom{v}{3}/4$ quadruples each taking $4(1+\lfloor 1+\log_{10} v \rfloor)+c_1+c_2$ bytes, where c_1 is the number of bytes used to store the newline character and c_2 is the amount of file overhead.

Assuming a value of $c_1 = 1$ and ignoring c_2, some exact storage requirements are given in Table III; but, in general, the file requires $O(v^3)$ bytes of external storage.

TABLE III

Number of points (v)	Number of keys ($3v$)	N_2-latin square memory requirements (bytes)	$S(3,4,v)$ system storage requirements (bytes)
8	24	256	126
16	48	1024	1820
32	96	4096	16120
64	192	16384	135408
128	384	65536	1450848
256	768	262144	11744960
512	1536	1048576	94514560
1024	3072	4194304	936773376
2048	6144	16777216	7505186304
4096	12288	67108864	60085509120

The file containing the v–cycle and N_2–latin square does not present such a problem in terms of external storage—it requires $O(v^2)$ bytes—generating it is another matter. As we saw earlier, it was necessary to implement several different constructions for N_2–latin squares, with the value of v determining which one to use. These constructions are very different from each other and tend not to fill in entries in any sort of localized manner—one row at a time, for example. Consequently, it seems prudent, for efficiency of time, to store the entire square in internal memory while it is being constructed. Unfortunately, $O(v^2)$ bytes ($4v^2$, to be exact) of internal memory does present a major problem. Table III also presents the memory requirements for the selected values of v. As can be seen from looking at Table III, a typical machine will run out of internal space at about the same time it runs out of external storage—approximately when $256 \leq v \leq 512$. A system with

a value of v in this range certainly does not have enough keys to discourage someone from checking all of them.

4.2. Improvements

Clearly it would be useful to find a way to improve the software to allow machines of average size and speed to handle $(4,4,4v)$–threshold schemes with a reasonably large number of keys. The most obvious place for improvement is in the amount of space required to store an $S(3,4,v)$ system. At first glance, one possibility would be to store the numbers (points) as integers, rather than in ASCII format as we have done; however, no savings is realized until $v \geq 1000$, since each integer requires four bytes of storage. Even then the savings would only be by some constant multiplier, not by an order of magnitude. What we really need is some novel method of storing these quadruple systems that will require no more than $O(v^2)$ bytes of storage.

Another way in which this system could be improved is perhaps more feasible, though certainly not trivial. The problem of insufficient memory space that we encounter when trying to construct a reasonably large N_2–latin square could be avoided by developing a paging scheme. This scheme would keep only a small portion of the square being generated in memory at any one time; and whenever a section of the square outside of that currently in memory were referenced, then the section in memory would be swapped with the appropriate section from storage. The difficulty is, as we alluded to earlier, is that most of the various constructions "jump" all around in terms of the entries being accessed. To make this idea efficient enough to be worthwhile would require redesigning the constructions so that

they reference entries of the square being generated in very localized patterns.

Finally, if we were able to accomplish the above improvements, then the speed of the decoding segment of the system would become of concern. Unfortunately, making improvements here appears to be very difficult. There seem to be too many variables involved to allow for a direct decoding algorithm. Perhaps, however, some more clever representation of the points would allow this. Of course, such a representation might also require more storage space, which would be counterproductive.

We now come back to the first part of our question: *Is the system of practical use?* and we answer affirmatively, even without making any of the above improvements. Consider a typical combination lock having only 36 different keys. It is a simple matter to check all of them, but by requiring a sequence of three of those 36 keys to open the lock, checking all 36^3 possibilities is prohibitive. Similarly, we can generate two or more random blocks of shadows from a scheme having say 768 keys ($v = 256$) and give each participant at most one shadow from each block. Dictating that the blocks of shadows be decoded in some particular order would be difficult, but also unnecessary. Even without stipulating any order, we would have $\binom{768}{2}$ different possibilities. In this way we believe that the system developed in this thesis can have practical use.

References

[B] G. R. Blakely, Safeguarding Cryptographic Keys, *Proc. N. C. C., vol 48,* AFIPS Conference Proceedings 48 (1979), 313-317.

[BJL] T. Beth, D. Jungnickel and H. Lenz, *Design Theory,* Cambridge University Press, Cambridge, 1986.

[D] R.H.F. Denniston, Remarks on Latin Squares with no Subsquares of Order Two, *Utilitas Mathematica,* 13 (1978), 299-302.

[H] H. Hanani, On Quadruple Systems, *Canadian Journal of Mathematics,* 12 (1960), 145-157.

[K] S.C. Kothari, Generalized Linear Threshold Scheme *Lecture Notes in Computer Science,* 196 (1985), 231-241.

[KLR] A. Kotzig, C.C. Lindner, and A. Rosa, Latin Squares with no Subsquares of Order Two and Disjoint Steiner Triple Systems, *Utilitas Mathematica,* 7 (1975), 287-294.

[KT] A. Kotzig and J. Turgeon, On Certain Constructions for Latin Squares with no Subsquares of Order Two, *Discrete Mathematics,* 16 (1978), 263-270.

[L1] C.C. Lindner, A Note on Disjoint Steiner Quadruple Systems, *Ars Combinatoria,* 3 (1977), 271-276.

[L2] C.C. Lindner, On the Construction of Pairwise Disjoint Steiner Quadruple Systems, *Ars Combinatoria,* 19 (1985), 153-156.

[LR] C.C. Lindner and A. Rosa, Steiner Quadruple Systems--A Survey, *Discrete Mathematics,* 22 (1978), 147-181.

[M] M. McLeish, On the Existence of Latin Squares with no Subsquares of Order Two, *Utilitas Mathematica,* 8 (1975), 41-53.

[Sh] A. Shamir, How to Share a Secret, *Communications of the ACM,* 22 (1979), 612-613.

[St] D.R. Stinson, Threshold Schemes from Combinatorial Designs, *preprint,* (1988).

[SV] D.R. Stinson and S.A. Vanstone, A Combinatorial Approach to Threshold Schemes, *SIAM Journal on Discrete Mathematics,* 1 (1988), 230-236.

[T] L. Tierlinck, On Large Sets of Disjoint Quadruple Systems, *Ars Combinatoria,* 17 (1984), 173-176.

[W] R.M. Wilson, Some Partitions of all Triples into Steiner Triple Systems, *Lecture Notes in Mathematics,* 411 (1974), 267-277.

APPENDIX A

Main Algorithms

In order to present the similarities and differences between the ***cons4v***, ***shadows*** and ***decode*** programs more concisely than the source code does, the pseudocoded algorithms for the main() section of each of these programs is given below.

Algorithm CONS4V

Global variables: v

Let $v \equiv 2,4 \; modulo \; 6,\; v > 4$.
Let L be an N_2–latin square of order v.
Let ϕ be a v–cycle on $\mathbb{Z}_v$.
Let $\pi_0 = \{(0,1),(2,3)\}$, $\pi_1 = \{(0,2),(1,3)\}$, $\pi_2 = \{(0,3),(1,2)\}$.

for $i \leftarrow 1$ to v do
    $\beta_0 \leftarrow \phi^i$
    $\beta_1 \leftarrow \phi^{i+2}$
    $\beta_2 \leftarrow \phi^{i+1}$
    $\alpha \leftarrow L_i$
    for $j \leftarrow 0$ to 3 do
        $OF \leftarrow$ "open $(3i+j)^{th}$ output file"
        if $j < 2$
            CONS4V1_01(OF, π_j, α, α^{-1})
        else
            CONS4V1_2(OF, α, α^{-1})
        fi
        CONS4V2(OF, π_j, β_0, β_j)
        CONS4V3(OF, π_j, β_0, β_j)
        CONS4V4(OF, π_j, β_0, β_j)
    end
end.

Algorithm SHADOWS

Global variables: v, b, key, blk

Let $v \equiv 2,4 \; modulo \; 6, \; v > 4.$

Let $b = \binom{v}{3}/4, \; blks1 = 2\binom{2v}{3}/4, \; blks2 = 8\binom{v}{2}, \; blks1 = 48\binom{v}{3}/4.$
Let $0 \leq key < 3v$.
Let L be an N_2–latin square of order v.
Let ϕ be a v–cycle on $\mathbb{Z}_v$.
Let $\pi_0 = \{(0,1),(2,3)\}, \; \pi_1 = \{(0,2),(1,3)\}, \; \pi_2 = \{(0,3),(1,2)\}$.

$i \leftarrow key/3 + 1$
$j \leftarrow key\%3$

$blk \leftarrow 0 < RANDOM() \leq \binom{4v}{3}/4$

$\beta_0 \leftarrow \phi^i$
$\beta_1 \leftarrow \phi^{i+2}$
$\beta_2 \leftarrow \phi^{i+1}$
$\alpha \leftarrow L_i$

```
if blk ≤ blks1
      if j < 2
            SHADOWS1_01(OF, π_j, α, α^-1)
      else
            SHADOWS1_2(OF, α, α^-1)
      fi
else if blk ≤ blks2
      blk ← blk − blks1
      SHADOWS2(OF, π_j, β_0, β_j)
else if blk ≤ blks3
      blk ← blk − blks2
      SHADOWS3(OF, π_j, β_0, β_j)
else
      blk ← blk − blks3
      SHADOWS4(OF, π_j, β_0, β_j)
fi.
```

Algorithm DECODE

Global variables: v, B

Let $v \equiv 2,4 \; modulo \; 6$, $v > 4$.
Let L be an N_2–latin square of order v.
Let B be the block of shadows to decode.
Let ϕ be a v–cycle on $\mathbb{Z}_v$.
Let $\pi_0 = \{(0,1),(2,3)\}$, $\pi_1 = \{(0,2),(1,3)\}$, $\pi_2 = \{(0,3),(1,2)\}$.

```
B ← INTERACTIVE_INPUT_QUAD(4v)
for i ← 1 to v do
      β0 ← φ^i
      β1 ← φ^(i+2)
      β2 ← φ^(i+1)
      α ← L_i
      for j ← 0 to 3 do
            key ← 3i + j
            if j < 2
                  DECODE1_01(OF, π_j, α, α^-1)
            else
                  DECODE1_2(OF, α, α^-1)
            fi
            DECODE2(OF, π_j, β0, β_j)
            DECODE3(OF, π_j, β0, β_j)
            DECODE4(OF, π_j, β0, β_j)
      end
end
print "Shadows match no key!"
```

APPENDIX B

Source Code

All the source code for this $(4,4,4v)$–threshold scheme implementation was written in the C programming language under the UNIX† operating system running on the AT&T 3B series of computers. This appendix contains all the source code for the following executables:

shadows, decode, N2cons, checkv, and *cycle.*

Also included is the Korn shell script *make_pfile* that combines the two programs, *N2cons* and *cycle,* into one command that can be used to generate the permutations file needed by the *shadows* and *decode* programs. The source files included are listed in the following order:

globals.c
globals.h
aux.c
shadows.c
shadows1_12.c
shadows1_3.c
shadows2.c
shadows3.c
shadows4.c
decode.c
decode1_12.c
decode1_3.c
decode2.c
decode3.c
decode4.c
N2cons.c
cycle.c
checkv.c
make_pfile.ksh

† UNIX is a Trademark of Bell Laboratories.

```c
/************************************************************************/
/*                                                                      */
/*  globals.c                                                           */
/*                                                                      */
/*      This file contains the global variable declarations and         */
/*      include files for the cons4v, shadows, and decode programs.     */
/*                                                                      */
/*      Author: W. John Monroe          Last Updated: 21 July 1989      */
/*      Institution: Rochester Institute of Technology                  */
/************************************************************************/

# include <ctype.h>
# include <stdio.h>
# include <string.h>

unsigned  v,            /* order of input quadruple system */
          vv,           /* # of points in threshold scheme (= 4*v) */
          b,            /* # blocks in SQS(v) */
          bb,           /* # blocks in SQS(vv) */
          key,          /* key value for which shadows are being generated */
          words,        /* v/WORDSIZE */
          sblk,         /* randomly picked block of shadows to use */
         *keyblk;       /* block of shadows to decode */

FILE     *OF, *PF, *QF; /* file pointers to output file, permutations */
                        /*    file, and quadruples file, respectively */

char     *calloc();
double    drand48();
void      srand48(), exit();
long      time();
```

```c
/************************************************************************/
/*                                                                      */
/*  globals.h                                                           */
/*                                                                      */
/*      This header file contains the external declarations for the     */
/*      global variables, defined constants, and the include files      */
/*      for the cons4v, shadows, and decode programs.                   */
/*                                                                      */
/*      Author: W. John Monroe          Last Updated: 21 July 1989      */
/*      Institution: Rochester Institute of Technology                  */
/************************************************************************/

# include <ctype.h>
# include <stdio.h>
# include <string.h>

# define WORDSIZE 32            /* number of bits in an int */
# define NAMELEN  22            /* max length of a file name */
# define PFIXLEN  15            /* max length of user-specified */
                                /*      part of file name      */

extern unsigned  v,             /* order of input quadruple system */
                 vv,            /* # of points in threshold scheme (= 4*v) */
                 b,             /* # blocks in SQS(v) */
                 bb,            /* # blocks in SQS(vv) */
                 key,           /* key value of shadows being generated */
                 words,         /* v/WORDSIZE */
                 sblk,          /* randomly picked block of shadows to use */
                *keyblk;        /* block of shadows to decode */

extern FILE     *OF, *PF, *QF;  /* file pointers to output file, permutations */
                                /*    file, and quadruples file, respectively */

extern char     *calloc();
extern double    drand48();
extern void      srand48(), exit();
extern long      time();
```

```c
/***********************************************************************/
/*                                                                     */
/*  aux.c                                                              */
/*                                                                     */
/*      This file contains auxiliary functions used by the             */
/*      cons4v, shadows, and decode programs.                          */
/*                                                                     */
/*      Author: W. John Monroe          Last Updated: 20 July 1989     */
/*      Institution: Rochester Institute of Technology                 */
/***********************************************************************/

# include "globals.h"


/****************************************************************/
/*  insert()                                                    */
/*                                                              */
/*      Inserts element x into bit-set A.                       */
/*      The bit-set may be more than one word long.             */

void insert(A, x)
  unsigned *A, x;
{
   A[x/WORDSIZE] |= 01 << x%WORDSIZE;
   return;
}

/****************************************************************/
/*  b_or()                                                      */
/*                                                              */
/*      Set C is assigned the union of sets A and B,            */
/*        i.e. bitwise OR of multiple word integers.            */
/*      Global variable:  words                                 */

void b_or(A, B, C)
  unsigned *A, *B, *C;
{
   unsigned i;
   for (i=0; i < words; i++)
      C[i] = A[i] | B[i];
   return;
}

/****************************************************************/
/*  b_and()                                                     */
/*                                                              */
/*      Set C gets the intersection of sets A and B,            */
/*        i.e. bitwise AND of multiple word integers            */
/*      Global variable:  words                                 */

void b_and(A, B, C)
  unsigned *A, *B, *C;
{
   unsigned i;
   for (i=0; i < words; i++)
      C[i] = A[i] & B[i];
   return;
}

/****************************************************************/
/*  compare()                                                   */
/*                                                              */
/*      Check to see if the quadruple sets A and B equal--      */
/*      if so return zero, if not return non-zero.              */
/*      Global variable:  words                                 */

int  compare(A, B)
 unsigned *A, *B;
{
   int i, diff = 0;
   for (i=0; (i < words) && (!diff); i++)
      diff = A[i] ^ B[i];
   return (diff);
}

/****************************************************************/
/*  readrow()                                                   */
/*                                                              */
/*      Reads one row of a file of non-negative integers        */
/*      into an array and returns 0 if successful and 1         */
/*      if end-of-file occurs before specified number of        */
/*      values are read in.                                     */

int  readrow(F, A, cols)
 FILE *F;                /* file from which to read elements */
 unsigned *A,            /* array to store elements in       */
          cols;          /* number of elements in the row    */
{
   unsigned i;
   for (i=0; i < cols; i++)
   {
      (void) fscanf(F, "%d", &A[i]);
      if (feof(F))
      {
         if (i != 0)
            (void) fprintf(stderr, "END OF FILE reached unexpectedly\n");
         return(1);
      }
   }
   return(0);
}

/****************************************************************/
/*  printquad()                                                 */
/*                                                              */
/*      Print out elements of the quadruple set A on file FP.   */
/*      Global variable:  v, words                              */

void printquad(FP, A)
 FILE *FP;
 unsigned *A;
{
   unsigned  i, k, m;
   for (i=0; i < words; i++)
      for (m=A[i], k=i*WORDSIZE; m != 0; m>>=1, k++)
         if (m & 01)
            (void) fprintf(FP, ' %3u", k);
   (void) fprintf(FP, "\n");
}
```

```c
/****************************************************************/
/*  lshift()                                                    */
/*                                                              */
/*      Shift all bits of set A left by n, shifting 0s in       */
/*      on right end.   Handles multiple word bit-sets.         */
/*      Global variable:  words                                 */

void lshift(A, n)
  unsigned *A, n;
{
   unsigned i, shoveamt, shftamt, invshft;
   shoveamt = n/WORDSIZE;
   if (shftamt = n%WORDSIZE)    /* skip if shifting by          */
   {                            /*         multiple of wordsize */
      invshft = WORDSIZE - shftamt;
      for (i=words-(1+shoveamt); i > (unsigned)0; i--)
      {
         A[i] <<= shftamt;
         A[i] |= A[i-1] >> invshft;
      }
      A[0] <<= shftamt;
   }
   if (shoveamt)                /* n >= length of one word */
   {
      for (i=words-1; i >= shoveamt; i--)
         A[i] = A[i-shoveamt];
      for (; i > (unsigned)0; i--)
         A[i] = 0;
      A[0] = 0;
   }
   return;
}

/****************************************************************/
/*  rshift()                                                    */
/*                                                              */
/*      Shift all bits of set A right by n, shifting 0s in      */
/*      on left end.    Handles multiple word bit-sets.         */
/*      Global variable:  words                                 */

void rshift(A, n)
  unsigned *A, n;
{
   unsigned i, shoveamt, shftamt, invshft;
   shoveamt = n/WORDSIZE;
   if (shftamt = n%WORDSIZE)    /* skip if shifting by          */
   {                            /*         multiple of wordsize */
      invshft = WORDSIZE - shftamt;
      for (i=shoveamt; i < words-1; i++)
      {
         A[i] >>= shftamt;
         A[i] |= A[i+1] << invshft;
      }
      A[words-1] >>= shftamt;
   }
   if (shoveamt)                /* n >= length of one word */
   {
      for (i=0; i < words-shoveamt; i++)
         A[i] = A[i+shoveamt];
      for (; i < words; i++)
         A[i] = 0;
   }
   return;
}
```

```
/************************************************************************/
/*                                                                      */
/*  shadows.c                                                           */
/*                                                                      */
/*      This program takes the name of a threshold scheme as input      */
/*      and, assuming the appropriate quadruples file and permutations  */
/*      file exist, produces a random block of four shadows in the      */
/*      range [0,4v) that are associated with the specified key.  The   */
/*      quadruples file must contain an S(3,4,v) system and the         */
/*      permutations file must contain a v-cycle followed by an         */
/*      N2-latin square of order v.  Subroutines are in separate        */
/*      files: shadows1_12.c, shadows1_3.c, shadows2.c, shadows3.c, and */
/*      shadows4.c.  See also cons4v.c and decode.c.                    */
/*                                                                      */
/*      Author: W. John Monroe          Last Updated: 24 July 1989      */
/*      Institution: Rochester Institute of Technology                  */
/************************************************************************/

# include "globals.h"

main(ac,av)
  int ac;
  char *av[];
{
   unsigned i, j, k, q, blks1, blks2, blks3,
            *junk, *perm, *iperm, *cycle, *pos, *alpha[3];
   static unsigned PI[3][4] = { {0, 1, 2, 3},
                                {0, 2, 1, 3},
                                {0, 3, 1, 2} };
   char     c, qfile[NAMELEN], pfile[NAMELEN];

   if (ac != 4)
   {
      (void) fprintf(stderr, "Usage: %s schemename 4v key\n", av[0]);
      exit(1);
   }
   vv = atoi(av[2]);
   if ((vv <= 8) || (vv%24 != 8  &&  vv%24 != 16))
   {
      (void) fprintf(stderr, "ERROR: %d is not a valid value for 4v\n", vv);
      exit(1);
   }
   key = atoi(av[3]);
   if (key >= 3*vv/4)
   {
      (void) fprintf(stderr, "ERROR: %d is too large a key for scheme\n", key);
      exit(1);
   }

   v = vv/4;
   b = (v*(v-1)*(v-2))/24;

   (void) strcpy(qfile, av[1]);
   qfile[PFIXLEN] = '\0';           /* truncate scheme name if necessary */
   (void) strcpy(pfile, qfile);
   if ( !(QF = fopen(strcat(qfile, '.quads\0"), "r")))
   {
      (void) fprintf(stderr, "ERROR: cannot open %s\n", qfile);
      exit(1);
   }
   if ( !(PF = fopen(strcat(pfile, ".perms\0"), "r")))
   {
      (void) fprintf(stderr, "ERROR: cannot open %s\n", pfile);
      exit(1);
   }

   words = vv/WORDSIZE + 1;
   junk  = (unsigned *) calloc(4, sizeof(unsigned));
   perm  = (unsigned *) calloc(v, sizeof(unsigned));
   iperm = (unsigned *) calloc(v, sizeof(unsigned));
   cycle = (unsigned *) calloc(v, sizeof(unsigned));
   pos   = (unsigned *) calloc(v, sizeof(unsigned));
   for (i=0; i < 3; i++)
      alpha[i] = (unsigned *) calloc(v, sizeof(unsigned));

   if (readrow(PF, cycle, v))
   {
      (void)fprintf(stderr, "ERROR: end of permfile reached unexpectedly\n");
      exit(1);
   }
   for (k=0; k < v; k++)
      pos[cycle[k]] = k;

   i = key/3 + 1;
   j = key%3;

   for (k=0; k < v; k++)
   {
      alpha[0][k] = cycle[(i   + pos[k]) % v];
      alpha[1][k] = cycle[(i+2 + pos[k]) % v];
      alpha[2][k] = cycle[(i+1 + pos[k]) % v];
   }

   for (k=1; k < i; k++)                /* skip to correct permutation */
      if (readrow(PF, junk, v))
         {
            (void) fprintf(stderr, "ERROR: end of permfile reached ");
            (void) fprintf(stderr, "unexpectedly\n");
            exit(1);
         }

   if (readrow(PF, perm, v))            /* read in correct (ith) permutation */
   {
      (void)fprintf(stderr, "ERROR: end of permfile reached unexpectedly\n");
      exit(1);
   }
   for (k=0; k < v; k++)
      iperm[perm[k]] = k;
```

```
    bb = (vv*(vv-1)*(vv-2))/24;
    srand48(time(0));
    sblk = ((int) (drand48() * bb)) % bb + 1;
    blks1 = 16*b + v*(v-1);
    blks2 = 4*v*(v-1) + blks1;
    blks3 = 48*b + blks2;

    OF = stdout;
    if (sblk <= blks1)
    {
       if (j<2)
          shadows1_12(OF, PI[j], perm, iperm);
       else
          shadows1_3(OF, perm, iperm);
    }
    else if (sblk <= blks2)
    {
       sblk -= blks1;
       shadows2(OF, PI[j], alpha[0], alpha[j]);
    }
    else if (sblk <= blks3)
    {
       sblk -= blks2;
       shadows3(OF, PI[j], alpha[0], alpha[j]);
    }
    else
    {
       sblk -= blks3;
       shadows4(OF, PI[j], alpha[0], alpha[j]);
    }

    (void) fclose(QF);
    (void) fclose(PF);
}
```

```
/**********************************************************************/
/*                                                                    */
/*  shadows1_12.c                                                     */
/*                                                                    */
/*      Used to generate any block of shadows that would be           */
/*      constructed by part 1 of Lindner's 4v construction when       */
/*      pi(1) and pi(2) are being used.                               */
/*      See also cons4v1_12.c and decode1_12.c.                       */
/*                                                                    */
/*      Author: W. John Monroe          Last Updated: 24 July 1989    */
/*      Institution: Rochester Institute of Technology                */
/**********************************************************************/

# include "globals.h"

void shadows1_12(FP, PI, perm, iperm)
  FILE *FP;
  unsigned PI[], *perm, *iperm;
{
   unsigned i, j, k, sblk1, quad[4], junk[4],
            *tmp1, *tmp2, *tmp3, *newq;

   tmp1 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp2 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp3 = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));


   if (sblk <= 16*b)
   {                                    /* part (1) of 2v construction */
      i = (sblk-1)/16;
      k = 4 - ((sblk-1)%16)/4;

      for (j=0; j < i; j++)
         if (readrow(QF, junk, 4))
         {
            (void) fprintf(stderr, "ERROR: ");
            (void) fprintf(stderr, "end of quadfile reached unexpectedly\n");
            exit(1);
         }
      if (readrow(QF, quad, 4))
      {
         (void)fprintf(stderr, "ERROR: end of quadfile reached unexpectedly\n");
         exit(1);
      }

      for (j=0; j < words; j++)
         tmp1[j] = tmp2[j] = tmp3[j] = 0;

      for (j=0; j < 4; j++)
      {
         if (j==(k-1))
         {
            insert(tmp2, perm[quad[j]] + PI[1]*v);
            insert(tmp3, iperm[quad[j]]);
         }
         else
            insert(tmp1, quad[j]);
      }
      switch ((sblk-1)%4)
      {
         case 0:  b_or(tmp1, tmp2, newq);
                 break;
         case 1:  b_or(tmp1, tmp2, newq);
                  lshift(newq, (PI[2])*v);
                 break;
         case 2:  lshift(tmp1, PI[1]*v);
                  b_or(tmp1, tmp3, newq);
                 break;
         case 3:  lshift(tmp1, PI[1]*v);
                  b_or(tmp1, tmp3, newq);
                  lshift(newq, (PI[2])*v);
                 break;
      }
      printquad(FP, newq);
   }

   else
   {                                    /* part (2) of 2v construction */
      sblk -= 16*b;
      for (i=0, sblk1=sblk; (sblk1-1) >= 2*(v-1-i); i++)
         sblk1 -= 2*(v-1-i);
      j = i+1 + (sblk1-1)/2;
      for (k=0; k < words; k++)
         newq[k] = 0;
      insert(newq, i);
      insert(newq, j);
      insert(newq, perm[i] + PI[1]*v);
      insert(newq, perm[j] + PI[1]*v);

      if ((sblk-1) % 2)
         lshift(newq, (PI[2])*v);
      printquad(FP, newq);
   }
}
```

```c
/**********************************************************************/
/*                                                                    */
/*  shadows1_3.c                                                      */
/*                                                                    */
/*      Used to generate any block of shadows that would be          */
/*      constructed by part 1 of Lindner's 4v construction when      */
/*      pi(3) is being used.  See also cons4v1_3.c and decode1_3.c.  */
/*                                                                    */
/*      Author: W. John Monroe          Last Updated: 24 July 1989   */
/*      Institution: Rochester Institute of Technology               */
/**********************************************************************/

# include "globals.h"

void shadows1_3(FP, perm, iperm)
  FILE *FP;
  unsigned *perm, *iperm;
{
   unsigned i, j, k, sblk1, quad[4], junk[4],
            *tmp1, *tmp2, *tmp3, *newq;

   tmp1 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp2 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp3 = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));

   if (sblk <= 16*b)
   {                                    /* part (1) of 2v construction */
      i = (sblk-1)/16;
      k = 4 - ((sblk-1)%16)/4;

      for (j=0; j < i; j++)
         if (readrow(QF, junk, 4))
         {
            (void) fprintf(stderr, "ERROR: ");
            (void) fprintf(stderr, "end of quadfile reached unexpectedly\n");
            exit(1);
         }
      if (readrow(QF, quad, 4))
      {
         (void)fprintf(stderr, "ERROR: end of quadfile reached unexpectedly\n");
         exit(1);
      }

      for (j=0; j < words; j++)
         tmp1[j] = tmp2[j] = tmp3[j] = 0;

      for (j=0; j < 4; j++)
      {
         if (j==(k-1))
         {
            insert(tmp2, perm[quad[j]] + 3*v);
            insert(tmp3, iperm[quad[j]]);
         }
         else
            insert(tmp1, quad[j]);
      }
      switch ((sblk-1)%4)
      {
         case 0:  b_or(tmp1, tmp2, newq);
                  break;
         case 1:  lshift(tmp1, v);
                  rshift(tmp2, v);
                  b_or(tmp1, tmp2, newq);
                  break;
         case 2:  lshift(tmp1, 3*v);
                  b_or(tmp1, tmp3, newq);
                  break;
         case 3:  lshift(tmp1, 2*v);
                  lshift(tmp3, v);
                  b_or(tmp1, tmp3, newq);
                  break;
      }
      printquad(FP, newq);
   }

   else
   {                                    /* part (2) of 2v construction */
      sblk -= 16*b;

      for (i=0, sblk1=sblk; (sblk1-1) >= 2*(v-1-i); i++)
         sblk1 -= 2*(v-1-i);
      j = i+1 + (sblk1-1)/2;

      for (k=0; k < words; k++)
         tmp1[k] = tmp2[k] = newq[k] = 0;
      insert(tmp1, i);
      insert(tmp1, j);
      insert(tmp2, perm[i] + 3*v);
      insert(tmp2, perm[j] + 3*v);

      if ((sblk-1) % 2)
      {
         lshift(tmp1, v);
         rshift(tmp2, v);
      }
      b_or(tmp1, tmp2, newq);
      printquad(FP, newq);
   }
}
```

```
/**************************************************************************/
/*                                                                        */
/*  shadows2.c                                                            */
/*                                                                        */
/*      Used to generate any block of shadows that would be               */
/*      constructed by part 2 of Lindner's 4v construction.               */
/*      See also cons4v2.c and decode2.c.                                 */
/*                                                                        */
/*      Author: W. John Monroe          Last Updated: 24 July 1989        */
/*      Institution: Rochester Institute of Technology                    */
/**************************************************************************/

# include "globals.h"

void shadows2(FP, PI, alpha, beta)
  FILE *FP;
  unsigned PI[], *alpha, *beta;
{
   unsigned i, j, x, y, sblk1,
            *X[4], *Y[4], *xa, *ya, *xb, *yb, *tmp[4], *newq;

   for (i=0; i < 4; i++)
   {
      X[i] = (unsigned *) calloc(words, sizeof(unsigned));
      Y[i] = (unsigned *) calloc(words, sizeof(unsigned));
      tmp[i] = (unsigned *) calloc(words, sizeof(unsigned));
   }
   xa = (unsigned *) calloc(words, sizeof(unsigned));
   ya = (unsigned *) calloc(words, sizeof(unsigned));
   xb = (unsigned *) calloc(words, sizeof(unsigned));
   yb = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));

   for (x=0, sblk1=sblk; (sblk1-1) >= 8*(v-1-x); x++)
      sblk1 -= 8*(v-1-x);

   y = x+1 + (sblk1-1)/8;
   for (i=0; i < words; i++)
   {
      xa[i] = ya[i] = xb[i] = yb[i] = 0;
      for (j=0; j < 4; j++)
         X[j][i] = Y[j][i] = 0;
   }
   for (i=0; i < 4; i++)
   {
      insert(X[i], x + v*PI[i]);
      insert(Y[i], y + v*PI[i]);
   }
   insert(xa, alpha[x] + v*PI[1]);
   insert(ya, alpha[y] + v*PI[1]);
   insert(xb, beta[x] + v*PI[3]);
   insert(yb, beta[y] + v*PI[3]);

   if (!(((sblk-1)%8)/4))
   {
      b_or(X[0], Y[0], tmp[0]);
      b_or(X[1], Y[1], tmp[1]);
      b_or(X[2], yb, tmp[2]);
      b_or(Y[2], xb, tmp[3]);
   }
   else
   {
      b_or(X[2], Y[2], tmp[0]);
      b_or(X[3], Y[3], tmp[1]);
      b_or(X[0], ya, tmp[2]);
      b_or(Y[0], xa, tmp[3]);
   }

   i = (((sblk-1)%8)%4)/2 + 2;
   j = (((sblk-1)%8)%4)%2;

   b_or(tmp[j], tmp[i], newq);
   printquad(FP, newq);
}
```

```c
/**********************************************************************/
/*                                                                    */
/*  shadows3.c                                                        */
/*                                                                    */
/*      Used to generate any block of shadows that would be           */
/*      constructed by part 3 of Lindner's 4v construction.           */
/*      See also cons4v3.c and decode3.c.                             */
/*                                                                    */
/*      Author: W. John Monroe          Last Updated: 24 July 1989    */
/*      Institution: Rochester Institute of Technology                */
/**********************************************************************/

# include "globals.h"

void shadows3(FP, PI, alpha, beta)
  FILE *FP;
  unsigned PI[],  *alpha, *beta;
{
   unsigned A, B, i, j, k, l, q, x, y, quad[4], junk[4],
         *X[4], *Y[4], *ai, *bi, *as, *bs, *aa, *ba, *ab, *bb, *tmp[4], *newq;
   for (i=0; i < 4; i++)
   {
      X[i] = (unsigned *) calloc(words, sizeof(unsigned));
      Y[i] = (unsigned *) calloc(words, sizeof(unsigned));
      tmp[i] = (unsigned *) calloc(words, sizeof(unsigned));
   }
   ai = (unsigned *) calloc(words, sizeof(unsigned));
   bi = (unsigned *) calloc(words, sizeof(unsigned));
   as = (unsigned *) calloc(words, sizeof(unsigned));
   bs = (unsigned *) calloc(words, sizeof(unsigned));
   aa = (unsigned *) calloc(words, sizeof(unsigned));
   ba = (unsigned *) calloc(words, sizeof(unsigned));
   ab = (unsigned *) calloc(words, sizeof(unsigned));
   bb = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));

   q = (sblk-1)/48;
   for (j=0; j < q; j++)
      if (readrow(QF, junk, 4))
      {
         (void) fprintf(stderr, "ERROR: ");
         (void) fprintf(stderr, "end of quadfile reached unexpectedly\n");
         exit(1);
      }
   if (readrow(QF, quad, 4))
   {
      (void)fprintf(stderr, "ERROR: end of quadfile reached unexpectedly\n");
      exit(1);
   }
   switch (((sblk-1)%48)/8)
   {
      case 0:  l = 1; k = 0;
               break;
      case 1:  l = 2; k = 0;
               break;
      case 2:  l = 3; k = 0;
               break;
      case 3:  l = 2; k = 1;
               break;
      case 4:  l = 3; k = 1;
               break;
      case 5:  l = 3; k = 2;
               break;
   }
   x = quad[k];
   y = quad[l];
   for (i=0; i==k || i==l; i++)
      ;
   A = quad[i];
   for (++i; i==k || i==l; i++)
      ;
   B = quad[i];

   for (i=0; i < words; i++)
   {
      aa[i] = ba[i] = ab[i] = bb[i] = 0;
      as[i] = ai[i] = bs[i] = bi[i] = 0;
      for (j=0; j < 4; j++)
         X[j][i] = Y[j][i] = 0;
   }

   for (i=0; i < 4; i++)
   {
      insert(X[i], x + v*PI[i]);
      insert(Y[i], y + v*PI[i]);
   }
   insert(ai, A);
   insert(bi, B);
   insert(as, A + v*PI[2]);
   insert(bs, B + v*PI[2]);
   insert(aa, alpha[A] + v*PI[1]);
   insert(ba, alpha[B] + v*PI[1]);
   insert(ab, beta[A] + v*PI[3]);
   insert(bb, beta[B] + v*PI[3]);

   if (!(((sblk-1)%8)/4))
   {
      b_or(X[0], Y[0], tmp[0]);
      b_or(X[1], Y[1], tmp[1]);
      b_or(as, bb, tmp[2]);
      b_or(bs, ab, tmp[3]);
   }
   else
   {
      b_or(X[2], Y[2], tmp[0]);
      b_or(X[3], Y[3], tmp[1]);
      b_or(ai, ba, tmp[2]);
      b_or(bi, aa, tmp[3]);
   }

   i = (((sblk-1)%8)%4)/2 + 2;
   j = (((sblk-1)%8)%4)%2;

   b_or(tmp[j], tmp[i], newq);
   printquad(FP, newq);
}
```

```
/***************************************************************************/
/*                                                                         */
/*  shadows4.c                                                             */
/*                                                                         */
/*      Used to generate any block of sbadows that would be                */
/*      constructed by part 4 of Lindner's 4v construction.                */
/*      See also cons4v4.c and decode4.c.                                  */
/*                                                                         */
/*      Author: W. John Monroe          Last Updated: 24 July 1989         */
/*      Institution: Rochester Institute of Technology                     */
/***************************************************************************/

# include "globals.h"

void shadows4(FP, PI, alpha, beta)
  FILE *FP;
  unsigned PI[], *alpha, *beta;
{
   unsigned A, B, i, *newq;

   newq = (unsigned *) calloc(words, sizeof(unsigned));

   A = (sblk-1)/v;
   B = (sblk-1)%v;
   for (i=0; i < words; i++)
      newq[i] = 0;
   insert(newq, A);
   insert(newq, alpha[A] + v*PI[1]);
   insert(newq, B + v*PI[2]);
   insert(newq, beta[B] + v*PI[3]);
   printquad(FP, newq);
}
```

```c
/*********************************************************************/
/*                                                                   */
/*  decode.c                                                         */
/*                                                                   */
/*      This program takes the name of a threshold scheme as input   */
/*      and, assuming the appropriate quadruples file and permutations */
/*      file exist, prompts for four shadows to be entered.  It then  */
/*      returns the key, if any, that the four shadows are associated */
/*      with.  Subroutines are in separate files: decode1_12.c,      */
/*      decode1_3.c, decode2.c, decode3.c, and decode4.c.            */
/*      See also cons4v.c and shadows.c.                             */
/*                                                                   */
/*      Author: W. John Monroe          Last Updated: 24 July 1989   */
/*      Institution: Rochester Institute of Technology               */
/*********************************************************************/

# include "globals.h"

main(ac,av)
  int ac;
  char *av[];
{
   unsigned i, j, k, sh,
            *perm, *iperm, *cycle, *pos, *alpha[3];
   static unsigned PI[3][4] = { {0, 1, 2, 3},
                                {0, 2, 1, 3},
                                {0, 3, 1, 2} };
   char     ch, qfile[NAMELEN], pfile[NAMELEN];

   if (ac != 2)
   {
      (void) fprintf(stderr, "Usage: %s schemename\n", av[0]);
      exit(1);
   }

   (void) strcpy(qfile, av[1]);
   qfile[PFIXLEN] = '\0';          /* truncate scheme name if necessary */
   (void) strcpy(pfile, qfile);
   if (    !(QF = fopen(strcat(qfile, ".quads\0"), "r"))
        || !(PF = fopen(strcat(pfile, ".perms\0"), "r")) )
   {
      (void) fprintf(stderr, "ERROR: %s is not a valid scheme name\n", av[1]);
      exit(1);
   }
   (void) fclose(QF);
   v = 0;                          /* determine value of v */
   ch = getc(PF);
   while (ch != '\n')
   {
      if (isdigit(ch))
      {
         ++v;
         while (isdigit(ch = getc(PF)))
            ;
      }
      else
         ch = getc(PF);
   }
   (void) fclose(PF);
   PF = fopen(pfile, "r");
   vv = v * 4;
   b = v * (v-1) * (v-2) / 24;
   words = vv/WORDSIZE + 1;
   keyblk = (unsigned *) calloc(words, sizeof(unsigned));
   perm  = (unsigned *) calloc(v, sizeof(unsigned));
   iperm = (unsigned *) calloc(v, sizeof(unsigned));
   cycle = (unsigned *) calloc(v, sizeof(unsigned));
   pos   = (unsigned *) calloc(v, sizeof(unsigned));
   for (i=0; i < 3; i++)
      alpha[i] = (unsigned *) calloc(v, sizeof(unsigned));

   (void) printf("\nPlease enter one shadow at each prompt.\n");
   for (i=1; i <= 4; i++)
   {
      (void) printf("\tshadow %1d: ", i);
      (void) scanf("%d", &sh);
      if (sh > vv)
      {
         (void) printf("\nSorry, shadow too large for scheme.\n");
         exit(1);
      }
      else
         insert(keyblk, sh);
   }
   (void) printf("\n\n");

   if (readrow(PF, cycle, v))
   {
      (void)fprintf(stderr, "ERROR: end of permfile reached unexpectedly\n");
      exit(1);
   }
   for (k=0; k < v; k++)
      pos[cycle[k]] = k;
   OF = stdout;
   for (i=1; i <= v; i++)
   {
      for (k=0; k < v; k++)
      {
         alpha[0][k] = cycle[(i   + pos[k]) % v];
         alpha[1][k] = cycle[(i+2 + pos[k]) % v];
         alpha[2][k] = cycle[(i+1 + pos[k]) % v];
      }
      if (readrow(PF, perm, v))
      {
         (void)fprintf(stderr, "ERROR: end of permfile reached unexpectedly\n");
         exit(1);
      }
      for (k=0; k < v; k++)
         iperm[perm[k]] = k;
      for (j=0; j < 3; j++)
      {
         key = ((i-1)*3 + j);
         QF = fopen(qfile, "r");
         if (j<2)
            decode1_12(PI[j], perm, iperm);
         else
            decode1_3(perm, iperm);
         (void) fclose(QF);
         decode2(PI[j], alpha[0], alpha[j]);
         QF = fopen(qfile, "r");
         decode3(PI[j], alpha[0], alpha[j]);
         (void) fclose(QF);
         decode4(PI[j], alpha[0], alpha[j]);
      }
   }
   (void) printf("The set of shadows given does not represent any key.\n");
   (void) fclose(PF);
}
```

```
/**********************************************************************/
/*                                                                    */
/*  decode1_12.c                                                      */
/*                                                                    */
/*      Performs decoding version of part 1 of Lindner's 4v           */
/*      construction when pi(1) and pi(2) are being used.             */
/*      See also cons4v1_12.c and shadows1_12.c.                      */
/*                                                                    */
/*      Author: W. John Monroe          Last Updated: 24 July 1989    */
/*      Institution: Rochester Institute of Technology                */
/**********************************************************************/

# include "globals.h"

void decode1_12(PI, perm, iperm)
  unsigned PI[], *perm, *iperm;
{
   unsigned i, j, k, quad[4],
            *tmp1, *tmp2, *tmp3, *newq;

   tmp1 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp2 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp3 = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));

/* part (1) of 2v construction */
   for (i=0; i < b; i++)
   {
      if (readrow(QF, quad, 4))
      {
         (void)fprintf(stderr, "ERROR: end of quadfile reached unexpectedly\n");
         exit(1);
      }

      for (k=4; k > (unsigned)0; k--)
      {
         for (j=0; j < words; j++)
            tmp1[j] = tmp2[j] = tmp3[j] = 0;
         for (j=0; j < 4; j++)
         {
            if (j==(k-1))
            {
               insert(tmp2, perm[quad[j]] + PI[1]*v);
               insert(tmp3, iperm[quad[j]]);
            }
            else
               insert(tmp1, quad[j]);
         }

         b_or(tmp1, tmp2, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }

         lshift(newq, (PI[2])*v);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }

         lshift(tmp1, PI[1]*v);
         b_or(tmp1, tmp3, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }

         lshift(newq, (PI[2])*v);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }
      }
   }

/* part (2) of 2v construction */
   for (i=0; i < v-1; i++)
      for (j=i+1; j < v; j++)
      {
         for (k=0; k < words; k++)
            newq[k] = 0;
         insert(newq, i);
         insert(newq, j);
         insert(newq, perm[i] + PI[1]*v);
         insert(newq, perm[j] + PI[1]*v);

         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }

         lshift(newq, (PI[2])*v);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }
      }
}
```

```
/*************************************************************************/
/*                                                                       */
/*  decode1_3.c                                                          */
/*                                                                       */
/*      Performs decoding version of part 1 of Lindner's 4v              */
/*      construction when pi(3) is being used.                           */
/*      See also cons4v1_3.c and shadows1_3.c.                           */
/*                                                                       */
/*      Author: W. John Monroe          Last Updated: 24 July 1989       */
/*      Institution: Rochester Institute of Technology                   */
/*************************************************************************/

# include "globals.h"

void decode1_3(perm, iperm)
  unsigned *perm, *iperm;
{
   unsigned i, j, k, quad[4],
            *tmp1, *tmp2, *tmp3, *newq;

   tmp1 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp2 = (unsigned *) calloc(words, sizeof(unsigned));
   tmp3 = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));

/* part (1) of 2v construction */
   for (i=0; i < b; i++)
   {
      if (readrow(QF, quad, 4))
      {
         (void)fprintf(stderr, "ERROR: end of quadfile reached unexpectedly\n");
         exit(1);
      }

      for (k=4; k > (unsigned)0; k--)
      {
         for (j=0; j < words; j++)
            tmp1[j] = tmp2[j] = tmp3[j] = 0;
         for (j=0; j < 4; j++)
         {
            if (j==(k-1))
            {
               insert(tmp2, perm[quad[j]]+ 3*v);
               insert(tmp3, iperm[quad[j]]);
            }
            else
               insert(tmp1, quad[j]);
         }

         b_or(tmp1, tmp2, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key_value: %d\n", key);
            exit(1);
         }

         lshift(tmp1, v);
         rshift(tmp2, v);
         b_or(tmp1, tmp2, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }

         lshift(tmp1, 2*v);
         b_or(tmp1, tmp3, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }

         rshift(tmp1, v);
         lshift(tmp3, v);
         b_or(tmp1, tmp3, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }
      }
   }

/* part (2) of 2v construction */
   for (i=0; i < v-1; i++)
      for (j=i+1; j < v; j++)
      {
         for (k=0; k < words; k++)
            tmp1[k] = tmp2[k] = newq[k] = 0;
         insert(tmp1, i);
         insert(tmp1, j);
         insert(tmp2, perm[i] + 3*v);
         insert(tmp2, perm[j] + 3*v);

         b_or(tmp1, tmp2, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }

         lshift(tmp1, v);
         rshift(tmp2, v);
         b_or(tmp1, tmp2, newq);
         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }
      }
}
```

```c
/************************************************************************/
/*                                                                      */
/*  decode2.c                                                           */
/*                                                                      */
/*      Performs decoding version of part 2 of Lindner's 4v            */
/*      construction.  See also cons4v2.c and shadows2.c.              */
/*                                                                      */
/*      Author: W. John Monroe          Last Updated: 24 July 1989     */
/*      Institution: Rochester Institute of Technology                 */
/************************************************************************/

# include "globals.h"

void decode2(PI, alpha, beta)
  unsigned PI[], *alpha, *beta;
{
   unsigned i, j, x, y,
            *X[4], *Y[4], *xa, *ya, *xb, *yb, *tmp[4], *newq;

   for (i=0; i < 4; i++)
   {
      X[i] = (unsigned *) calloc(words, sizeof(unsigned));
      Y[i] = (unsigned *) calloc(words, sizeof(unsigned));
      tmp[i] = (unsigned *) calloc(words, sizeof(unsigned));
   }
   xa = (unsigned *) calloc(words, sizeof(unsigned));
   ya = (unsigned *) calloc(words, sizeof(unsigned));
   xb = (unsigned *) calloc(words, sizeof(unsigned));
   yb = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));

   for (x=0; x < v-1; x++)
      for (y=x+1; y < v; y++)
      {
         for (i=0; i < words; i++)
         {
            xa[i] = ya[i] = xb[i] = yb[i] = 0;
            for (j=0; j < 4; j++)
               X[j][i] = Y[j][i] = 0;
         }
         for (i=0; i < 4; i++)
         {
            insert(X[i], x + v*PI[i]);
            insert(Y[i], y + v*PI[i]);
         }
         insert(xa, alpha[x] + v*PI[1]);
         insert(ya, alpha[y] + v*PI[1]);
         insert(xb, beta[x] + v*PI[3]);
         insert(yb, beta[y] + v*PI[3]);

         b_or(X[0], Y[0], tmp[0]);
         b_or(X[1], Y[1], tmp[1]);
         b_or(X[2], yb, tmp[2]);
         b_or(Y[2], xb, tmp[3]);
         for (i=2; i <= 3; i++)
            for (j=0; j <= 1; j++)
            {
               b_or(tmp[j], tmp[i], newq);
               if (!compare(keyblk, newq))
               {
                  (void) printf("Key value: %d\n", key);
                  exit(1);
               }
            }

         b_or(X[2], Y[2], tmp[0]);
         b_or(X[3], Y[3], tmp[1]);
         b_or(X[0], ya, tmp[2]);
         b_or(Y[0], xa, tmp[3]);
         for (i=2; i <= 3; i++)
            for (j=0; j <= 1; j++)
            {
               b_or(tmp[j], tmp[i], newq);
               if (!compare(keyblk, newq))
               {
                  (void) printf("Key value: %d\n", key);
                  exit(1);
               }
            }
      }
}
```

```c
/***********************************************************************/
/*                                                                     */
/*  decode3.c                                                          */
/*                                                                     */
/*      Performs decoding version of part 3 of Lindner's 4v            */
/*      construction.  See also cons4v3.c and shadows3.c.              */
/*                                                                     */
/*      Author: W. John Monroe          Last Updated: 24 July 1989     */
/*      Institution: Rochester Institute of Technology                 */
/***********************************************************************/

# include "globals.h"

void decode3(PI, alpha, beta)
  unsigned PI[], *alpha, *beta;
{
   unsigned A, B, i, j, k, l, q, x, y, quad[4],
           *X[4], *Y[4], *ai, *bi, *as, *bs, *aa, *ba, *ab, *bb, *tmp[4], *newq;

   for (i=0; i < 4; i++)
   {
      X[i] = (unsigned *) calloc(words, sizeof(unsigned));
      Y[i] = (unsigned *) calloc(words, sizeof(unsigned));
      tmp[i] = (unsigned *) calloc(words, sizeof(unsigned));
   }
   ai = (unsigned *) calloc(words, sizeof(unsigned));
   bi = (unsigned *) calloc(words, sizeof(unsigned));
   as = (unsigned *) calloc(words, sizeof(unsigned));
   bs = (unsigned *) calloc(words, sizeof(unsigned));
   aa = (unsigned *) calloc(words, sizeof(unsigned));
   ba = (unsigned *) calloc(words, sizeof(unsigned));
   ab = (unsigned *) calloc(words, sizeof(unsigned));
   bb = (unsigned *) calloc(words, sizeof(unsigned));
   newq = (unsigned *) calloc(words, sizeof(unsigned));

   for (q=0; q < b; q++)
   {
      if (readrow(QF, quad, 4))
      {
         (void)fprintf(stderr, "ERROR: end of quadfile reached unexpectedly\n");
         exit(1);
      }
      for (k=0; k < 3; k++)
         for (l=k+1; l < 4; l++)
         {
            x = quad[k];
            y = quad[l];
            for (i=0; i==k || i==l; i++)
               ;
            A = quad[i];
            for (++i; i==k || i==l; i++)
               ;
            B = quad[i];
            for (i=0; i < words; i++)
            {
               aa[i] = ba[i] = ab[i] = bb[i] = 0;
               as[i] = ai[i] = bs[i] = bi[i] = 0;
               for (j=0; j < 4; j++)
                  X[j][i] = Y[j][i] = 0;
            }

            for (i=0; i < 4; i++)
            {
               insert(X[i], x + v*PI[i]);
               insert(Y[i], y + v*PI[i]);
            }
            insert(ai, A);
            insert(bi, B);
            insert(as, A + v*PI[2]);
            insert(bs, B + v*PI[2]);
            insert(aa, alpha[A] + v*PI[1]);
            insert(ba, alpha[B] + v*PI[1]);
            insert(ab, beta[A] + v*PI[3]);
            insert(bb, beta[B] + v*PI[3]);

            b_or(X[0], Y[0], tmp[0]);
            b_or(X[1], Y[1], tmp[1]);
            b_or(as, bb, tmp[2]);
            b_or(bs, ab, tmp[3]);
            for (i=2; i <= 3; i++)
               for (j=0; j <= 1; j++)
               {
                  b_or(tmp[j], tmp[i], newq);
                  if (!compare(keyblk, newq))
                  {
                     (void) printf("Key value: %d\n", key);
                     exit(1);
                  }
               }

            b_or(X[2], Y[2], tmp[0]);
            b_or(X[3], Y[3], tmp[1]);
            b_or(ai, ba, tmp[2]);
            b_or(bi, aa, tmp[3]);
            for (i=2; i <= 3; i++)
               for (j=0; j <= 1; j++)
               {
                  b_or(tmp[j], tmp[i], newq);
                  if (!compare(keyblk, newq))
                  {
                     (void) printf("Key value: %d\n", key);
                     exit(1);
                  }
               }
         }
   }
}
```

```
/***********************************************************************/
/*                                                                     */
/*  decode4.c                                                          */
/*                                                                     */
/*      Performs decoding version of part 4 of Lindner's 4v            */
/*      construction.  See also cons4v4.c and shadows4.c.              */
/*                                                                     */
/*      Author: W. John Monroe          Last Updated: 24 July 1989     */
/*      Institution: Rochester Institute of Technology                 */
/***********************************************************************/

# include "globals.h"

void decode4(PI, alpha, beta)
  unsigned PI[], *alpha, *beta;
{
   unsigned A, B, i, *newq;
   newq = (unsigned *) calloc(words, sizeof(unsigned));

   for (A=0; A < v; A++)
      for (B=0; B < v; B++)
      {
         for (i=0; i < words; i++)
            newq[i] = 0;

         insert(newq, A);
         insert(newq, alpha[A] + v*PI[1]);
         insert(newq, B + v*PI[2]);
         insert(newq, beta[B] + v*PI[3]);

         if (!compare(keyblk, newq))
         {
            (void) printf("Key value: %d\n", key);
            exit(1);
         }
      }
}
```

```
/*********************************************************************/
/*                                                                   */
/*  N2cons.c                                                         */
/*                                                                   */
/*      This program makes use of several different constructions to */
/*      generate N2-latin squares of any order n, n <> 1,2,4.  The    */
/*      largest possible square will be determined by the amount of  */
/*      memory available to hold the square, amount of disk space    */
/*      available to store the file, or by the maximum integer that  */
/*      the machine can handle, whichever comes first.  In some cases */
/*      the program will randomly generate one of a number of non-   */
/*      isomorphic squares, while in other cases the same square will */
/*      always be generated.  In most literature, the entries of latin */
/*      squares are elements of {1,2,...,n}; however, for the        */
/*      convenience of programs making use of the squares generated  */
/*      here, the entries will be elements of {0,1,...,n-1}.         */
/*                                                                   */
/*      Author: W. John Monroe          Last Updated: 24 July 1989   */
/*      Institution: Rochester Institute of Technology               */
/*********************************************************************/

# include <stdio.h>

/* the three non-isomorphic N2-latin squares of order 8 (Denniston 1978) */
unsigned sq8[3][8][8] = { { {0, 1, 2, 3, 4, 5, 6, 7},
                            {1, 2, 0, 4, 5, 6, 7, 3},
                            {2, 4, 3, 0, 6, 7, 5, 1},
                            {3, 5, 7, 2, 0, 4, 1, 6},
                            {4, 0, 6, 7, 2, 1, 3, 5},
                            {5, 6, 4, 1, 7, 3, 0, 2},
                            {6, 7, 1, 5, 3, 2, 4, 0},
                            {7, 3, 5, 6, 1, 0, 2, 4}  },

                          { {0, 1, 2, 3, 4, 5, 6, 7},
                            {1, 2, 0, 4, 5, 6, 7, 3},
                            {2, 5, 3, 0, 1, 7, 4, 6},
                            {3, 7, 1, 6, 0, 2, 5, 4},
                            {4, 6, 5, 2, 7, 0, 3, 1},
                            {5, 4, 6, 7, 3, 1, 0, 2},
                            {6, 3, 7, 5, 2, 4, 1, 0},
                            {7, 0, 4, 1, 6, 3, 2, 5}  },

                          { {0, 1, 2, 3, 4, 5, 6, 7},
                            {1, 2, 0, 4, 5, 6, 7, 3},
                            {2, 5, 3, 6, 1, 7, 0, 4},
                            {3, 7, 6, 1, 2, 4, 5, 0},
                            {4, 3, 7, 0, 6, 2, 1, 5},
                            {5, 6, 4, 7, 0, 3, 2, 1},
                            {6, 4, 1, 5, 7, 0, 3, 2},
                            {7, 0, 5, 2, 3, 1, 4, 6}  } };

/* the T4 and T6 N2-latin squares of order 3 used by Kotzig & Turgeon 1978 */
unsigned T[2][3][3] =   { { {2, 1, 0},
                            {0, 2, 1},
                            {1, 0, 2}  },

                          { {0, 2, 1},
                            {1, 0, 2},
                            {2, 1, 0}  } };
char        *calloc();
void         exit(), srand48();
double       drand48();
long         time();

/*****************************************************************/
/*                                                               */
/*  gcd()                                                        */
/*  _                                                            */
/*      Return the greatest common divisor of a0 and a1;         */
/*      a0 and a1 are assumed to be positive integers.           */
/*      (modification of Euclidean algorithm)                    */

unsigned gcd(a0, a1)
  unsigned a0, a1;
{
   unsigned a2, tmp;
   if (a0 < a1)
   {            /* swap a0 and a1 */
      tmp = a0;  a0 = a1;  a1 = tmp;
   }

   do
   {
      a2 = a0 % a1;  a0 = a1;  a1 = a2;
   }while (a1 != (unsigned) 0);

   return( a0 );
}

/*****************************************************************/
/*                                                               */
/*  cayley()                                                     */
/*                                                               */
/*      For odd n, produce the Cayley table of the cyclic        */
/*      group of order n.  It will be an N2-latin square.        */

void Cayley(sq, n)
  unsigned **sq, n;
{
   unsigned i, j;
   for (i=(unsigned)0; i < n; i++)
      for (j=(unsigned)0; j < n; j++)
         sq[i][j] = (unsigned) ((i+j) % n);
}

/*****************************************************************/
/*                                                               */
/*  KLR1()                                                       */
/*                                                               */
/*      Constructs N2-latin squares for any n = 2 (mod 4), n>2,  */
/*      using part 1 of the algorithm of Kotzig, Lindner, and    */
/*      Rosa given in their paper "Latin Squares with No         */
/*      Subsquares of Order Two and Disjoint Steiner Triple      */
/*      Systems."                                                */

void KLR1(sq, n)
  unsigned **sq, n;
{
   unsigned i, j, k;
   k = n/2;
   for (i=(unsigned)0; i < k; i++)
      for (j=(unsigned)0; j < k; j++)
      {
         sq[i][j] = sq[i+k][j+k] = (i-j+1+k) % k;
         sq[i][j+k] = ((k+i+j-1) % k) + k;
         sq[i+k][j] = ((k+i+j-2) % k) + k;
      }
}
```

```
/****************************************************************/
/*                                                              */
/*  KLR2()                                                      */
/*                                                              */
/*      Constructs N2-latin squares for any n = 2 (mod 4), n    */
/*      not a power of 2, using part 2 of the algorithm of      */
/*      Kotzig, Lindner, and Rosa given in their paper "Latin   */
/*      Squares with No Subsquares of Order Two and Disjoint    */
/*      Steiner Triple Systems."                                */

void KLR2(sq, n, m)
  unsigned **sq, n, m;  /* m is the odd factor of n */
{
   unsigned i, j, k;
   k = m;
   KLR1(sq, 2*k);

   for ( ; k <= n/4; k*=2)
      for (i=(unsigned)0; i < k; i++)
         for (j=(unsigned)0; j < k; j++)
                                        /* fill in lower right quarter */
         {
            sq[i+2*k][j+2*k] = sq[i+3*k][j+3*k] = sq[i][j];     /* A blocks */
            sq[i+3*k][j+2*k] = sq[i+k][j];                      /* B hlock  */
            sq[i+2*k][j+3*k] = sq[i][j+k];                      /* C block  */

                         /* fill in lower left and upper right quarters */
            sq[i][j+3*k] = sq[i+k][j+2*k] = sq[i+2*k][j] = sq[i+3*k][j+k]
                         = sq[i][j] + 2*k;                      /* A' blocks */
            sq[i][j+2*k] = sq[i+3*k][j] = sq[i+k][j] + 2*k;     /* B' blocks */
            sq[i+k][j+3*k] = sq[i+2*k][j+k] = sq[i][j+k] + 2*k; /* C' blocks */
         }
}
```

```
/****************************************************************/
/*                                                              */
/*  Denniston()                                                 */
/*                                                              */
/*      Chooses one of 3 non-isomorphic N2-latin squares of     */
/*      order 8 with no subsquares of order 2.  The three       */
/*      squares are given by Denniston (1978).                  */

void Denniston(sq)
  unsigned **sq;
{
   unsigned i, j, k;
   srand48(time(0));
   k = (int) (drand48() * 3);

   for (i=0; i < 8; i++)
      for (j=0; j < 8; j++)
        sq[i][j] = sq8[k][i][j];
}
```

```
/****************************************************************/
/*                                                              */
/*  KT()                                                        */
/*                                                              */
/*      Uses the algorithm of Kotzig and Turgeon [KT] to        */
/*      construct an N2-latin square of order n, where n is a   */
/*      power of 2 greater than 8, and n-3 is not a multiple    */
/*      of 5.                                                   */

void KT(sq, n)
  unsigned **sq, n;
{
   int g, k, i, j, x;
   n -= 3;
   srand48(time(0));
   g = ((int) (drand48() * (n-4))) + 2;

   while ( gcd((unsigned)((g-1)*g*(g+1)*(g+2)*(2*g+1)), n) != 1 )
      if ((g = g+1) == n-2)
         g = 2;

   Cayley(sq, n);

   for (k=(-1); k <= 1; k++)
      for (j=0; j < n; j++)
      {
         i = (g*(j+1-k)-k) - 1;
         if (i < 0)
            i += n;
         i %= n;
         sq[i][n+1+k] = sq[n+1-k][j] = sq[i][j];
         sq[i][j] = (unsigned) (n+1+k);
      }

   x = (int) (drand48() * 2);

   for (i=0; i < 3; i++)
      for (j=0; j < 3; j++)
         sq[n+i][n+j] = n + T[x][i][j];
}
```

```
/****************************************************************/
/*                                                              */
/*  dp()                                                        */
/*                                                              */
/*      Computes the direct product of squares Q1 and Q2 of     */
/*      orders n1 and n2, respectively, and places the          */
/*      resulting square in sq.                                 */

void dp(sq, Q1, Q2, n1, n2)
  unsigned **sq, **Q1, **Q2, n1, n2;
{
   unsigned i1, i2, j1, j2;
   for (i1=0; i1 < n1; i1++)
      for (i2=0; i2 < n2; i2++)
         for (j1=0; j1 < n1; j1++)
            for (j2=0; j2 < n2; j2++)
               sq[i1*n2+i2][j1*n2+j2] = (Q1[i1][j1]*n2 + Q2[i2][j2]);
}
```

```
/******************************************************************/
/*                                                                */
/*  McLeish()                                                     */
/*                                                                */
/*      From McLeish's 1975 paper we know that the direct         */
/*      product of any two N2-latin squares is also an N2-latin  */
/*      square.  She went on to use the singular direct product  */
/*      of N2-latin squares to show the existence of all         */
/*      N2-latin squares of order 2^k, k>5.  At the time, the    */
/*      question of existence of N2-latin squares of orders 8,   */
/*      16 and 32 was open.  All three cases have since been     */
/*      answered affirmatively, so we will simply use the        */
/*      direct product Q1 x Q2 where Q1 and Q2 are N2-latin      */
/*      squares of orders 8 and 16^h, h>0, respectively.  Thus   */
/*      we obtain N2-latin squares of order 2^k where k>5 and    */
/*      (2^k)-3 == 3 (mod 5).                                     */

void McLeish(sq, n)
  unsigned **sq, n;
{
   unsigned i, m, **Q1, **Q2;
   Q1 = (unsigned **) calloc(8, sizeof(unsigned*));
   for (i=0; i < 8; i++)
      Q1[i] = (unsigned *) calloc(8, sizeof(unsigned));

   Denniston(Q1);
   m = n/8;
   Q2 = (unsigned **) calloc(m, sizeof(unsigned*));
   for (i=0; i < m; i++)
      Q2[i] = (unsigned *) calloc(m, sizeof(unsigned));

   KT(Q2, m);
   dp(sq, Q1, Q2, 8, m);
}

/******************************************************************/
/*                                                                */
/*  power2()                                                      */
/*                                                                */
/*      Determines which construction to use for squares with    */
/*      orders that are powers of two.                            */

void power2(sq, n)
  unsigned **sq, n;
{
   switch ( n )
   {
      case 1:   /* these cases are included for completeness */
      case 2:   /* only--they are errors and should be       */
      case 4:   /* detected prior to this point               */
                break;
      case 8:   Denniston(sq);
                break;
      default:  switch ((n-3)%5)
                {
                   case 0:  McLeish(sq, n);
                            break;
                   default: KT(sq, n);
                            break;
                }
                break;
   }
}
```

```
/******************************************************************/
/*                                                                */
/*  main()                                                        */
/*                                                                */

main(ac,av)
  int ac;
  char *av[];
{
   unsigned i, j, m, n, **square;

   if (ac != 2)          /* command line OK? */
   {
      (void) fprintf(stderr, "Usage: %s n\n", av[0]);
      exit(1);
   }

   n = (unsigned) atoi(av[2]);      /* get n from command line*/
   if (n<=2 || n==4)
   {
      (void) fprintf(stderr, "No N2-latin square of order %lu exists.\n", n);
      exit(1);
   }

/* allocate storage for square */
   square = (unsigned **) calloc(n, sizeof(unsigned*));
   for (i=(unsigned)0; i < n; i++)
      square[i] = (unsigned *) calloc(n, sizeof(unsigned));

/* determine appropriate construction */
   if (n&1)                   /* n is odd */
      Cayley(square, n);
   else if (n&2)              /* n = 2 (mod 4) */
      KLR1(square, n);
   else                       /* n = 0 (mod 4) */
   {
      for (m=n; !(m&1); m>>=1)
         ;
      if (m>>1)               /* n has an odd factor, m */
         KLR2(square, n, m);
      else                    /* n is a power of 2 */
         power2(square, n);
   }

/* write out square */
   for (i=(unsigned)0; i < n; i++)
   {
      for (j=(unsigned)0; j < n; j++)
         if (n < 100)
            (void) printf("%3u", square[i][j]);
         else if (n < 1000)
            (void) printf("%4u", square[i][j]);
         else
            (void) printf("%5u", square[i][j]);
      (void) printf("\n");
   }
}
```

```c
/*************************************************************************/
/*                                                                       */
/*  cycle.c                                                              */
/*                                                                       */
/*      This file contains a simple program to generate a random        */
/*      n-cycle on {0,1,...,n-1}.  It makes use of the Shellsort        */
/*      described on p.290 of Aho, Hopcroft, and Ullman (1983).         */
/*                                                                       */
/*      Author: W. John Monroe          Last Updated: 21 July 1989      */
/*      Institution: Rochester Institute of Technology                  */
/*************************************************************************/

# include <stdio.h>

main(ac,av)
  int ac;
  char *av[];
{
   char     *calloc();
   double    drand48();
   void      srand48(), exit();
   long      time();
   int      *c, i, j, n, *s, incr, tmps;
   double   *r, tmpr;

/* command line OK? */
   if (ac != 2)
   {
      (void) fprintf(stderr, "Usage: %s n\n", av[0]);
      exit(1);
   }

/* get n from command line*/
   n = atoi(av[1]);
   if (n<=0)
   {
      fprintf(stderr, "ERROR: n must be greater than zero\n");
      exit(1);
   }

/* allocate storage for arrays*/
   c = (int *) calloc(n, sizeof(int));
   s = (int *) calloc(n, sizeof(int));
   r = (double *) calloc(n, sizeof(double));

/* generate list of random #s in (0,1] and initialize random sequence */
   srand48(time(0));
   for (i=0; i < n; i++)
   {
      r[i] = drand48();
      s[i] = i;
   }

/* sort random #s to generate random sequence (shellsort) */
   incr = n>>1;   /* incr = n/2 */
   while ( incr )
   {
      for (i=incr; i < n; i++)
      {
         j = i-incr;
         while (j >= 0)
            if (r[j] > r[j+incr])
            {
               tmpr = r[j];  r[j] = r[j+incr];  r[j+incr] = tmpr;
               tmps = s[j];  s[j] = s[j+incr];  s[j+incr] = tmps;
               j -= incr;
            }
            else
               j = -1;      /* break */
      }
      incr >>= 1;    /* incr = incr/2 */
   }

/* generate random cycle from random sequence */
   for (i=0; i < n; i++)
      c[s[i]] = s[(i+1)%n];

/* output the n-cycle */
   for (i=0; i < n; i++)
      if (n < 100)
         (void) printf("%3d", c[i]);
      else if (n < 1000)
         (void) printf("%4d", c[i]);
      else
         (void) printf("%5d", c[i]);
   (void) printf("\n");
}
```

```
/**********************************************************************/
/*                                                                    */
/*  checkv.c                                                          */
/*                                                                    */
/*      This program will be called by the shell script, make_perms,  */
/*      to check that the value, v, entered by the user is valid,     */
/*      i.e. v == 2 or 4 modulo 6, v > 4.                             */
/*                                                                    */
/*      Author: W. John Monroe          Last Updated: 21 July 1989    */
/*      Institution: Rochester Institute of Technology                */
/**********************************************************************/

# include <stdio.h>

void exit();

main(ac,av)
  int   ac;
  char *av[];
{
   int v;
   v = atoi(av[1]);
   if (v <= 4)
   {
      fprintf(stderr, "ERROR: v must be greater than 4\n");
      exit(1);
   }
   if (!((v%6 == 2) || (v%6 == 4)))
   {
      fprintf(stderr, "ERROR: v must be congruent to 2 or 4 modulo 6\n");
      exit(1);
   }
   exit(0);
}
```

```
#
# make_pfile.ksh
#
# This shell script is used to create the perms file needed to build a
# threshold scheme.  To do this it prompts the user for the scheme name
# and the value, v (the "length" of the permutations).  After checking to be
# sure that the user entered a valid value for v, the two programs,
# cycle and N2cons, are run and their outputs concatenated and placed into
# the appropriate file.
#
# Author: W. John Monroe                Last Updated: 21 July 1989
# Institution: Rochester Institute of Technology
#
#
read sname?'Please enter scheme name: '
read v?'Please enter the value, v, of the scheme: '
checkv $v
#
# If v is valid, continue; else quit.
#
if (( ! $? ))
then    cycle $v > _CYCLETMP
        N2cons $v > _PERMTMP
        cat _CYCLETMP _PERMTMP > $sname'.perms'
        rm -f _CYCLETMP _PERMTMP
fi
#
# END
```

APPENDIX C

User's Manual

In addition to those programs necessary to actually make use of a $(4,4,4v)$–threshold scheme with $3v$ keys, this manual also includes descriptions of the auxiliary programs that were written and used in the process of implementing and testing the main system. The program descriptions will be given in alphabetical order.

cons2v blockfile permfile 2v

Description:

cons2v will generate one or more $S(3,4,2v)$ systems from an $S(3,4,v)$ system using Lindner's 2v construction. The $S(3,4,v)$ system should be located in *blockfile* which must be an ASCII file with one quadruple per line of the file. Each quadruple is represented as four integers separated by whitespace. The integers must all be in the range $0,1,...,v-1$. The number of $S(3,4,2v)$ systems generated depends on the the number of permutations in *permfile.* One system is generated for each permutation. The $S(3,4,2v)$ systems constructed will be pairwise disjoint if and only if the permutations form an N_2–latin square. The format of *permfile* is analagous to the format *blockfile.* The value $2v$ expected on the command line is the order of the Steiner Quadruple System(s) to be constructed. It must be exactly twice the order of the quadruple system in *blockfile.* The newly generated quadruple systems will each be placed in a separate output file: quads.*2v.i* where *i* is replaced with the number of the row of *permfile* that generated it.

cons4v blockfile permfile 4v

Description:

cons4v will generate $3v$ $S(3,4,2v)$ systems from an $S(3,4,v)$ system using Lindner's 4v construction. The $S(3,4,v)$ system should be located in *blockfile* which must be an ASCII file with one quadruple per line of the file. Each quadruple is represented as four integers separated by whitespace. The integers must all be in the range $0,1,...,v-1$. *permfile* must contain $v-1$ rows each containing v integers also in the range $0,1,...,v-1$. The first row must be a v–cycle on $\mathbb{Z}_v$. If the remaining v rows form an N_2–latin square, then the $3v$ systems of order $4v$ produced will be pairwise disjoint. The value $4v$ is the

order of the quadruple systems to be produced. The newly generated quadruple systems will each be placed in a separate output file: **quads.*2v.i*** where *i* is replaced with the number of the row (minus 1) of *permfile* that generated it.

cycle n

Description:

cycle generates a random n–cycle on the integers $0,1,...,n$ and writes it on standard output.

decode schemename

Description:

decode prompts the user(s) to enter four shadows and then returns on standard output the key, if any, that block of shadows represents in threshold scheme *schemename*. The files *schemename*.quads and *schemename*.perms must exist. *schemename*.quads must contain an $S(3,4,v)$ system and *schemename*.perms must contain a v–cycle followed by an N_2–latin square of order v. See the descriptions of *cons2v* and *cons4v* for more information on file formats.

latin rectangle rows [columns]

Description:

latin is used to test the file *rectangle* to see if it contains a latin rectangle. The number of *rows* and *columns* of the file must be specified on the command line. *columns* is optional when the rectangle is square.

make_pfile

Description:

make_pfile can be used to construct an appropriate permutations file for a threshold scheme. It will prompt the user for the name of the threshold scheme for which to generate the file, and also for the value v of the system. First a random v–cycle is generated and then an N_2–latin square of order v. These are concatenated together and placed in the file *schemename*.perms.

N2cons n

Description:

N2cons will generate an N_2–latin square of order n and write it on standard output. For some values of n, the same square will always be produced, while for other values, one of a few possible squares will be randomly chosen.

N2test squarefile squaresize

Description:

N2test will test *squarefile* containing a square of size *squaresize* to see if it contains any subsquares of order 2. To be sure that *squarefile* contains an N_2–latin square, the program *latin* must also be used.

shadows schemename 4v key

Description:

shadows will generate a random block of four shadows associated with the key *key* of the threshold scheme *schemename* having order $4v$. As with *decode*, *shadows* expects the files *schemename*.quads and *schemename*.perms to exist. *schemename*.quads must contain an $S(3,4,v)$ system and *schemename*.perms must contain a v–cycle followed by an N_2–latin square of order v. See the descriptions of *cons2v* and *cons4v* for more information on file formats. $4v$ must be congruent to $8, 16$ *modulo* 24 and $0 \leq key < 3v$.

APPENDIX D

Some Quadruple Systems

Below are listed two $S(3,4,8)$ systems and one $S(3,4,10)$ system. The ***cons2v*** program described in Appendix B may be used to obtain quadruple systems of orders $\{16,20,32,40,64,80,...\}$ from these systems. See [LR] for methods of obtaining quadruple systems of other admissible orders.

Two Pairwise Disjoint $S(3,4,8)$ Systems

0 1 2 7	0 1 2 6
0 1 3 6	0 1 3 4
0 1 4 5	0 1 5 7
0 2 3 5	0 2 3 7
0 2 4 6	0 2 4 5
0 3 4 7	0 3 5 6
0 5 6 7	0 4 6 7
1 2 3 4	1 2 3 5
1 2 5 6	1 2 4 7
1 3 5 7	1 3 6 7
1 4 6 7	1 4 5 6
2 3 6 7	2 3 4 6
2 4 5 7	2 5 6 7
3 4 5 6	3 4 5 7

An $S(3,4,10)$ System

0 1 2 4	0 4 6 8	2 3 4 9
0 1 3 6	0 5 6 7	2 3 6 8
0 1 5 8	1 2 3 7	2 4 5 8
0 1 7 9	1 2 5 6	2 4 6 7
0 2 3 5	1 2 8 9	2 5 7 9
0 2 6 9	1 3 4 8	3 4 5 6
0 2 7 8	1 3 5 9	3 5 7 8
0 3 4 7	1 4 5 7	3 6 7 9
0 3 8 9	1 4 6 9	4 7 8 9
0 4 5 9	1 6 7 8	5 6 8 9