The Construction and Implementation of Authentication and Secrecy Codes

by Wendy Jane White

A thesis presented to

the University of Manitoba

in partial fulfillment of the requirements

for the degree of

Master of Science

in the Department of Computer Science.

ISBN 0-315-71755-6

THE CONSTRUCTION AND IMPLEMENTATION OF AUTHENTICATION AND SECRECY CODES

BY

WENDY JANE WHITE

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the sole purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocoping or by any other means, in total or in part, at the request of other institutuions or individuals for the purpose of scholarly research.

Wendy J. White.

Abstract

In a cryptographic system, a transmitter sends a message representing a source state to a receiver. An opponent may overhear this message. Secrecy codes are cryptographic systems where overhearing a message gives the opponent no information about the source state. Authentication codes are cryptographic systems where overhearing one or more messages makes it no easier for the opponent to send his or her own messge to the receiver and have it considered authentic.

This thesis examines these structures. We look at combinitorial bounds on their size, and some techniques for constructing authentication and secrecy codes close to optimal size. We then explore how to implement these codes efficiently, which leads us into a search for efficient algorithms for performing arithmetic operations over finite fields.

Acknowledgements

I would like to thank my supervisor, Doug Stinson, for his advice and his unwillingness to let me submit anything that had not been examined from all the angles that I could think of (and then Doug would find a couple more). This document is much more interesting because of that. I would also like to thank the rest of the CS grad students for making long hours typing, revising, and rethinking somewhat more bearable, and my friend Twilla, for reminding me that there was life outside my thesis. Finally, my family, not only for their repeated offers to take me away from this cold flat place, but also for demonstrating that the acquisition of knowledge is worth working for.

Table of Contents

Chapter 1 : Introduction

1.1 Aims and objectives

The aim of this thesis is to detail the state of the field of secrecy and authentication codes with unconditional security.

Chapter 2 introduces us to the concept of a cryptosystem, and some of the unconditionally secure codes that can be used in that system. In a cryptosystem, a transmitter sends information to a receiver via a channel, into which an opponent may be listening. A code is a collection of encoding rules (which the opponent may know). The transmitter and receiver are given which encoding rule to use by the key source, and this information is kept secret. A secrecy code is a code in which the opponent can get no information about the source state (or information) being conveyed by overhearing one or more messages in the channel. A code that is secure against an authentication attack is a code in which the opponent can place messages into the channel, but has no added chance of fooling the receiver into thinking that they are authentic by seeing zero or more messages in the channel. We discuss codes that can provide both secrecy and authentication properties simultaneously. Finally, we look at a code that provides perfect disclosure whilst maintaining authentication properties (i.e. anyone listening into the channel knows what information (source state) is being conveyed, but has no added chance of faking a message by listening in to the channel). Chapter 2 goes on to prove some bounds on the size of the codes needed to achieve the properties. The bounds are proven independently of the probability distributions on the possible source states. We

define the concept of an optimal code that achieves a certain level of secrecy or authentication, which has the smallest number of encoding rules needed.

Chapter 3 details constructions of infinite classes of optimal and near-optimal codes from combinatorial designs such as t-designs, Latin squares, transversal designs and perpendicular arrays. The encoding rules can be written into an encoding array, which we use to formalize their properties in terms of required combinations of messages in rows and columns. This provides the link between combinatorial designs and the related secrecy and authentication codes.

Chapter 4 is concerned with implementations. Making a code close to optimal is not going to do us much good unless we can implement it efficiently. Typically, storing the entire encoding array is infeasible, due to its large size. Using the algebraic structure of the underlying combinatorial designs can make implementing the codes efficient in terms of time and space. We write some algorithms, in a language that assumes that operations in finite fields can be performed.

Chapter 5 is concerned with lower level implementations. Many of the codes in Chapter 4 are based on infinite classes of combinatorial structures computed using finite fields. Therefore we implement these codes using arithmetic in the underlying field. Hence, the algorithms that we choose to implement finite field arithmetic are important to the design of the transmitter's and receiver's algorithms. Chapter 5 makes a foray into the literature on finite field algorithms, in software and hardware, sequentially and in parallel, from the straightforward approaches found in Knuth [37] to the sophistication of the Coppersmith-Aldeman logarithm algorithm.

Chapter 6 ties all this together, and draws some conclusions about the nature and effectiveness of the algorithms in Chapter 4 based on what was discovered in Chapter 5.

Chapter 2 : Optimal Codes

2.1 Shannon's Model of a Cryptosystem

In 1949, Shannon [57] developed the theory of Secrecy Codes. It was extended by Simmons [58] to include Authentication Codes in 1982. The *secrecy model* contains three participants, the *transmitter*, the *receiver*, and the *opponent*. The transmitter and the receiver are connected by means of an *open channel*, into which the opponent may be listening. The transmitter places *messages* (or cyphertext) into the channel, in order to communicate a *source state* (or plaintext) to the receiver. The possible source states have a probability distribution associated with them, which is known to the opponent. An *encoding rule* (or *key*) is a one-to-one function from the set of source states, $\mathcal{S}$, to the set of messages, $\mathcal{M}$. $\mathcal{E}$ is the set of encoding rules. So, for $e \in \mathcal{E}$ and $s \in \mathcal{S}$, $e(s) \in \mathcal{M}$. The encoding rules must be one-to-one so that the messages may be uniquely decoded. A *code* is a set of encoding rules. The encoding rules in that set are usually chosen carefully to give the code certain properties. The *encoding array* is an array whose rows are indexed by the possible encoding rules and whose columns are indexed by the source states. The entry in the row indexed by e and the column indexed by s is the message e(s). The opponent may be assumed to know this array. Before any messages are sent, the *key source* selects one of the encoding rules at random with a fixed (and known) probability distribution on the rules and notifies the transmitter and the receiver of which rule it has chosen by means of a *secure channel*. Simmons extends the model so that the

opponent can place messages in the channel as well as just read them, in an attempt to fool the receiver. See Figure 2.1.

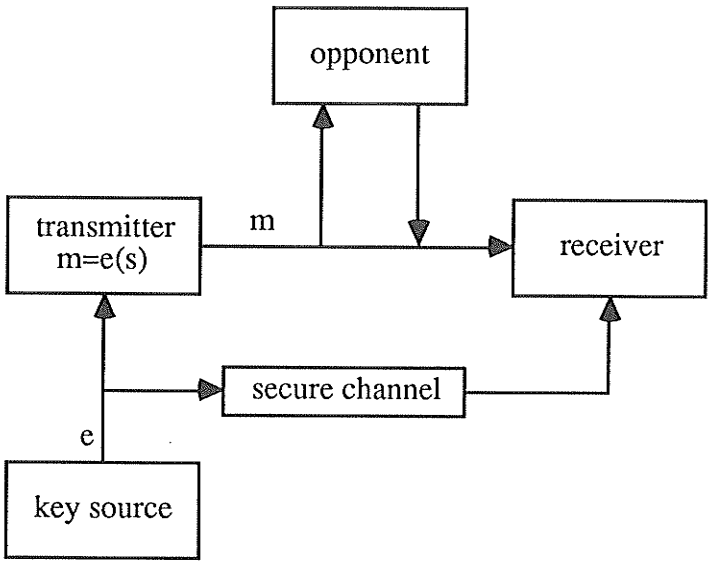

Figure 2.1 Shannon's model of a cryptosystem.

In this discussion, $k = |\mathcal{S}|$ is reserved to mean the number of source states, $v = |\mathcal{M}|$ the number of messages and $b = |\mathcal{E}|$ the number of encoding rules. (These variables were chosen because of the connection to balanced incomplete block designs which will become apparent later). So the encoding array is a b by k array, with v distinct entries. For any encoding rule e, let p(e) be the probability that e is chosen by the key source. Similarly, for any source state s, , we let p(s) be the (a prori) probability that the source state is s. We require that for every encoding rule e, $p(e) > 0$, and for every source state s, $p(s) > 0$. This is easy to achieve, by just deleting source states and encoding rules which cannot occur from the respective sets $\mathcal{S}$ and $\mathcal{E}$.

Example 2.1.1. Consider the following secrecy code.

The $k = 3$ source states are labelled s_1, s_2, s_3.

The $v = 4$ messages are 1, 2, 3, 4.

The $b = 4$ encoding rules are labelled e_1, e_2, e_3, e_4.

The key source selects each rule with probability $\frac{1}{4}$.

The encoding array is as follows.

	s_1	s_2	s_3
e_1	1	2	3
e_2	4	1	2
e_3	3	4	1
e_4	2	3	4

So, for example, if the key source sent encoding rule e_2, and the transmitter wished to communicate the source state s_3, then a 2 would be placed in the channel.

Notice how each message occurs at most once in each encoding rule.

Theorem 2.1.2. In any code $v \geq k$.

Proof. Encoding rules are one-to-one functions, so $e(s) \neq e(t)$, for $s \neq t$, $s,t \in \mathbf{S}$. ■

2.2 Secrecy

Cryptosystems are designed to achieve one of three types of security. *Computational* security occurs when the opponent cannot get any information from the message in a reasonable time because no method has yet been discovered to do so. *Provable* security occurs when the task of extracting information from a message in the channel without knowing which of the encoding rules was used can be shown to be equivalent to some presumably difficult problem (for example, factoring a large number). *Unconditional* security (called *theoretical* by Shannon), which our codes will achieve, occurs when the opponent can obtain no information from the message, no matter what computational resources he or she has available. This is the strongest form of security.

There is always a probability that the opponent may 'guess' the source state, so unconditional security is defined in terms of probabilities. Conceptually, perfect secrecy is when the probability that the transmitter is sending source state $s \in \mathbf{S}$, given that the

message $m \in \mathcal{M}$ has been seen in the channel, is the same as the a proiri probability that the source state is s. Multiple levels of perfect secrecy can be defined in the same way. Suppose L_S distinct messages are seen in the channel, that is, there are no duplicate messages. Suppose further that the same encoding rule is known to be used to encode them. We consider the messages to be unordered, for mathematical simplicity. (See Massey [44] for a discussion of ordered L_S-tuples of messages or Godlewski and Mitchell [29] for a comparison of different possible definitions of unconditional security). Given any L_S distinct source states, if the probability of these being the source states that correspond to the messages (setwise) is the same as the a proiri probability that those are the set of source states, then the code is said to be perfectly L_S-secret.

More formally, let P(x) be the probability that x happens, and let P(x | y) be the probability that x occurs given that y has already occurred. We say that a code has *perfect L_S-fold secrecy* if, for every $L \leq L_S$, and every set M of L messages observed in the channel, and every set S of L source states, we have that P(S | M) = P(S).

Example 2.2.1. Consider the code of example 2.1.1, with encoding array

	s_1	s_2	s_3
e_1	1	2	3
e_2	4	1	2
e_3	3	4	1
e_4	2	3	4

We now examine the secrecy level of this code.

Suppose that a 1 is seen in the channel. Then the probability that the source state being transmitted is s_1 is given by

$$P(s_1 \mid 1) = \frac{P(1 \mid s_1)\, P(s_1)}{P(1)}$$

$$= \frac{P(e_1)\, P(s_1)}{P(e_1)\, P(s_1) + P(e_2)\, P(s_2) + P(e_3)\, P(s_3)}$$

$$= \frac{\frac{1}{4}P(s_1)}{\frac{1}{4}P(s_1) + \frac{1}{4}P(s_2) + \frac{1}{4}P(s_3)}$$

$$= P(s_1).$$

Similarly, it can be shown that $P(s_2 \mid 1) = P(s_2)$ and $P(s_3 \mid 1) = P(s_3)$.

So, seeing a 1 in the channel gives the opponent no information about the source state. The same argument holds for seeing any of the other messages in the channel. That is, the code in Example 2.1.1 has 1-fold secrecy.

To examine whether the code has 2-fold secrecy, suppose that the set of messages $\{1,2\}$ is seen in the channel (in some order). Suppose also that it is known that the same encoding rule was used to create both the message 1 and the message 2.

Then that encoding rule was either e_1 or e_2, and the source states encoded as $\{e_1,e_2\}$ are $\{s_1,s_2\}$ or $\{s_2,s_3\}$. So $P(\{s_1,s_3\} \mid \{e_1,e_2\}) = 0 \neq P(\{s_1,s_3\})$. So seeing 2 messages in the channel (by the same encoding rule) gives the opponent some information about the source states being transmitted. So the code in Example 2.1.1 does not have 2-fold secrecy. It does have 3-fold secrecy, trivially.

2.3 Some Necessary Probability Definitions

A more formal language is needed in which to prove subsequent combinatorial bounds on the size of the codes So in this section we redefine some of the concepts that we've been looking at in statistical terms

Given an encoding rule e, define $M(e) = \{e(s): s \in \mathcal{S}\}$, the set of all messages *valid* under e.

For a set M of distinct messages, and an encoding rule e, define $f_e(M) = \{s: e(s) \in M\}$, the set of source states encoded by e to a message in M.

Give a set M of distinct messages, define $E(M) = \{e \in \mathcal{E}: M \subseteq M(e)\}$, the set of encoding rules under which all the messages in M are valid.

2.4 Authentication

Let us now consider the opponent's ability to place a message not yet seen into the channel with the intention of having the receiver accept it as authentic. To guard against such an attack, it is necessary to introduce redundancy into the code. If the opponent observes L_A messages in the channel which were encoded using the same encoding rule, and then adds a new message of his or her own, it is called a *spoofing attack of level* L_A. The special case $L_A = 0$ is called *impersonation*, and the case $L_A = 1$ is called *substitution.*

Let Pd_i be the probability that the opponent can deceive the receiver with a spoofing attack of level i (so that the opponent is using his or her best stategy). Now, there are k source states, so any encoding rule contains k allowable messages, of the v possible. If i of each have been seen in the channel, then it seems reasonable to expect that $Pd_i \geq \frac{k-i}{v-i}$. This is indeed true, as we prove in the following theorem.

Theorem 2.4.1. (Massey [44, p. 12]) In any authentication code with k source states and v messages, $Pd_i \geq \frac{k-i}{v-i}$.

Proof. Let $M = \{m_1, m_2, \ldots, m_i\}$ be the i messages observed in the channel. For $m \in \mathcal{M} \setminus M$, a message not already seen, define payoff (M,m) in the following way.

payoff(M,m)

$$= P(m \text{ is accepted as authentic} \mid \text{the messages in M have been seen in the channel})$$

$$= \frac{P(\{m\} \cup M \text{ is accepted})}{P(M \text{ is accepted})}$$

$$= \frac{\sum_{e \in E(\{m\} \cup M)} P(e)P(S=f_e(M))}{\sum_{e \in E(M)} P(e)P(S=f_e(M))}.$$

$$\text{So} \quad \sum_{m \in \mathcal{M} \setminus M} \text{payoff}(M,m) = \frac{\sum_{m \in \mathcal{M} \setminus M} \sum_{e \in E(\{m\} \cup M)} P(e)P(S=f_e(M))}{\sum_{e \in E(M)} P(e)P(S=f_e(M))}$$

$$= \frac{(k-i) \sum_{e \in E(M)} P(e)P(S=f_e(M))}{\sum_{e \in E(M)} P(e)P(S=f_e(M))}$$

$$= k-i.$$

This is true since each $e \in E(M)$ contains precisely k-i other valid messages besides those in M, and hence corresponds to k-i of the messages in E(M). That is, there are k-i 'unused' source states left, and each is encoded by e to some message e(m).

So the average payoff is $\frac{k-i}{v-i}$, and the opponents strategy is to choose the $m \in \mathcal{M} \setminus M$, whose payoff is greater than or equal to the average. ■

So, for example, $Pd_0 \geq \frac{k}{v}$, and $Pd_1 \geq \frac{k-1}{v-1}$.

A code (often called an *authentication* code) is *L_A-fold secure against spoofing*, or *able to withstand a spoofing attack of level L_A* if $Pd_L = \frac{k-L}{v-L}$, for all $0 \leq L \leq L_A$. the bound is called the *combinatorial* bound, since it is independent of the probability distributions on the source states and encoding rules.

Example 2.4.2. The code in Example 2.1.1 with encoding array

	s_1	s_2	s_3
e_1	1	2	3
e_2	4	1	2
e_3	3	4	1
e_4	2	3	4

is 0-fold secure against spoofing, since each of the messages 1, 2, 3, or 4 is accepted with probability $\frac{3}{4}$.

It is not 1-fold secure against spoofing in general.

Suppose that a 1 is seen in the channel, and a 2 is inserted. The 2 is accepted as genuine if the encoding rule was e_1 or e_2.

The probability that the 2 is accepted is given by

$$P(e_1 \mid 1) + P(e_2 \mid 1)$$

$$= \frac{P(1 \mid e_1)\, P(e_1)}{P(1)} + \frac{P(1 \mid e_2)\, P(e_2)}{P(1)}$$

$$= \frac{P(s_1)\, P(e_1) + P(s_2)\, P(e_2)}{P(1)}$$

$$= \frac{\frac{1}{4}(P(s_1) + P(s_2))}{P(s_1)\, P(e_1) + P(s_2)\, P(e_2) + P(s_3)\, P(e_3)}$$

$$= P(s_1) + P(s_2),$$

which does not equal $\frac{k-1}{v-1} = \frac{2}{3}$ in general.

Similarly, the probability that 3 is accepted if a 1 has been seen is $P(s_1) + P(s_3)$, and the probability that 4 is accepted if a 1 has been seen is $P(s_2) + P(s_3)$. So this code is 1-fold secure against spoofing only if the source states are equiprobable. For more discussion on equiprobable probability distributions on the source states, see §2.6.

Example 2.4.3. Consider the following authentication code.

The $k = 3$ source states are labelled s_1, s_2, s_3.

The $v = 4$ messages are 1, 2, ... ,7.

The $b = 21$ encoding rules are labelled e_1, e_2, ... , e_{21}.

The key source selects each rule with probability $\frac{1}{21}$.

The encoding array is as follows.

	s_1	s_2	s_3
e_1	1	2	4
e_2	4	1	2
e_3	2	4	1
e_4	2	3	5
e_5	5	2	3
e_6	3	5	2
e_7	3	4	6
e_8	6	3	4
e_9	4	6	3
e_{10}	4	5	7
e_{11}	7	4	5
e_{12}	5	7	4
e_{13}	5	6	1
e_{14}	1	5	6
e_{15}	6	1	5
e_{16}	6	7	2
e_{17}	2	6	7
e_{18}	7	2	6
e_{19}	7	1	3
e_{20}	3	7	1
e_{21}	1	3	7

The rows of this code form a (7,3,1)-BIBD in which each row is expanded into a Latin square of order 3. We examine the level of authentication of this code.

The code is 0-fold secure against spoofing, (or can resist an impersonation attack), since every message is accepted with probability $\frac{3}{7}$. This is so because each message occurs in 9 of the 21 encoding rules.

To determine whether the code is 1-fold secure against spoofing, suppose the opponent sees a 1 in the channel and places a 2 in the channel. The probability that the 2 is accepted as authentic is given by

$$P(e_1 \mid 1) + P(e_2 \mid 1) + P(e_3 \mid 1)$$

$$= \frac{P(1 \mid e_1)\,P(e_1)}{P(1)} + \frac{P(1 \mid e_2)\,P(e_2)}{P(1)} + \frac{P(1 \mid e_3)\,P(e_3)}{P(1)}$$

$$= \frac{P(s_1)\,P(e_1) + P(s_2)\,P(e_2) + P(s_3)\,P(e_3)}{P(1)}$$

$$= \frac{\frac{1}{21}(P(s_1) + P(s_2) + P(s_3))}{\frac{3}{21}(P(s_1) + P(s_2) + P(s_3))}$$

$$= \frac{1}{3} = \frac{2}{6} = \frac{k\text{-}1}{v\text{-}1}.$$

This result holds for any two messages since each pair of messages occurs in the same number of rows, and within those rows, each of the pair occurs the same number of times in each column. So the code is 1-fold secure against spoofing.

The code is not 2-fold secure against spoofing. If the set of messages {1,2} is seen in the channel, then the encoding rule used was e_1, e_2 or e_3, and the message 4 is accepted with probability 1, (i.e. all the time) not $\frac{k\text{-}2}{v\text{-}2} = \frac{1}{5}$ as required.

It is often desirable to use a code that has both secrecy and authentication properties. An (L_S, L_A)-code is a code that has perfect L_S-fold secrecy and is able to withstand a spoofing attack of level L_A. The cases $L_A = L_S$ and $L_A = L_S + 1$ are the most widely discussed, the former corresponding to the scenario in which the opponent can modify existing messages, and the latter to the one in which the opponent can merely insert messages.

2.5 Perfect Disclosure Codes

A code is said to be a *perfect disclosure* code if each message only corresponds to one source state. More formally, for any two distinct encoding rules e_1 and e_2, and two source state s_1 and s_2, $e_1(s_1) = e_2(s_2)$ implies that $s_1 = s_2$. This means that for each message $m \in \mathcal{M}$, valid under some encoding rule, there is a source state $s \in \mathcal{S}$ such that $p(s \mid m) = 1$.

Theorem 2.5.1. (Stinson, [60, Theorem 5.1]) If a code has perfect disclosure then $Pd_0 \geq \frac{k}{v}$. Moreover, if $Pd_0 = \frac{k}{v}$, then $Pd_i \geq \frac{k}{v}$, for all $i \geq 0$.

Proof. $Pd_0 \geq \frac{k}{v}$, by Theorem 2.4.1.

Suppose that $Pd_0 = \frac{k}{v}$. Then if the code has perfect disclosure, there are exactly $\frac{v}{k}$ messages encoding each source state. Let $M = \{m_1, m_2, \ldots, m_i\}$ be the i messages observed in the channel. Consider a column C of the encoding array containing none of the messages in M. Define payoff(M,m) for each of the messages in column C as we did in the proof of Theorem 2.4.1. Then

$$\sum_{m \in C} \text{payoff}(M,m) = 1,$$

since each encoding rule containing M has an entry in C. So the average payoff is $\frac{k}{v}$, and hence $Pd_i \geq \frac{k}{v}$, for all $i \geq 0$. ■

However, for some probability distributions, we can arrange $Pd_i < \frac{k}{v}$ for some $i \geq 1$ if we let $Pd_0 > \frac{k}{v}$. Consider the following example.

Example 2.5.2. Consider the following perfect disclosure code, where $k = 3$, $v = 5$, and $b = 4$. The encoding rules are chosen equiprobably.

Let $p(s_1) = \frac{9}{10}$, $p(s_2) = p(s_3) = \frac{1}{20}$.

	s_1	s_2	s_3
e_1	1	2	4
e_2	1	3	4
e_3	1	2	5
e_4	1	3	5

Notice that each message occurs only once in each row. So if a 1 is seen in the channel, the opponent knows that the source state being sent is s_1, if a 2 or a 3 is seen, the source state is s_2, and if a 4 or a 5 is seen, then the source state is s_3.

An impersonation attack (i.e. a spoofing attack of level 0) will always succeed if the opponent places a 1 into the channel. Hence $Pd_0 = 1$.

If a 2, 3, 4 or 5 is seen in the channel, then a 1 placed in the channel will be automatically be accepted as authentic. However, if a 1 is seen in the channel, then the probability that a 2 is accepted as authentic is

$$P(e_1 \mid 1) + P(e_2 \mid 1)$$

$$= \frac{P(1 \mid e_1)\, P(e_1)}{P(1)} + \frac{P(1 \mid e_2)\, P(e_2)}{P(1)}$$

$$= \frac{P(s_1)\, P(e_1) + P(s_1)\, P(e_2)}{P(1)}$$

$$= \frac{\frac{1}{2} P(s_1)}{P(s_1)}$$

$$= \frac{1}{2}.$$

Similarly, the probability that 3 or 4 or 5 is accepted if a 1 is seen is $\frac{1}{2}$.

So $Pd_1 = P(1) \cdot P(\text{success if a 1 seen}) + P(2,3,4,\text{or } 5) \cdot P(\text{success if } 2,3,4,\text{or } 5 \text{ seen})$

$$= \frac{9}{10} \cdot \frac{1}{2} + \frac{2}{20} \cdot 1$$

$$= \frac{11}{20} < \frac{k}{v}.$$

We say that a perfect disclosure code is *L_A-fold secure against spoofing* or *able to withstand a spoofing attack of Level L_A* if $Pd_i = \frac{k}{v}$, for all $0 \le i \le L_A$. (Compare to authentication codes in §2.4.)

Example 2.5.3. Consider the following perfect disclosure code, where k = 3, v = 9 and b = 9. The encoding rules are chosen equiprobably.

	s_1	s_2	s_3
e_1	1	4	7
e_2	1	5	8
e_3	1	6	9
e_4	2	4	9
e_5	2	5	7
e_6	2	6	8
e_7	3	4	8
e_8	3	5	9
e_9	3	6	7

Notice that each message appears in only one column.

Each message appears in $\frac{1}{3}$ of the equiprobable encoding rules, so $Pd_0 = \frac{1}{3}$.

Each pair of messages from different columns appears in precisely one of the equiprobable encoding rules, so $Pd_1 = \frac{1}{3}$. (This bound can, of course be proven more formally by the same sort of probability calculations done in earlier examples.)

$Pd_2 = 1$, since choosing the unique message in the row containing the 2 messages seen in the channel is automatically accepted.

So this perfect disclosure code is 1-fold but not 2-fold secure against spoofing.

2.6 Arbitrary and Specific Source State Probability Distributions

Theorem 2.6.1. (Stinson [60, Theorem 2.11]) Suppose a code C achieves perfect L_S-fold secrecy for a given source distribution p_0. Then C achieves perfect L_S-fold secrecy for an arbitrary source distribution p_1.

Proof. Let p be the probability distribution on the encoding rules. Let $L \leq L_S$. Let $S \subseteq \mathcal{S}$ be a set of L source states. Let $M \subseteq \mathcal{M}$ be a set of L messages. Since C is L_S-fold secret, $p_0(S \mid M) = p_0(S)$.

By Bayes' theorem, $p_0(M) = p_0(M \mid S)$, and

$$\sum_{\{e:M\subseteq M(e)\}} p(e)p_0(f_e(M)) = \sum_{\{e:S=f_e(M)\}} p(e) .$$

We attempt to formulate the same equality for p_1.

So,
$$\sum_{\{e:M\subseteq M(e)\}} p(e)p_1(f_e(M)) = \sum_{\{S\subseteq \mathbf{S}:|S|=L\}} p_1(S) \cdot \sum_{\{e:S=f_e(M)\}} p(e)$$

$$= \sum_{\{S\subseteq \mathbf{S}:|S|=L\}} p_1(S) \cdot \sum_{\{e:M\subseteq M(e)\}} p(e)p_0(f_e(M)) \quad \text{(from above)}$$

$$= 1 \cdot \sum_{\{e:M\subseteq M(e)\}} p(e)p_0(f_e(M)) \qquad \text{(since } \sum_{\{S\subseteq \mathbf{S}:|S|=L\}} p_1(S) = 1\text{)}$$

$$= \sum_{\{e:S=f_e(M)\}} p(e)$$

as required. ■

This nice state of affairs unfortunately does not hold for authentication.

For example, consider the following code for k=3, v=7, b=7. It is secure against a spoofing attack of order 1 if the each source state is equiprobable. (Note, in passing, the similarities between this code and the projective plane of order two.)

	s_1	s_2	s_3
e_0	1	2	4
e_1	2	3	5
e_2	3	4	6
e_3	4	5	0
e_4	5	6	1
e_5	6	0	2
e_6	0	1	3

Encoding rules are chosen equiprobably.

If a 1 is seen in the channel, then 2 or 4 is accepted if the encoding rule is e_0 and the original source state is s_1, 5 or 6 is accepted if e_4 and s_3 occur, and 0 or 3 is accepted if e_6 and s_2 occur.

If $P(s_1) = P(s_2) = P(s_3) = \frac{1}{3}$, then the probability of deception is $\frac{1}{3} = \frac{k}{v}$.

For example, if a 1 was seen in the channel, then $p(e_0) = p(e_4) = p(e_6) = \frac{1}{3}$. Then any message other than 1 is accepted with probability $\frac{1}{3}$.

Suppose, however, that $P(s_1) = \frac{1}{2}$, and $P(s_2) = P(s_3) = \frac{1}{4}$. Then the optimal substitution strategy is as follows.

If $0 \leq x \leq 6$ is seen in the channel, then $P(e_{x-1}) = \frac{1}{2}$, where x-1 is taken modulo 7. So choose one of the other messages valid under e_{x-1}. This messages is accepted with probability $\frac{1}{2}$. For example, if a 1 is seen in the channel, then $P(e_0) = \frac{1}{2}$, so 2 or 4 is accepted with probability $\frac{1}{2}$.

So under this optimal strategy, the probability of deception is $\frac{1}{2} > \frac{k}{v}$.

So the code is not 1-fold secure against spoofing for an arbitrary source distribution.

2.7 Combinatorial Bounds and Optimal Codes

It is important to keep the number b of encoding rules in a code relatively small. This minimizes the amount of secret information that the key source must send to the transmitter and receiver, and that must subsequently be kept guarded. It is clear that in order to be able to communicate a rule, $\log_2 b$ bits of information must be sent.

We develop combinatorial bounds on b, that is, bounds independent of the probability distributions on the source states and encoding rules. A code is considered optimal if b is 'as small as possible', a concept that will be formalized forthwith.

The rest of this section deals with establishing these bounds.

Theorem 2.7.1 deals with the bound on the number of encoding rules required for a code to have secrecy properties.

Theorem 2.7.1. (Stinson [60, Theorem 2.1]) In a code that achieves perfect L_S-fold secrecy,

$$b \geq \binom{k}{L_S}.$$

Proof. Consider a code, C, that achieves perfect L_S-fold secrecy. Let e be an encoding rule of C. Let M(e) be the set of messages valid under e. Let M_1 be a subset of M(e) containing L_S messages. Let S_1 be any set of L_S source states. If there is no encoding rule e_1 such that $S_1 = f_{e_1}(M(e))$ then $p(S_1 \mid M_1) = 0 \neq p(S_1)$, contradicting perfect L_S-fold secrecy. So there is an encoding rule for every $M_1 \subseteq M(e)$. There must be at least $\binom{k}{L_S}$ encoding rules in order to achieve this. ■

In Theorem 2.7.2, we establish a bound on the number of encoding rules required for authentication codes. This bound is valid for both equiprobable and arbitrary source state probability distributions.

Theorem 2.7.2. (Massey [44]) In an authentication code that is L_A-fold secure against spoofing (and regardless of the source state probability distribution),

$$b \geq \frac{\binom{v}{L_A+1}}{\binom{k}{L_A+1}}.$$

Proof. Let $e \in \mathcal{E}$ be an encoding rule. Let $M \subseteq M(e)$ be a set of $i \leq L_A$ messages. Let $x \in \mathcal{M} \setminus M$. Suppose there is no rule $g \in \mathcal{E}$ under which all the messages in the set $M \cup \{x\}$ are valid. Then, as in the proof of Theorem 2.4.1,

$$\sum_{m \in \mathcal{M} \setminus M} \text{payoff}(M,m) = k\text{-}i.$$

Now payoff(M,g) = 0, so there is some message $y \in \mathcal{M} \setminus M$ such that $\text{payoff}(M,y) > \frac{k\text{-}i}{v\text{-}i}$, contradicting L_A-fold security against spoofing.

So every L_A+1-subset of messages is valid under at least one encoding rule. That is, $b\binom{k}{L_A+1} \geq \binom{v}{L_A+1}$. ■

Theorems 2.7.3 and 2.7.4 develop bounds on the number of encoding rules required for codes that have both secrecy and authentication properties.

Theorem 2.7.3. (Stinson [61, Theorem 2]) In a $(L_S, L_S\text{-}1)$-code, $b \geq \binom{v}{L_S}$.

Proof. Let e be an encoding rule of the code. Let $M \subseteq M(e)$ be a subset of messages, $|M| = i \leq L_S\text{-}1$. Let x be a message not in M. Suppose that there is no encoding rule under which $M \cup \{x\}$ is valid. Then, since

$$\sum_{m \in \mathcal{M} \setminus M} \text{payoff}(M,m) = k\text{-}i,$$

and payoff(M,x) = 0, then there is some message $y \in \mathcal{M} \setminus M$ such that $\text{payoff}(M,y) > \frac{k\text{-}i}{v\text{-}i}$, contradicting $(L_S\text{-}1)$-fold security against spoofing. So every L_S-subset of messages, M, is valid under at least one encoding rule.

In order to achieve perfect L_S-fold secrecy, the messages in M must be an encoding of each of the $\binom{k}{L_S}$ possible L_S-subsets of source states. So M is a valid set of messages under at least $\binom{k}{L_S}$ encoding rules.

We count pairs of the form (e,M), where e is an encoding rule, $|M| = L_S$, and $M \subseteq M(e)$. If e is chosen first, then each of the b choices for e generates $\binom{k}{L_S}$ messages, one for each L_S-subset of source states encoded. The number of such pairs is exactly $b\binom{k}{L_S}$. Conversely, suppose that M is chosen first There are $\binom{v}{L_S}$ possible subsets M, and at least $\binom{k}{L_S}$ choices for e given M. So we get $b\binom{k}{L_S} \geq \binom{v}{L_S}\binom{k}{L_S}$ that is, $b \geq \binom{v}{L_S}$. ■

Theorem 2.7.4. (Stinson [60, Theorem 4.1]) In a (L_S, L_S)-code valid for an arbitrary source state probability distribution, $b \geq \binom{v}{L_S}\frac{v\text{-}L_S}{k\text{-}L_S}$.

Proof. As in the proof of theorem 2.7.2, every set of $L_S + 1$ messages is valid under at least one encoding rule. As in theorem 2.7.3, every set of L_S messages is the encryption of every possible set of L_S source states.

Let $e \in \mathcal{E}$ be an encoding rule. Let $M \subseteq M(e)$ be a set of $i \leq L_S$ messages. Let $S = f_e(M)$. Suppose that $p(S) = 1 - \varepsilon$, where ε is a small positive real number.

Put $E = \{e: \{e(s): s \in S\} = M\}$, the set of encoding rules that encode the source states in S to the messages in M. Because we have L_S-fold security against spoofing, for all $x \notin M$, there exists an $e \in E$ under which x is valid. So $|E| \geq \frac{v-L_S}{k-L_S}$.

Count triples of the form (e,S,M), where e is an encoding rule, S is a set of L_S source states, M is a set of L_S messages and e(S) = M. Choosing any e and any S there is a unique M, giving $b\binom{k}{L_S}$ triples. Choosing any M, and any S, there are at least $\frac{v-L_S}{k-L_S}$ encoding rules e such that e(S) = M. So there are at least $\binom{v}{L_S}\binom{k}{L_S}\frac{v-L_S}{k-L_S}$ triples. Hence $b\binom{k}{L_S} \geq \binom{v}{L_S}\binom{k}{L_S}\frac{v-L_S}{k-L_S}$, as required. ■

Finally, Theorem 2.7.5 establishes a bound on the number of encoding rules required for perfect disclosure codes to have authentication capabilities.

Theorem 2.7.5. (Stinson [60, Theorem 5.2]) In a code which has perfect disclosure and $Pd_i = \frac{k}{v}$, for $0 \leq i \leq L$, then $b \geq \left(\frac{v}{k}\right)^{L+1}$.

Proof. We prove the following assertion.

Assertion. Every set of messages M, |M| = n, such that no two messages in M correspond to the same source state, is valid under at least one encoding rule. When n = 0, it is clear that every message is valid under at least one encoding rule, since otherwise $Pd_0 > \frac{k}{v}$. So the assertion holds for n = 0.

Assume that the assertion holds for all $n \leq i$, where $0 \leq i \leq L$.

Then the assertion holds for all n = i+1, that is for every set of i messages corresponding to different source states, since $Pd_i = \frac{k}{v}$.

So by induction on n, the assertion is true for all n = L+1.

Hence $b \geq \left(\frac{v}{k}\right)^{L+1}$, as required. ■

If a code meets, with equality, the appropriate bound for b above, then it is referred to as *optimal.*

2.8 Preview of codes constructed

The following table outlines the optimal and near optimal (L_S,L_A)-codes that we will construct, implement and analyze in terms of time and storage required.

Note that q is reserved to represent a prime power unless otherwise specified.

L_S	L_A	k	v	b
1	0	any integer	$k > v$	optimal
2	1	any odd integer	$q \equiv 1 \bmod 2k$	optimal
3	2	$q+1$, $q \equiv 3 \bmod 4$	q^2+1	3 times optimal
1	1	q	q^2	optimal
1	1	q+1	q^2+q+1	optimal
2	2	q+1, q is a Mersenne prime.	q^2+1	2 times optimal
2	2	q, q is a Fermat prime.	q^2+2q+1	optimal

Also, discussed are the following L_S-fold secrecy codes:

(1) the optimal 1-fold secrecy code with k and v arbitrary integers,

(2) the optimal 2-fold secrecy code with q source states and q messages, where q is odd,

and the following perfect disclosure codes:

(1) the code with $\frac{q^d-1}{q-1}$ source states, having $\frac{q(q^d-1)}{q-1}$ messages and q^d encoding rules, for which $Pd_0 = Pd_1 = \frac{1}{q}$,

(2) the optimal code with q+1 source states and q^2+q messages, where $Pd_i = \frac{1}{q}$ for $0 \le i \le (t-1)$, $t < q$.

Chapter 3 : Constructions for Optimal and Near-optimal Codes

3.1 Motivation

Combinatorial designs can be used to construct infinite classes of optimal and near-optimal codes. We look at the types of designs needed, and in so doing formalize the structures of the encoding arrays for secrecy and authentication codes. For more information on the designs used we recommend Hughes and Piper [35], Beth, Jungnickel and Lenz [11] or Street and Street [65]; or for cyclic designs, Baumert [6]. Projective and affine planes are discussed in Batten [5], Dembowski [22] and Hirschfeld [34].

3.2 Constructions for Perfect L_S-fold Secrecy Codes

Perpendicular arrays can be used to construct optimal perfectly L_S-fold secrecy codes. A *Perpendicular Array* $PA_\lambda(t,k,v)$ is a $\lambda\binom{v}{t}$ by k array A of the symbols $\{1,...v\}$ which satisfies the following properties.

(i) every row of A contains k distinct symbols.

(ii) for every t columns of A, and for any t distinct symbols, there are precisely λ rows r of A such that the t given symbols all occur in row r in the t given columns.

We need the following property of perpendicular arrays.

Theorem 3.2.1. (Kramer, Kreher, Rees and Stinson [38 Theorem 1.1]) If $0 \le t' \le t$ and $\binom{k}{t} \ge \binom{k}{t'}$, then a $PA_\lambda(t,k,v)$ is also a $PA_{\lambda_{t'}}(t',k,v)$, where $\lambda_{t'} = \dfrac{\lambda\binom{v-t'}{t-t'}}{\binom{t}{t'}}$. Hence $\binom{t}{t'}$ divides $\lambda\binom{v-t'}{t-t'}$.

Proof. Let A be a $PA_\lambda(t,k,v)$. Number the columns 1 to k. Let S' be a set of any t' (distinct) symbols. For any set T' of t' distinct columns, define I(T') to be the number of rows of A in which the t' symbols in the columns in T' are the symbols in S'. Now, for any set T of t columns, we get the following equation.

$$\sum_{T' \subseteq T,\ |T'|=t'} I(T') = \lambda\binom{v-t'}{t-t'}.$$

In this way we get $\binom{k}{t}$ equations in $\binom{k}{t'}$ unknowns. If $\binom{k}{t} \ge \binom{k}{t'}$, it can be shown that the system has a unique solution for every T', that is,

$$I(T') = \lambda \frac{\binom{v-t'}{t-t'}}{\binom{t}{t'}}.$$

Consequently, A is a $PA_{\lambda_{t'}}(t',k,v)$, with $\lambda_{t'}$ as defined above. ■

Note. We note that if $k \ge 2t-1$ then $\binom{k}{t} \ge \binom{k}{t'}$ for all $0 \le t' \le t$.

We can construct secrecy codes from perpendicular arrays in the following manner.

Theorem 3.2.2. (Stinson [60, Theorem 2.3]) If there exists a $PA_\lambda(t,k,v)$, where $k \ge 2t-1$, then there is a perfect t-fold secrecy code for k source states with v messages and $\lambda\binom{v}{t}$ encoding rules.

Proof. Let A be a $PA_\lambda(t,k,v)$. For each row $r = (x_1, \ldots, x_k)$ of A, define an encoding rule e_r by $e_r(s) = x_s$, where $1 \le s \le k$ is a source state. We use each encoding rule with probability $\dfrac{1}{\lambda\binom{v}{t}}$.

To prove that this is a perfect t-fold secrecy code, we prove that we have perfect t'-fold secrecy for all $0 \le t' \le t$.

Let $0 \le t' \le t$. Since $k \ge 2t-1$, $\binom{k}{t} \ge \binom{k}{t'}$; hence A is a $PA_{\lambda_{t'}}(t',k,v)$ by Theorem 3.2.1. Therefore, any set of t' messages corresponds equally often to every set of t' source states. Let S_1 be a set of t' source states, and M_1 be a set of t' messages.

Now, using the notation of chapter 2,

$$p(S_1 \mid M_1) = \frac{p(M_1 \mid S_1)\, p(S_1)}{p(M_1)} \quad \text{(by Bayes' Theorem)}$$

$$= \frac{\frac{\lambda_{t'}}{b} p(S1)}{\sum_{\{e:\, M1 \subseteq M(e)\}} p(e)\, p(f_e(M_1))}$$

$$= \frac{\frac{\lambda_{t'}}{b} p(S1)}{\frac{1}{b}\lambda_{t'}}$$

$$= p(S_1).$$

Hence we have perfect t'-fold secrecy, and that completes the proof. ■

Note: The secrecy code constructed in Theorem 3.2.2 above is optimal if and only if $\lambda = 1$ and $k = v$.

We go on to show the equivalence of optimal secrecy codes and perpendicular arrays with $\lambda = 1$ and $k = v$.

Theorem 3.2.3. (Stinson [60, Theorem 2.4], the case $L_S = 1$ is due earlier to Shannon [51, page 681]) If there exists an optimal L_S-fold secrecy code for k source states, then there exists a $PA_1(L_S,k,k)$.

Proof. Let e_0 be an encoding rule. Let M_1 be a set of L_S messages, such that $M_1 \subseteq M(e_0)$. Let S_1 be a set of L_S source states. As in the proof to Theorem 2.7.1, there is at least one encoding rule e_1 such that $S_1 = f_{e_1}(M_1)$. Since the code is optimal, b = \B(\A\CO1(k,L_S)), and e_1 is unique. Hence $M_1 \subseteq M(e)$ for all encoding rules e. Now, there are $\binom{k}{L_S}$ different L_S-subsets of the messages $M(e_0)$, each of which occurs in $\binom{k}{L_S}$ encoding rules. Conversely, each of the $\binom{k}{L_S}$ encoding rules contains $\binom{k}{L_S}$ different

L_S-subsets of messages. Therefore $M(e_0) = M(e)$ for every encoding rule e. So the encoding array is a $PA_1(L_S,k,k)$ on the symbols $M(e_0)$. ■

The next couple of results establish the existence of some 1-fold and 2-fold secrecy codes, (and the associated perpendicular arrays).

Theorem 3.2.4. (Stinson [60, Theorem 2.5]) For positive integers k, there exists a $PA_1(1,k,k)$.

That is, for all such k, there exists an optimal 1-secret code for k source states.

Proof. Any Latin square of order k is a $PA_1(1,k,k)$. A Latin square of order k always exists. For example, take the row 0, 1, ... , k-1 and develop it modulo k. Code 4.1 in chapter 4 is an implementation of this code. ■

Theorem 3.2.5. (Mullin, Schellenberg, van Rees and Vanstone [50, Corollary 2.5]) For all odd prime powers q, there exists a $PA_1(2,q,q)$. That is, for all such q, there exists an optimal code for q source states having perfect 2-fold secrecy.

Proof. This construction is outlined as code 4.2 in chapter 4. ■

Some constructions for t-fold secrecy codes where $t \geq 3$ come from t-homogeneous permutation groups.

A permutation group has *degree* n if it acts on a set S, $|S| = n$.

A permutation group is *t-homogeneous* if for all t-subsets S_1,S_2 of S, there are exactly the same number of permutations $\pi \in G$ such that $(S_1)\pi = S_2$.

The number of such π must be $\frac{|G|}{\binom{n}{t}}$.

Examples 3.2.6. (Biggs and White [13]) The symmetric group S_n on the set {1, ..., n} consists of all permutations of that set. This group is t-homogeneous on {1, ..., n}, for any $0 \leq t \leq n$. There are precisely t!(n-t)! distinct permutations that map one given t-set of elements onto another given t-set of elements.

The alternating group A_n on the set $\{1, ..., n\}$ consists of all even permutations of that set. (A permutation is even if it can be achieved by exchanging an even number of pairs of elements.) this group is (n-2)-homogeneous on$\{1, ..., n\}$. There are precisely (n-2)! distinct permutations that map one given (n-2)-set of elements onto another given (n-2)-set of elements.

For all prime powers q, the projective general linear group PGL(2,q) is 3-homogeneous on the (q+1) points of PG(1,q). There are precisely 6 distinct permutations that map a given 3-set of elements to another given 3-set of elements.

Theorem 3.2.7. (Stinson and Teirlinck [64, Theorem 3.1]) If G is a t-homogeneous permutation group of degree n, then there exists a $PA_\lambda(t,n,n)$ where

$$\lambda = \frac{|G|}{\binom{n}{t}}.$$

Hence there exists a t-code for n source states with n messages and |G| encoding rules.

Proof. Write down the permutations of G as the encoding array. ■

Examples 3.2.8.

(Biggs and White [13]) As in Example 3.2.6, if q is a prime power, PGL(2,q) is a 3-homogeneous permutations group that yields a $PA_6(3,q+1,q+1)$, and hence a 3-fold secrecy code that has q+1 source states, q+1 messages, (q-1)q(q+1) encoding rules which is 6 times the optimal number. S_n yields a $PA_{t!(n-t)!}(t,n,n)$ for any $1 \le t \le n$, and A_n yields a $PA_{(n-2)!}(n-2,n,n)$, neither of which yield codes with close to the optimal number of encoding rules.

So let's look at a couple of groups that generate optimal or close to optimal codes.

(Stinson and Teirlinck [64, Lemma 3.4]) AGL(1,8) and AΓL(1,32) are 3-homogeneous permutation groups, where $|AGL(1,8)| = \binom{8}{3}$ and $|A\Gamma L(1,32)| = \binom{32}{3}$. See Beth, Jungnickel and Lenz [11]. Hence there exists $PA_1(3,8,8)$ and $PA_1(3,32,32)$ and the associated optimal 3-fold secrecy codes.

(Stinson [61, Theorem 2.6]) PGL(2,8) and PΓL(2,32) are 4-homogeneous permutation groups that yield a $PA_4(4,9,9)$ and a $PA_4(4,33,33)$ respectively. That is, they yield codes that have four times the optimal number of encoding rules and perfect 4-fold secrecy.

Theorem 3.2.9. (Stinson and Teirlinck [64, Lemma 3.5]) If $q \equiv 3 \bmod 4$ is a prime power, then there exists a $PA_3(3,q+1,q+1)$, and hence a code with perfect 3-fold secrecy, $q+1$ source states and $\frac{q^3-q}{2}$ encoding rules, which is three times the optimal number.

Proof. The group PSL(2,q) is 3-homogeneous of degree $q+1$ if q is a prime number and $q \equiv 3 \bmod 4$. Apply theorem 3.2.7 to get a $PA_3(3,q+1,q+1)$, as required. This is code 4.3 from chapter 4. ■

3.3 Constructions for (L_S, L_S-1)-codes

We need our perpendicular arrays to have an extra property to construct codes with the additional property of authentication .

A $PA_\lambda(t,k,v)$, A, is an *authentication* PA and denoted $APA_\lambda(t,k,v)$ if, for all $t' \le t-1$ and for any $t'+1$ distinct symbols x_i $(1 \le i \le t'+1)$ we have that amongst all the rows of A which contain the symbols x_i $(1 \le i \le t'+1)$, the t' symbols x_i $(1 \le i \le t')$ occur equally often in all possible subsets of t' columns.

Theorem 3.3.1. (Stinson [61, page 16]) An $APA_\lambda(t,k,v)$ is also an $APA_{\lambda_{t'}}(t',k,v)$, for all $t' \le t$. Hence $\binom{k}{t'}$ divides $\lambda_{t'-1}\binom{k}{t'+1}$. ■

Note. If $k \ge 2t+1$, a $PA_\lambda(t,k,k)$ is also an $APA_\lambda(t,k,k)$.

Theorem 3.3.2. (Stinson [61, Theorem 3.3]) If an $APA_\lambda(t,k,v)$ exists, then there is a (t,t-1)-code with k source states, v messages, and $\lambda\binom{v}{t}$ encoding rules.

Proof. Let A be an $APA_\lambda(t,k,v)$. A code is constructed as in the proof to theorem 3.2.2. Suppose that the opponent observes the distinct messages x_i, $1 \le i \le t'$ in the channel, for some t', $0 \le t' \le t$, and then sends the message $x_{t'+1}$, where $x_{t'+1} \ne x_i$, $1 \le i \le t'$. The chance of sucessful deception is calculated to be

$$\frac{\sum_{\{e:\, x_i \in M(e),\, i=1,2,\ldots,t'+1\}} p(e)\, p(\{s_1, \ldots, s_{t'}\} = \{f_e(x_1), \ldots, f_e(x_{t'})\})}{\sum_{\{e:\, x_i \in M(e),\, i=1,2,\ldots,t'\}} p(e)\, p(\{s_1, \ldots, s_{t'}\} = \{f_e(x_1), \ldots, f_e(x_{t'})\})}$$

$$= \frac{\sum_{\{e:\, x_i \in M(e),\, i=1,2,\ldots,t'+1\}} p(\{s_1, \ldots, s_{t'}\} = \{f_e(x_1), \ldots, f_e(x_{t'})\})}{\sum_{\{e:\, x_i \in M(e),\, i=1,2,\ldots,t'\}} p(\{s_1, \ldots, s_{t'}\} = \{f_e(x_1), \ldots, f_e(x_{t'})\})} \qquad \text{(as p(e) is constant.)}$$

$$= \frac{\dfrac{\lambda_{t'+1}\binom{k}{t'+1}}{\binom{k}{t'}}}{\sum_{\{e:\, x_i \in M(e),\, i=1,2,\ldots,t'\}} p(\{s_1, \ldots, s_{t'}\} = \{f_e(x_1), \ldots, f_e(x_{t'})\})} \qquad \text{(the PA is an APA)}$$

$$= \frac{\lambda_{t'+1}\binom{k}{t'+1}}{\lambda_{t'}\binom{k}{t'}}$$

$$= \frac{(t'-1)\binom{k}{t'+1}}{(v-t')\binom{k}{t'}} \qquad \left(\text{as } \frac{\lambda_{t'+1}}{\lambda_{t'}} = \frac{t'-1}{v-t'} \text{ by theorem 3.2.1.}\right)$$

$$= \frac{k-t'}{v-t'}.$$

That is, $Pd_{t'} = \frac{k-t'}{v-t'}$, as required. ■

Note. This code is optimal if and only if $\lambda = 1$.

Theorem 3.3.3. (Stinson [63, Theorem 3.4]) For all positive integers v, and all positive integers $k \leq v$, there is an $APA_1(1,k,v)$, and hence an optimal (1,0)-code with k source states and v messages.

Proof. This is code 4.4 described in chapter 4. ■

Examples 3.3.4.

There is an $APA_1(2,3,v)$ if and only if $v \geq 7$ is odd (Stinson [60] and Stinson and Teirlinck [64]).

There is an $APA_1(2,5,v)$ if and only if $v \equiv 1$ or 5 mod 10, $v \geq 11$, $v \neq 15$ (Lindner and Stinson [37]).

Theorem 3.3.5. (Granville, Moisadis and Rees [30]) If k is odd and $q \equiv 1 \bmod 2k$ is a prime power, then there exists an $APA_1(2,k,q)$, and hence an optimal (2,1)-code with k source states and q messages.

Proof. This is code 4.5 from chapter 4. ■

Example 3.2.8 told us how to construct the following perpendicular arrays from multiply transitive groups; $PA_1(3,8,8)$, $PA_1(3,32,32)$, $PA_4(4,9,9)$ and $PA_4(4,33,33)$.

We apply theorem 3.3.1 to get the following examples.

Examples 3.3.6. (Stinson [60, Theorem 3.7])

There is an $APA_1(3,8,8)$ and an $APA_1(3,32,32)$.

There is an $APA_4(4,9,9)$ and an $APA_4(4,33,33)$.

Theorem 3.3.7. (Stinson [60, theorem 3.7]) If $q \equiv 3 \bmod 4$ is a prime power, $q \geq 7$, then there is an $APA_3(3,q+1,q+1)$, and hence a (3,2)-code with q+1 source states, q+1 messages and three times the optimal number of encoding rules.

Proof. We constructed a $PA_3(3,q+1,q+1)$ in theorem 3.2.9. If $q \geq 7$ then we can apply theorem 3.3.1. This is code 4.6 from chapter 4. ■

Having an authentication code with the same number of messages as encoding rules doesn't really do us much good, since a successful deception is automatically assured. So we use the trick in the following theorem to expand our codes.

Theorem 3.3.8. (Stinson and Teirlinck [64, Theorem 3.2]) If there is a t-design $S_{\lambda'}(t,k,v)$ and an $APA_\lambda(t,k,k)$ then there is also an $APA_{\lambda\lambda'}(t,k,v)$.

Proof. Each block in the t-design $S_{\lambda'}(t,k,v)$ is a set of k elements. An $APA_\lambda(t,k,k)$ is constructed on each block. The union of all the rows from all the $APA_\lambda(t,k,k)$ is an $APA_{\lambda\lambda'}(t,k,v)$. ■

So we find some t-designs with small λ. Some of these are discussed by Hughes and Piper in [35], and Beth, Jungnickel and Lenz in [11].

An *inversive geometry* is a 3-design $S_1(3,q+1,q^d+1)$.

Theorem 3.3.9. (Beth, Jungnickel and Lenz [11]) If q is a prime power and d is a positive integer, then an inversive geometry $S_1(3,q+1,q^d+1)$ exists.

Proof. See Beth, Jungnickel and Lenz [11], for a formal proof, or code 4.7 in chapter 4 for an outline of the construction of the inversive geometry. ■

Theorem 3.3.10. (Stinson and Teirlinck [64, Theorem 3.5]) If $q \equiv 3 \bmod 4$ is a prime power, $q \geq 7$, and d is a positive integer, there is an $APA_3(3,q+1,q^d+1)$, and hence a (3,2)-code with q+1 source states, q^d+1 messages, and three times the optimal number of encoding rules.

Proof. Let the lines of the inversive geometry $S_1(3,q+1,q+1)$ be blocks. There is a $PA_3(3,q+1,q+1)$ (by theorem 3.2.9). We combine these using theorem 3.3.8. See code 4.7 in chapter 4 for an outline of the construction. ■

In Examples 3.3.6 we showed an $APA_1(3,8,8)$ and an $APA_1(3,32,32)$. Combining these with the projective geometries $S_1(3,8,8)$ and $S_1(3,32,32)$ we get the following.

Examples 3.3.11. (Stinson and Teirlinck [64]) If d is a positive integer, there exists an $APA_1(3,8,7^d+1)$ and an $APA_1(3,32,31^d+1)$, and hence an optimal (3,2)-codes with 8 source states and 7^d+1 messages and 31 source states and 31^d+1 messages.

3.4 Constructions for (L_S,L_S)-codes

We can combine a perpendicular array and a t-design to construct a code that has the same authentication level and secrecy level.

Theorem 3.4.1. (Stinson [60, Theorem 4.2]) If there exist a $PA_\lambda(t,k,k)$ and a (t+1)-design $S_{\lambda'}(t+1,k,v)$ where $k \geq 2t-1$, then there is a (t,t)-code for k source states, for an arbitrary source state probability distribution, having v messages and $\frac{\lambda\lambda'(v-t)}{k-t}\binom{v}{t}$ encoding rules.

Proof. Each block in the (t+1)-design $S_{\lambda'}(t+1,k,v)$ is a set of k elements. A $PA_\lambda(t,k,k)$ is constructed on each of these blocks. The union of all the rows from all the $PA_\lambda(t,k,k)$ is a

$PA_{\lambda\lambda'}(t,k,v)$. Now $k \geq 2t-1$, and hence the resulting code has perfect t-fold secrecy. Also, the number of encoding rules is clearly $\frac{\lambda\lambda'(v-t)}{k-t}\binom{v}{t}$.

So, it remains to verify that the code is t-fold secure against spoofing. Now $k \geq 2t-1$, so the $PA_\lambda(t,k,k)$ is also an $APA_\lambda(t,k,k)$, as we remarked earlier. Then the code is (-1)-secure against spoofing , by theorems 3.3.2 and 3.3.8. So, to prove that the code is t-fold secure against spoofing, suppose the opponent observes the distinct messages x_i, $1 \leq i \leq t$, in the channel. Suppose that the opponent places the further distinct message x_{t+1} in the channel. Then, by a similar argument to the proof of theorem 3.3.2, it can be shown that the chance of sucessful deception is $\frac{\lambda'}{\lambda} = \frac{k-t}{v-t}$. Hence, $Pd_t = \frac{k-t}{v-t}$. ■

Note that the code constructed in theorem 3.3.1 above is optimal if and only if $\lambda = \lambda' = 1$. We use this construction in the following theorems to construct infinite classes of optimal (1,1)-codes and near-optimal (2,2)-codes (for an arbitrary source state probability distribution).

Theorem 3.4.2. (Stinson [63, Corollary 3.11]) If there exists a $S_1(2,k,v)$, (i.e. a (v,k,1)-BIBD), then there also exists an optimal (1,1)-code for an arbitrary source state distribution., with v source states and v messages.

Proof. Theorem 3.2.4 states that a $PA_1(1,k,k)$ exists for all k. ■

The infinite classes of $S_1(2,k,v)$ that spring most readily to mind are the affine geometries $S_1(2,q,q^d)$ and the projective geometries $S_1(2,q+1,\frac{q^{d+1}-1}{q-1})$, for all prime powers q and all positive integers d. This leads us to theorems 3.4.3 and 3.4.4 below.

Theorem 3.4.3. (Stinson [60, Theorem 4.4]) For all prime powers q and all positive integers d, there exists an optimal (1,1)-code for an arbitrary source state probability distribution, with q source states and q^d messages.

Proof. We take the lines of the affine geometry of order q and dimension d as blocks. We expand each block into a $PA_1(1,k,k)$, which we have shown exists (see theorem 3.2.4). This is code 4.8 in chapter 4. ■

Theorem 3.4.4. (Stinson [60]) For all prime powers q and all positive integers d , there exists an optimal (1,1)-code for an arbitrary source state probability distribution, with q+1 source states and $\frac{q^{d+1}-1}{q-1}$ messages.

Proof. We take the lines of the projective geometry of order q and dimension d as blocks. We expand each block into a $PA_1(1,k,k)$, which we have shown exists (see theorem 3.2.4). This is code 4.9 in chapter 4.

To construct (2,2)-codes we need the concept of an orthogonal array.

An *orthogonal array* OA(k,v) is a v^2 by k array, A, of the symbols {1, ... ,v}, such that

> for any t columns $c_1,..., c_t$ of A, and for any t distinct symbols $x_1,..., x_t$, there is a unique row r of A such that x_i occurs in column c_i of row r, for $1 \le i \le t$.

Now, for any prime power q, an OA(q,q) can be constructed from the affine plane of order q, with the property that for each element α of GF(q), there is a row that contains α repeated q times. If these q rows are deleted, a $PA_2(2,q,q)$ remains. This construction is exemplified in code 4.9 in chapter 4.

The following constructions for (2,2)-codes require finite fields GF(n) and GF(n-1) and so use Fermat and Mersenne primes.

Theorem 3.4.5. (Stinson [60, Theorem 4.5]) For any Mersenne prime q, there is a (2,2)-code for an arbitrary source state probability distribution with q+1 source states and $\frac{(q^d+1)(q^d)(q^d-1)}{q-1}$ messages. This code has twice the optimal number of encoding rules.

Proof. We know that a $S_1(3,q+1,q^d+1)$ exists. We also know that an OA(q+1,q+1) can be constructed from the affine plane of order q+1. Apply theorem 3.4.1. This construction is outlined as code 4.9 in chapter 4. ■

Theorem 3.4.6. (Stinson [60, Theorem 4.6]) For any Fermat prime q, there is a (2,2)-code for an arbitrary source state probability distribution with q source states and $(q-1)^d+1$ messages. This code is optimal.

Proof. There is a $PA_1(2,q,q)$ and an $S_1(3,q,(q-1)^d+1)$. Apply theorem 3.4.1. ■

Note. This may not be an infinite class of codes, since only a finite number of Fermat primes are known.

3.5 Constructions for Perfect Disclosure Codes

We construct optimal perfect disclosure codes using transversal designs.

A *transversal design* $TD_\lambda(t,k,n)$ is a triple (X, G, A), where X is a set of kn *points*, G is a partition of X into k *groups* of n points each, and A is a set of λn^t *blocks* each of which meets each group in a point, such that every t-subset of points from distinct groups occurs in exactly λ blocks.

Theorem 3.5.1. (Stinson [60, Theorem 5.3], the special case $t = 2$ and $\lambda = 1$ is due to Brickell [16]) If there exists a transversal design $TD_\lambda(t,k,n)$, then there exists a perfect disclosure code for k source states, having kn messages and λn^t blocks, and for which $Pd_i = \frac{1}{n} \left(= \frac{k}{v}\right)$, for $0 \le i \le t-1$. The code is optimal if and only if $\lambda = 1$.

We have a partial converse to theorem 3.5.1, that gives us a transversal design corresponding to every optimal perfect disclosure code.

Theorem 3.5.2. (Stinson, [60, Theorem 5.4]) Suppose we have an optimal perfect disclosure code for k source states, v messages and $\left(\frac{v}{k}\right)^t$ encoding rules. Suppose further that, in this code, $Pd_i = \frac{k}{v}$ for $0 \le i \le t-1$. Then, there exists a transversal design $TD_1(t,k,n)$, where $n = \frac{v}{k}$.

Proof. Now, every set M of t messages corresponding to distinct source states appears in at least one encoding rule, as in the proof to theorem 2.7.5. Since the code is optimal, M must appear in exactly one encoding rule. Thus we have a $TD_1(t,k,n)$, where $n = \frac{v}{k}$. ■

Theorem 3.5.3. (Hanani [31, Lemma 3.5]) If q is a prime power and $t \le q$ is an integer, there is a $TD_1(t,q+1,q+1)$, and hence an optimal perfect disclosure code with q+1 source states and q^2+q+1 messages, and for which $Pd_i = \frac{1}{q} = \frac{k}{v}$, for $0 \le i \le t-1$.

Proof. This is code 4.11 in chapter 4. ■

The code guaranteed by theorem 3.5.2 allows us to use the same encoding rule up to q-1 times (as opposed to the two or three times our other infinite classes of codes allowed us).

Theorem 3.5.4. (Stinson [63, Theorem 3.7]) If q is a prime power and $d \geq 2$ is an integer, there is a $TD_{q^d-2}(2, \frac{q^d-1}{q-1}, q)$, and hence a perfect disclosure code with $\frac{q^d-1}{q-1}$ source states, $\frac{q(q^d-1)}{q-1}$ messages and q^d encoding rules, and for which $Pd_i = \frac{1}{q}$ for $i = 0$ and $i = 1$.

Proof. This is code 4.12 in chapter 4. ■

Chapter 4 : Implementations

4.1 Motivation

Optimal and near-optimal secrecy and authentication codes are good in theory, but if they cannot be implemented efficiently, then the effort to make the number of encoding rules as small as possible is somewhat wasted. Storing an entire encoding array for a code is not going to be possible once the code gets large. However, we also have to ensure that the time taken to encode source states and decode messages does not become too large. This chapter outlines pseudo-code algorithms for the infinite classes of codes that were found in Chapter 3.

4.2 Finite Field Arithmetic

These implementations are going to use finite field arithmetic, so we include here a very brief note on operations in $GF(p^n)$, the finite (or Galois) field of characteristic p and degree n. We assume here that the reader is familiar with the definition of a finite field, and that it always has a prime power order. GF(p), where p is prime, can be constructed by considering the arithmetic operations modulo p. $GF(p^n)$ can be considered as a n-tuple of elements of GF(p). $GF(p^n)$ is most commonly (but not always, see chapter 5) considered as the set of polynomials with coefficients in GF(p) modulo some nth degree polynomial irreducible over GF(p). Every finite field has a *generator* α, such that every element of the field except the zero element can be represented as a power of α. If x is an element of GF(q), then $x^{q-1} = 1$, the multiplicative identity. So every non-zero element

could also be considered as a power of α between 0 and q-2. Every element except 0 has a multiplicative inverse. So addition, subtraction, multiplication and non-zero division are all defined. From multiplication we define exponentiation in the obvious way. We also define a finite logarithm in the following manner. If α is a primitive element of the GF(q), and x is any non-zero element of GF(q), then the logarithm of x to the base α, $\log_\alpha x$ is the power (between 0 and q-1) to which α must be raised to get x. In other words, the following two statements are equivalent for x and α as defined above and the integer y, $0 \le y \le q-1$:

$$\log_\alpha x = y$$

and $$x = \alpha^y.$$

Example 4.1. Consider the finite field $GF(2^3)$. The elements can be represented as {0,1} polynomials modulo $f(x) = x^3+x+1$. The element α, where $f(\alpha) = 0$, is a generator of the field. The element 0 is not a power of α, but the rest of the field is demonstrated in the table below.

α^i	i
1	0
x	1
x^2	2
x+1	3
x^2+x	4
x^2+x+1	5
x^2+1	6

Then, for example, $\log_\alpha(x^2+x) = 4$.

Another thing we wish to do in finite fields is select a random element from an equiprobable distribution. Each element would be therefore be selected with probability $\frac{1}{q}$.

Also, we need to be able to impose an ordering on the elements of a finite field. This can be done in a number of ways. To order elements over GF(p), represented as integers modulo p, it is probably easiest to order the field 0, 1, 2, ..., p-1, although the field could also be ordered 0, α^0, α^1, ..., α^{p-2}, where α is a primitive element. To order the elements of GF(p^n), when we have an ordering of GF(p), and each element of GF(p^n) is represented as an ordered n-tuple of elements of GF(p) with respect to some basis, we could order the n-tuples lexicographically. Else we could order GF(p^n) as powers of a primitive element.

Example 4.1. (part 2). Suppose we have the ordering 0, 1 on GF(2), and we wish to order GF(8), as above.

Then, the lexicographic ordering is 0, 1, x, x+1, x^2, x^2+1,x^2+x, x^2+x+1.

Or, as ordered 3-tuples with respect to the basis (α^2, α, 1):

(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1).

Alternatively, if ordered by powers of the generating element α, the ordering is as per the table above.

See chapter 5 for a detailed look at algorithms, costs and running times of finite field operations.

A final technical note before we begin: all random elements are chosen from an equiprobable distribution unless otherwise specified.

Code 4.1

An optimal code having perfect 1-fold secrecy, k source states, k messages and k encoding rules, for any integer k.

See Code 4.4, the (1,0)-code, and put v = k.

Code 4.2

An optimal code for q source states, q messages, $\frac{q(q-1)}{2}$ encoding rules and perfect 2-fold secrecy where q is an odd prime power.

Let GF(q) have primitive element β.

The encoding array is made up of the rows

$$x \qquad x+\beta^n \qquad x+\beta^{n+1} \qquad x+\beta^{n+2} \qquad \ldots \qquad x+\beta^{n+q-2}$$

for each $x \in GF(q)$, and each $0 \le n \le \frac{q-3}{2}$.

The source states are numbered 0 to q-1, and the messages are elements of GF(q).

Example 4.2. Consider q = 5. Then $\beta = 2$ is a primitive element of GF(5). The encoding array is as follows. We index the encoding rules by x and n.

encoding rule		source state				
x	n	0	1	2	3	4
0	0	0	1	2	4	3
0	1	0	2	4	3	1
1	0	1	2	3	0	4
1	1	1	3	0	4	2
2	0	2	3	4	1	0
2	1	2	4	1	0	3
3	0	3	4	0	2	1
3	1	3	0	2	1	4
4	0	4	0	1	3	2
4	1	4	1	3	2	0

or equivalently, the two rows corresponding to x = 0 developed modulo 5.

Algorithm 4.2.

The key source.

Step 1

Choose a random n, $0 \leq n \leq \frac{q-3}{2}$.

Step 2.

Choose a random x, $x \in GF(q)$.

Step 3.

Send (n,x) to the transmitter and the receiver by a secure channel.

The number of bits sent by the key source is $\lceil \log_2 q \rceil + \lceil \log_2 \left(\frac{q-1}{2}\right) \rceil \approx \log_2 \left(\frac{q^2-q}{2}\right)$ bits, which is approximately $\log_2 b$ bits or about 2 bits of key for every bit of message.

The transmitter (sending source state i).

Step 1.

If i = 0 then message := x.

else message := $x+\beta^{n+(i-1)}$.

Step 2.

Send message to receiver via an open channel.

The receiver.

Step 1.

If the message is not an element of GF(q) then reject the message.

If message = x then the source state is 0.

If the message $\neq$ x then the source state is

$[(\log_\beta(\text{message}-x) - n) \bmod (q-1)] + 1$.

Code 4.3

A perfect 3-fold secret code with q+1 source states and $\frac{(q+1)q(q-1)}{2}$ encoding rules, which is three times the optimal number, where $q \equiv 3 \bmod 4$ is a prime power.

This is the (3,2) code , i.e. code 4.6.

Code 4.4

An optimal (1,0)-code with k source states, v messages and v encoding rules, where k and v are integers, with $v \geq k$.

The encoding array is the row

0 1 2 ... k-1

developed modulo v.

The source states are numbered from 0 to k-1 and the messages are integers between 0 and v-1.

Example 4.3. When k = 4 and v =7 we get

	0	1	2	3
e_0	0	1	2	3
e_1	1	2	3	4
e_2	2	3	4	5
e_3	3	4	5	6
e_4	4	5	6	0
e_5	5	6	0	1
e_6	6	0	1	2

Notice that each possible message occurs once in each column.

Algorithm 4.4.

The key source.

Step 1.

Select a random integer n, $0 \leq n \leq v-1$.

Step 2.

Send n to transmitter and receiver via a secure channel.

There are $\lceil \log_2 v \rceil = \lceil \log_2 b \rceil$ bits sent by the key source, or about 1 bit of key for every bit of message.

The transmitter (sending source state i).

Step 1.

message := (i+n) mod v.

Step 2.

Send message to receiver by open channel.

The receiver.

If message is not an element of {0, ... ,v-1} then reject message.

Step 1.

x := (message-n) mod v.

Step 2.

If $0 \leq x \leq k-1$ then the source state is x else reject the message.

Code 4.5

An optimal (2,1) code with k source states, q messages and $\frac{q(q-1)}{2}$ encoding rules, where k is odd and $q \equiv 1 \bmod 2k$ a prime power.

Let β be a primitive element of GF(q).

Let $\alpha = \beta^{\frac{q-1}{k}}$, a kth root of unity.

The encoding array has as its rows

$$\omega + \beta^x\alpha^y \quad \omega + \beta^x\alpha^{1+y} \quad \ldots \quad \omega + \beta^x\alpha^{(k-1)+y}$$

for all $\omega \in$ GF(q), all integers x such that $1 \le x \le \frac{q-1}{2k}$, and all integers y such that $0 \le y \le k-1$.

Without using α explicitly, the row becomes,

$$\omega + \beta^{x+\frac{(q-1)y}{k}} \quad \omega + \beta^{x+\frac{(q-1)(1+y)}{k}} \quad \ldots \quad \omega + \beta^{x+\frac{(q-1)((k-1)+y)}{k}}$$

Example 4.5. When k = 3 and q=13, we note first that 3 is odd and that 13 is a prime congruent to 1 mod 6. $\beta = 2$ is a generator of GF(13), and $\alpha = \beta^4 = 3$. The adding of ω causes the development mod q, and the changing y values, with x and ω held constant causes the rotation of the rows.

The encoding array consists of each of the following rows developed mod 13.

x	y	rows		
1	0	2	6	5
1	1	6	5	2
1	2	5	2	6
2	0	4	12	10
2	1	12	10	4
2	2	10	4	12

Notice that, in this case, {2,6,5} and {4,10,12} are a pair of complementary difference sets, so that the resulting array is a cyclic BIBD with $\lambda = 1$, with each row expanded into a Latin square. This happens whenever $\frac{q-1}{2k} = 2$, since the two x values correspond to quadratic residues and quadratic non-residues. For $\frac{q-1}{2k} = 1$, the "starter row" is a difference set, and the resulting array is also a cyclic BIBD with $\lambda = 1$, with each row expanded into a Latin square.

Algorithm 4.5.

The key source.

Step 1.

Choose a random integer x, $1 \leq x \leq \frac{q-1}{2k}$.

Step 2.

Choose a random y, $0 \leq y \leq k-1$.

Step 3.

Choose a random $\omega \in GF(q)$.

Step 4.

Send (x, y, ω) to transmitter and receiver via secure channel.

The key source transmits $\lceil \log_2 \left(\frac{q-1}{2k}\right) \rceil + \lceil \log_2 k \rceil + \lceil \log_2 q \rceil \approx \log_2 \left(\frac{(q-1)q}{2}\right) = \log_2 b$ bits, or about 2 bits of key for every bit of message.

The transmitter (sending source state i).

Step 1.

message := $\omega + \beta^{\frac{q-1}{k}(y+i)+x}$

Step 2.

Send message to receiver via a secure channel.

The receiver.

If the message is not in GF(q) then reject message.

Step 1.

If message = ω then reject message.

Step 2.

temp := $(\log_\beta(\text{message}-\omega) - x) \bmod (q-1)$.

Step 3.

If $\frac{q-1}{k}$ divides temp exactly (that is, if $\beta^{temp} \in$ the subfield GF(q)),

then source state is $\left(\frac{k(temp)}{q-1} - y\right) \bmod k$

else reject the message.

Code 4.6

A (3,2)-code with q+1 source states, q+1 messages and $\frac{(q+1)q(q-1)}{2}$ encoding rules, which is three times the optimal number, where q ≡ 3 mod 4 is a prime power.

The construction utilizes PSL(2,q), the special linear group on q+1 points, GF(q)∪{∞}. Note that PSL(2,q) is a 3-homogeneous permutation group. Note also that $|PSL(2,q)| = \frac{q^3-q}{2}$.

The permutations of PSL(2,q) written as the rows of an array, form a (3,2)-code.

These permutations can be defined on GF(q) ∪ {∞} by the mappings

$x \rightarrow \frac{ax+b}{cx+d}$, ad-bc is a non-zero square in GF(q),

for each a, b, c, d in GF(q), with sensible rules about ∞.

More specifically, when $c \neq 0$, if $x = \frac{-d}{c}$, then $x \rightarrow \infty$, and if $x = \infty$, then $x \rightarrow \frac{a}{c}$.

If $c = 0$, then $\infty \rightarrow \infty$, that is, ∞ is fixed.

Note that the mappings defined by (a,b,c,d) and (ka,kb,kc,kd) are the same for any $k \in GF(q) \setminus \{0\}$. Also, if $a = 0$, then $b \neq 0$. So the elements of PSL(2,q) are mappings defined by (1,b,c,d), where d-bc is a non-zero square, and (0,1,c,d), where -c is a non-zero square. To choose a non-zero square equiprobably, an element of $GF(q) \setminus \{0\}$ is chosen equiprobably, and squared. The source states are numbered from 0 to q and the messages are elements of $GF(q) \cup \{\infty\}$.

Example 4.6. For q = 3 we get

a	b	c	d	permutation in PSL(2,q)			
1	0	0	1	0	1	2	∞
1	0	1	1	0	2	∞	1
1	0	2	1	0	∞	1	2
1	1	0	1	1	2	0	∞
1	1	1	2	2	∞	0	1
1	1	2	0	∞	1	0	2
1	2	0	1	2	0	1	∞
1	2	1	0	∞	0	2	1
1	2	2	2	1	0	∞	2
0	1	2	0	∞	2	1	0
0	1	2	1	1	∞	2	0
0	1	2	2	2	1	∞	0

Note that every unordered triple occurs the same number of times in each set of three columns. Within those triples, each unordered pair occurs in precisely one of the three possible pair of columns. Note also that the probability of a successful impersonation attack is $\frac{k}{v} = 1$, and the probability of a sucessful substitution $\frac{k-1}{v-1} = 1$. So the code fulfills

the conditions for authentication, but a spoofing or impersonation attack is always successful.

Algorithm 4.6.

The key source.

Step 1.

Chose a random real number r, $0 < r < 1$, from a uniform distribution.

If $r < \frac{1}{q+1}$ then

a := 0.

b := 1.

d := a random element of GF(q).

m := a random element of GF(q)\{0}.

c := $- m^2$.

If $r > \frac{1}{q+1}$ then

a := 1

b := a random element of GF(q).

c := a random element of GF(q).

m := a random element of GF(q)\{0}.

d := $m^2 + bc$.

Step 2.

Send (a,b,c,d) to transmitter and receiver via a secure channel.

Number of bits sent is $3\lceil \log_2 q \rceil + 1$, or about 3 bits of key for every bit of message. This is only slightly more than the $\lceil \log_2 b \rceil \approx 3\log_2 q$ bits required if the number of the encoding rule was transmitted.

The transmitter (sending source state i).

Step 1.

If i = q then x := ∞, else x := the ith element of GF(q).

Step 2.

If c = 0 and x = ∞ then message := ∞.

If c = 0 and x $\neq \infty$ then message := $\frac{ax+b}{d}$.

If c $\neq$ 0 and x = ∞ then message := $\frac{a}{c}$.

If c $\neq$ 0 and x = $\frac{d}{c}$ then message := ∞.

If c $\neq$ 0 and x $\neq \infty$ and x $\neq \frac{d}{c}$ then message := $\frac{ax+b}{cx+d}$.

Step 3.

Send message to receiver via an open channel.

The receiver.

Step 1. { x := $\frac{b-d(message)}{c(message)-a}$.}

If c = 0 and message = ∞ then x := ∞.

If c = 0 and message $\neq$ 0 then x:= $\frac{b-d(message)}{c}$.

If c $\neq$ 0 and message = ∞ then x:= $\frac{-d}{c}$.

If c $\neq$ 0 and message $\neq \infty$ or $\frac{a}{c}$ then x := $\frac{b-d(message)}{c(message)-a}$.

If c $\neq$ 0 and message = $\frac{a}{c}$ then x := ∞.

Step 2.

If x = ∞ then source state is q.

If x $\neq \infty$ then the source state is j, where x is the jth element of GF(q).

Code 4.7

A (3,2)-code with q+1 source states, q^d+1 messages, and $(q^d+1)q^d(q^d-1)$ encoding rules which is six times the optimal number where q is a prime power and d is any positive integer.

Let $GF(q^d)$ have primitive element β.

Then there is a subfield, GF(q) with primitive element $\beta^{\frac{q^d-1}{q-1}} = \alpha$. It makes sense, therefore, to talk about adding or multiplying elements from these two different fields together.

In theorem 3.3.10 we promised a code with three times the optimal number of encoding rules. Its encoding array is based on the set of blocks of an inversive plane, $S_1(3,q+1,q^d+1)$. Each block is expanded into a $APA_3(3,q+1,q+1)$ by writing the permutations of PSL(2,q) acting on each block.

The initial block of the $S_1(3,q+1,q^d+1)$ is $GF(q)\cup\{\infty\}$. Each block of $S_1(3,q+1,q^d+1)$ is the image of this initial block under an element of $PGL(2,q^d)$, the general linear group on $q^d + 1$ elements. That is, the permutations of $GF(q^d)\cup\{\infty\}$, defined by

$$x \rightarrow \frac{ex+f}{gx+h}, \qquad e,f,g,h \in GF(q^d),\ eh-fg \neq 0.$$

Each block is obtained from (q-1)q(q+1) different permutations, since PGL(2,q) is the subgroup of $PGL(2,q^d)$ that fixes $GF(q)\cup\{\infty\}$. So to choose a block, a permutation in $PGL(2,q^d)$ is chosen from an equiprobable distribution. Note that (e,f,g,h) generates the same permutation as $(\alpha e,\alpha f,\alpha g,\alpha h)$, $\alpha \in GF(q^d)\backslash\{0\}$. Note also that if e = 0, then $f \neq 0$. So the permutations in PGL(2,q) can be written as (1,f,g,h), for $h \neq fg$, and (0,1,g,h), for $g \neq 0$.

However, if we pick a base block {0, 1, ..., q-1, ∞}with a fixed ordering and let our encoding rules be indexed by the appropriate (e,f,g,h) then we run into a problem. Each

of the (q-1)q(q+1) permutations that define a given block will define it in a different order. Fortunately, all is not lost. We can still construct a code with six times the optimal number of encoding rules, which has the added advantage of being easy to implement. (No-one wants to have to sort each block.) The set of permutations of $PGL(2,q^d)$ is the set of blocks of $S_1(3,q+1,q^d+1)$, with each block expanded into an $APA_6(3,q+1,q^d+1)$. This is so because $PGL(2,q^d)$ is sharply 3-transitive, so that each ordered 3-tuple of elements occurs precisely once in each (ordered) set of three columns.

Choosing a random encoding rule, then, is a random permutation from $PGL(2,q^d)$, of $GF(q^d)\cup\{\infty\}$.

Algorithm 4.7.

The key source.

Step 2. {Define a permutation, ψ, to give a block of $S_1(3,q+1,q^d+1)$ }

Choose a random real number r, $0 < r < 1$, from a uniform distribution.

If $r < \frac{1}{q^d-1}$, then

e := 0.

f := 1.

g := a random element of $GF(q^d)\backslash\{0\}$.

h := a random element of $GF(q^d)$.

If $r > \frac{1}{q^d-1}$, then

e := 1.

f := a random element of $GF(q^d)$.

g := a random element of $GF(q^d)$.

h := a random element of $GF(q^d)\backslash\{bc\}$.

Step 4.

Send (e,f,g,h) to transmitter and receiver via a secure channel.

The number of bits sent is $3\lceil d\log_2 q \rceil + 1 \approx \log_2 b$, or 3 bits of key for every bit of message.

The transmitter (sending source state i).

Step 1.{Find the ith element of the initial block.}

If i = q then x := ∞.

If i = q-1 then x := 0.

If i ≠ q, and i ≠ q-1 then x := the ith element of GF(q).

Step 2.{message := $\frac{ex+f}{gx+h}$.}

If c = 0 and x = ∞ then message := ∞.

If c = 0 and x ≠ ∞ then message := $\frac{ex+f}{h}$.

If c ≠ 0 and x = ∞ then message := $\frac{e}{g}$.

If c ≠ 0 and x = $\frac{h}{g}$ then message := ∞.

If c ≠ 0 and x ≠ ∞ and x ≠ $\frac{d}{c}$ then message := $\frac{ex+f}{gx+h}$.

Step 3.

Send message to receiver by an open channel

The receiver.

Step 1. $\{ x := \frac{f\text{-}h(message)}{g(message)\text{-}e} .\}$

If $c = 0$ and message $= \infty$ then $x := \infty$.

If $c = 0$ and message $\neq 0$ then $x := \frac{f\text{-}h(message)}{g}$.

If $c \neq 0$ and message $= \infty$ then $x := -\frac{h}{g}$.

If $c \neq 0$ and message $= \frac{e}{g}$ then $x := \infty$.

If $c \neq 0$ and message $\neq \infty$ and message $\neq \frac{e}{g}$ then $x := \frac{f\text{-}h(message)}{g(message)\text{-}e}$.

Step 2.

If $x \notin GF(q) \cup \{\infty\}$ then reject message.

Step 3.

If $x = \infty$ then source state is q.

If $x \neq \infty$ then the source state is j, where x is the jth element of GF(q).

Code 4.8

An optimal (1,1)-code with q source states, q^2 messages and $q^2(q+1)$ encoding rules, where q is a prime power.

Let GF(q) have primitive element β. The encoding array is based on the affine plane of order q. Each block of the affine plane is expanded into a Latin square by a series of cyclic shifts.To construct an affine plane, consider $GF(q) \times GF(q) = \{(\alpha,\beta) \mid \alpha,\beta \in GF(q)\}$. Take lines of slope α, for every $\alpha \in GF(q)$. More specifically, each line is a set $\{(x,y) \mid y = \alpha x+\beta\}$, $\beta \in GF(q)$. Add the lines of slope ∞, $\{(x,y) \mid x = \beta\}$, for each β.

Our messages are (α,β) pairs, (where $\alpha,\beta \in GF(q)$).

Example 4.8.1. Consider $GF(3) = \{0,1,2\}$.

Then $GF(3) \times GF(3) = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$.

We abbreviate (a,b) to ab for ease of reading.

The lines of the affine plane are enumerated in the table below.

slope 0	slope 1	slope 2	slope ∞
00,10,20	00,11,22	00,12,21	00,01,02
01,11,21	01,12,20	01,10,22	10,11,12
02,12,22	02,10,21	02,11,20	20,21,22

The points in each block of AG(2,q) need some ordering so we can apply a series of cyclic shifts to them to get a Latin square. The lines with slope $\alpha \in$ GF(q) are ordered by the first element of the pair where some ordering is defined on GF(q). The blocks of slope ∞ are ordered by the second element (since the first elements are all the same).

Example 4.8.2. Consider PG(2,3) (over GF(3)). Let GF(3) be ordered 0, 1, 2. Then the block with slope 2 through 11 is ordered 02, 11, 20. The block of slope ∞ through 11 in PG(2,3) as in Example 4.8.1 is ordered 10, 11, 12.

Each block gives rise to q rows in the encoding array.

Example 4.8.3. Consider PG(2,3) as constructed in Example 4.8.1.

The 3 rows determined by the block 02, 11, 20 are

row 1	02	11	20
row 2	11	20	02
row 3	20	02	11

We select the block by choosing coefficients in the equation $\alpha x_1+\beta x_2+\gamma = 0$.

The transmitter calculates message = (x_1,x_2) which lies on the line $\alpha x_1+\beta x_2+\gamma = 0$.

To send source state s_i, where $0 \le i \le q-1$, we do the following.

If $\beta \neq 0$ then x_1 := the (i+n mod q)th element of GF(q),

and $x_2 := \frac{-\alpha.x_1 - \gamma}{\beta}$.

If $\beta = 0$ then $x_1 := \frac{\gamma}{\alpha}$,

and x_2 := the (i+n mod q)th element of GF(q).

Algorithm 4.8.

The key source.

Step 1.

Choose a random real number r, $0 < r < 1$, from a uniform distribution.

Step 2.

If $r < \frac{1}{q+1}$ then do **Step 2a.**

Step 2a.

$\alpha := 0$,

$\beta := 1$,

γ := a random element of GF(q).

If $r \geq \frac{1}{q+1}$ then do **Step 2b.**

Step 2b.

$\alpha := 1$,

β := random element of GF(q),

γ := random element of GF(q).

Step 3.

Choose a random integer n, $0 \leq n \leq q-1$.

Step 4.

Send (α,β,γ,n) to transmitter and receiver.

The key source sends $3\lceil \log_2 q \rceil + 1 \approx \log_2 b$ bits of key, or 1.5 bits of key for every bit of message.

The transmitter (sending source state i).

Step 1.

$j := i+n \bmod q$.

groupj := the jth element of GF(q).

Step 2.

If $\beta \neq 0$ then $x_1 :=$ groupj,

and $x_2 := \frac{-\alpha x_2 - \gamma}{\beta}$,

If $\beta = 0$ then $x_1 := -\frac{\gamma}{\alpha}$,

and $x_2 :=$ groupj.

message := (x_1,x_2).

Step 3.

Send message to receiver via an open channel.

The receiver.

If message is not of the form (x_1,x_2), where $x_1,x_2 \in$ GF(q), then reject message.

Step 1.

If $\alpha x_1 + \beta x_2 + \gamma \neq 0$ then reject message.

If $\alpha x_1 + \beta x_2 + \gamma = 0$ and $\beta = 0$ then groupj := x_2.

If $\alpha x_1 + \beta x_2 + \gamma = 0$ and $\beta \neq 0$ then groupj := x_1.

Step 2.

Retrieve j, where groupj is the jth element of GF(q).

Source state is (j-n) mod q.

Code 4.9

An optimal (1,1)-code with q+1 source states, q^2+q+1 messages and $(q^2+q+1)(q+1)$ encoding rules, where q is a prime power.

The encoding array is a projective plane whose blocks are expanded into Latin squares, by choosing an arbitrary order of points on each block and rotating cyclically.

First construction for PG(2,q).

A projective plane can be constructed as an extension of an affine plane. Consider GF(q) X GF(q) = $\{(\alpha,\beta) \mid \alpha,\beta \in GF(q)\}$. Take lines of slope α, for every $\alpha \in GF(q)$, that is, a set $\{(x,y) \mid y = \alpha x+\beta\}$, for each $\beta \in GF(q)$. Add the lines of slope ∞, namely $\{(x,y) \mid x = \beta\}$, for each β. This is the affine plane.

To each of these parallel classes of lines (lines of the same slope), we adjoin a point ∞_α (or ∞_∞). We add a line through these '∞' points, $\{\infty_\alpha, \alpha \in GF(q) \cup \{\infty\}\}$. Our messages are (α,β) pairs, (where $\alpha,\beta \in GF(q)$), together with the symbols ∞_α for $\alpha \in GF(q)$, and ∞_∞.

Example 4.9.1. For example, consider GF(3) = {0,1,2}.

Then GF(3)XGF(3) = {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)}.

We abbreviate (a,b) to ab for ease of reading.

The lines of the affine plane are in the table below.

slope 0	slope 1	slope 2	slope ∞
00,10,20	00,11,22	00,12,21	00,01,02
01,11,21	01,12,20	01,10,22	10,11,12
02,12,22	02,10,21	02,11,20	20,21,22

We affix a point ∞_0 to all lines of slope 0, ∞_1 to those of slope 1, ∞_2 to those of slope 2 and ∞_∞ to those of slope ∞. The '∞' points are then connected.

This gives us the following blocks:

00	10	20	∞_0
01	11	21	∞_0
02	12	22	∞_0
00	11	22	∞_1
01	21	20	∞_1
02	10	21	∞_1
00	12	21	∞_2
01	10	22	∞_2
02	11	20	∞_2
00	01	02	∞_∞
10	11	12	∞_∞
20	21	22	∞_∞
∞_0	∞_1	∞_2	∞_∞

The points in each block of PG(2,q) need some ordering so we can apply a series of cyclic shifts to them to get a Latin square. First order the points of GF(q). The lines with slope $\alpha \in$ GF(q) are ordered by the first element of the pair, with ∞_α last.

Example 4.9.1. (Part 2) For example, in PG(2,3) (over GF(3)),if GF(3) is ordered 0, 1, 2, the block with slope 2 through 11 is ordered 02, 11, 20, ∞_2. The blocks of slope ∞ are ordered by the second element, with ∞_∞ last. For example, the block of slope ∞ through 11 in PG(3) as above is ordered 10, 11, 12, ∞_∞ The '∞' line is ordered by subscript. In PG(3) as above, the ordering of this line is ∞_0, ∞_1, ∞_2, ∞_∞.

Each block becomes q+1 rows in the encoding array.

Example 4.9.1. (Part 3) For example, the 4 rows arising from the block 02, 11, 20, ∞_2 are

	source state			
	0	1	2	3
row 1	02	11	20	∞_2
row 2	11	20	∞_2	02
row 3	20	∞_2	02	11
row 4	∞_2	02	11	20

Algorithm 4.9.1

The key source.

Step 1. {Choose a random line of PG(2,q)}

Choose a real number r, $0<r<1$ from a uniform distribution.

If $r < \frac{1}{q^2+q+1}$, then line := inf, (the line through the '∞'s).

If $r \geq \frac{1}{q^2+q+1}$, then α := a random element of GF(q)$\cup\{\infty\}$,

{the slope},

If $\alpha = \infty$ then

β := a random element of GF(q),

$\gamma := 0$.

If $\alpha \neq \infty$ then

$\beta := 0$,

γ := a random element of GF(q).

{(β,γ) is a point on the line.}

line := (α,β,γ).

Step 2. {Pick a random row of the Latin square,

i.e. a cyclic shift.}

n := a random integer, $0 \leq n \leq q$.

Step 3.

Transmit (line,n) to transmitter and receiver via a secure channel.

The number of bits sent by the key source is about $1+3\log_2(q+1)$ (since we need 1 bit to represent the line inf, $\lceil \log_2(q+1) \rceil$ to represent α, $\lceil \log_2(q+1) \rceil$ to represent β or γ, depending on what α is, and $\lceil \log_2(q+1) \rceil$ to represent n), or 2 bits of key for every bit of message. The number of bits sent is not much more than the $\lceil \log_2 b \rceil \approx 3\log_2(q+1)$ required to specify the encoding rule.

The transmitter (sending source state i).

Step 1.

j := (i+n) mod (q+1)

Step 2.

(message is the jth element of the block chosen by the key source.)

If line = inf is received from the key source then

message := $\infty_{(\text{jth element of GF(q)})}$.

If slope is $\alpha \in$ GF(q) then

message := (jth element of GF(q), corresponding point on the line).

If slope is ∞ then message := (β, jth element of GF(q)).

Step 3.

Transmit message to receiver via an open channel.

The receiver.

If the message is not an element of
$GF(q)XGF(q)\cup\{\infty_x, x\in GF(q)\cup\{\infty\}\}$ then reject message.

Case 1.

line = inf is received from the key source.

Unless the message is in $\{\infty_x, x\in GF(q)\cup\{\infty\}\}$, reject message.

If message is ∞_x and x is the jth element of GF(q) then the source state is (j-n) mod (q+1).

If message is ∞_∞.then the source state is q+1-n

Case 2.

(∞, β, γ) is received from the key source.

If message is an element of $\{\infty_x, x\in GF(q)\}$ reject message.

If message is ∞_∞, then source state is q+1-n.

If message is (μ,η),for some $\mu,\eta\in GF(q)$, then if $\mu\neq\beta$, reject message.

Else note j, where η is the jth element of GF(q).

Source state is (j-n) mod (q+1)

Case 3.

$(\alpha, \beta, \gamma, n)$ is received from the key source.

If message is an element of $\{\infty_x, x\in GF(q)\cup\{\infty\}\}$, then if $x\neq\alpha$ reject message. If $x = \alpha$, then source state is (- n) mod (q+1).

If the message is (μ,η),for some $\mu,\eta\in GF(q)$, then if $\eta-\gamma \neq \alpha(\mu-\beta)$ then reject message. Else the point lies on the line and source state is (j-n) mod (q+1) where μ is the jth element of GF(q).

Second construction for PG(2,q).

PG(2,q) can be constructed using Singer's theorem. That is, consider $GF(q^3)$ with primitive element β and identify it with $[GF(q)]^3$. Then the points of PG(2,q) are the 1-dimensional subspaces of $[GF(q)]^3$. A line of PG(2,q) is the kernel of a linear functional from $[GF(q)]^3$ to GF(q). We choose one whose kernel is easy to recognize:

L:$(\alpha,\gamma,\delta) \rightarrow \delta$. The kernel of L is the block represented by $(1,\alpha,0)$, $\alpha \in GF(q)$ and the point (0,1,0). We interpret these representative points as powers of β and develop the powers modulo q^2+q+1. We note that any two representatives of a 1-dimensional subspace are scalar multiples of one another. That is, if (a,b,c) and (d,e,f) are representatives of the same 1-dimensional subspace, then there is a constant $\lambda \in GF(q)$ such that $a = \lambda d$, $b = \lambda e$, and $c = \lambda f$. That is, $(a,b,c) = (0,0,\lambda)\ (d,e,f)$, and $(0,0,\lambda) = \beta^{n(q^2+q+1)}$, for some integer n.

Example 4.9.4. For example, $GF(27) = GF(3^3)$ is generated by β, a root of x^3+2x+1. The kernel of the linear functional L:$(\alpha,\gamma,\delta) \rightarrow \delta$ is represented by

(1,0,0)	(1,1,0)	(1,2,0)	and	(0,1,0).
β^2	β^{10}	β^4		β

Now $q^2+q+1 = 13$, so the projective plane of order 3 is the row

2 10 4 1 developed modulo 13.

We number the source states from 0 to q.

Algorithm 4.9.2.

> **The key source.**
> **Step 1.**
> Choose n, $0 \le n \le q^2+q$.
> **Step 2.**
> Choose x, $0 \le x \le q$.
> **Step 3.**
> Transmit (n,x) to transmitter and receiver.

The number of bits sent by the key source is $\lceil \log_2(q^2+q+1) \rceil + \lceil \log_2(q+1) \rceil$ which is $\lceil \log_2 b \rceil$ or $\lceil \log_2 b \rceil + 1$, about 1.5 bits of key for every bit of message.

The transmitter (sending source state i).

Step 1

If $(i+x) \bmod q+1 = q$ then $P := (0,1,0)$

else $P :=$ (1, ith element of GF(q),0).

message $:= (\log_\beta(P)+n) \bmod (q^2+q+1)$.

Step 2.

Send message to receiver via an open channel.

The receiver.

If the message is not an element of $\{0, \ldots, q^2+q\}$ then reject message.

Step 1.

$P := \beta^{(\text{message}-n) \bmod (q^2+q+1)}$.

Step 2.

If P is not of the form $(p_1,p_2,0)$ then reject message.

If $p_1 = 0$ then $i := (q-x) \bmod q+1$.

else retrieve j from $\frac{p_2}{p_1}$, the jth element in GF(q).

$i := (j-x) \bmod q+1$.

Code 4.10

A (2,2)-code with q+1 source states, q^d+1 messages and $\frac{(q^d+1)q^d(q^d-1)}{q-1}$ encoding rules which is two times the optimal number, where q is a Mersenne prime and d is any integer.

Let $GF(q^d)$ have primitive element β.

Then α, the primitive element of GF(q) is $\beta^{\frac{q^d-1}{q-1}}$.

The encoding array is constructed from the inversive plane of order q, $S_1(3,q+1,q^d+1)$. Each block is expanded into q(q+1) rows of the encoding array by using an orthogonal array OA(q+1,q+1), created from the affine plane of order q+1, noting that q+1 is a prime power. To construct the OA(q+1,q+1), let the points in AG(2,q+1) be GF(q+1)XGF(q+1).The lines are

$y = mx + c$ for $m,c \in GF(q+1)$

$x = d$ for $d \in GF(q+1)$.

The lines y = c and x = d are used as a reference grid and hence ignored.

The lines $y = mx + c$, $m \in GF(q+1)\backslash\{0\}$, $c \in GF(q+1)$ are used to determine entries in the OA(q+1,q+1). The entry in the xth column of the row indexed by (m,c) is y = mx + c. The source states are numbered from 0 to q and the messages are in $GF(q^d)\cup\{\infty\}$. The initial block of the $S_1(3,q+1,q^d+1)$ is $GF(q)\cup\{\infty\}$. Each block of $S_1(3,q+1,q^d+1)$ is the image of this initial block under an element of $PGL(2,q^d)$, the general linear group on $q^d + 1$ elements, consisting of the permutations of $GF(q^d)\cup\{\infty\}$, defined by

$$x \rightarrow \frac{ex+f}{gx+h}, \qquad e,f,g,h \in GF(q^d),\ eh-fg \neq 0.$$

Each block is generated by (q-1)q(q+1) different permutations, since PGL(2,q) is the subgroup of $PGL(2,q^d)$ that fixes $GF(q)\cup\{\infty\}$. In order to choose a block equiprobably, a permutation in $PGL(2,q^d)$ is chosen equiprobably. Note that (e,f,g,h) generates the same permutation as $(\alpha e,\alpha f,\alpha g,\alpha h)$, $\alpha \in GF(q^d)\backslash\{0\}$. Note also that if e = 0, then $f \neq 0$. So the permutations are (1,f,g,h), for $h \neq fg$, and (0,1,g,h), for $g \neq 0$.

However, this leaves us with the same problem that we had in Code 4.7. However, our solution here is not as simple, since merely listing the permutations of $PGL(2,q^d)$ that act on $GF(q)\cup\{\infty\}$ leads us to a code with (q-1) times the optimal number of encoding rules (as compared to twice the optimal number in code 4.7.)

So we are faced with imposing some other order on the elements of a block.

We could calculate the entire block and then sort it - that means O(q) field operations in the encoding and decoding stages. Alternatively, we could choose one set (e,f,g,h) for

each block beforehand and store it on some cheap serially-accessed device (for example magnetic tape). Then, all the key source needs to do is to choose an integer x between 1 and $\frac{(q^d-1)q^d(q^d+1)}{(q-1)q(q+1)}$, and access the the xth record (e,f,g,h) on the tape. The amount of storage required for each record is $(1 + 3d\log_2 q)$ bits (since e is 0 or 1), which gives us the total amount of storage required as $\frac{(q^d-1)q^d(q^d+1)(1+3d\log_2 q)}{(q-)q(q+1)} = O(q^{d-3}d\log_2 q)$ bits. The slow step is left in the key source calculation, rather when our operatives are out in the field transmitting information. So we adopt this solution.

Algorithm 4.10.

The key source.
Step 1. {Choose a block of the inversive plane.}
Choose a random integer x, $0 \le x \le \frac{(q^d-1)q^d(q^d+1)}{(q-1)q(q+1)}$.
Access the xth precalculated entry (e,f,g,h) on the tape.
Step 2. {Choose a row of the orthogonal array.}
m := a random element of GF(q+1)\{0}.
c := a random element of GF(q+1).
Step 3.
Send (e,f,g,h,m,c) to transmitter and receiver via a secure channel.

The number of bits sent by the key source is $1 + 3\lceil d\log_2 q \rceil + 2\lceil \log_2(q+1) \rceil$, approximately $(3d+2)\log_2 q$, or about 3 bits of key for every bit of message. The number of bits sent is not much more than the $\lceil \log_2 b \rceil \approx (3d-1)\log_2 q$ required to specify the encoding rule.

The transmitter (sending source state i).

Step 1.

groupi := the ith element of GF(q+1).

Step 2.

y := (m(groupi) + c) in GF(q+1).

Step 3.

Note k, where y is the kth element of GF(q+1).

If $k \neq q$ then

groupk := the kth element of GF(q).

if groupk $\neq -\frac{h}{g}$ then message := $\frac{e(\text{groupk}) + f}{g(\text{groupk}) + h}$, (in $GF(q^d)$).

if groupk $= -\frac{h}{g}$ then message := ∞.

If $k = q$ then

if $g \neq 0$ then message := $\frac{e}{g}$, (in $GF(q^d)$).

if $g = 0$ then message := ∞.

Step 4.

Send message to receiver via an open channel.

The receiver.

Step 1.

If message = ∞ then groupk := $-\frac{h}{g}$, (in GF(q^d)).

If message = $\frac{e}{g}$ then groupk := ∞.

If message $\neq \infty$, and message $\neq \frac{e}{g}$ then groupk := $\frac{f - h(\text{message})}{g(\text{message}) - e}$.

Step 2.

If groupk $\notin$ GF(q)$\cup\{\infty\}$ then reject message.

If groupk = ∞ then k := q.

If groupk $\in$ GF(q) then note k, where groupk is the kth element of GF(q).

Step 3.

y := the kth element of GF(q+1).

Step 4.

groupi := $\frac{c - y}{m}$, (in GF(q+1))

Step 5.

source state is i, where groupi is the ith element of GF(q+1).

Code 4.11

An optimal perfect disclosure code for q+1 source states, q^2+q messages and q^t encoding rules, where $Pd_i = \frac{1}{q}$ for $0 \le i \le (t-1)$. q is a prime power and $t \le q$.

Let GF(q) have primitive element β. The messages are elements of $\{0, ..., q\}$XGF(q), and the source states are numbered 0 to q. The perfect disclosure is apparent because the first coordinate of the message is the source state number. The encoding array has rows

$$(0,\alpha_0) \quad (1,\alpha_{t-1}) \quad \left(2, \sum_{j=0}^{t-1}\alpha_j\right) \quad \left(3, \sum_{j=0}^{t-1}\alpha_j\beta^j\right) \quad \ldots \quad \left(q, \sum_{j=0}^{t-1}\alpha_j\beta^{(q-2)j}\right)$$

for every t-tuple $(\alpha_0,\alpha_1, \ldots ,\alpha_{t-1})$, $\alpha_k \in$ GF(q).

Example 4.11. When q = 3 and t = 2, there are 4 source states, 12 messages, and 9 encoding rules.

α_1	α_2	encoding array			
0	0	00	10	20	30
0	1	00	11	21	31
0	2	00	12	22	32
1	0	01	10	21	32
1	1	01	11	22	30
1	2	01	12	20	31
2	0	02	10	22	31
2	1	02	11	20	32
2	2	02	12	21	30

Algorithm 4.11.

The key source.

Step 1.

For every integer j, $0 \le j \le t-1$,

α_j := a random element of GF(q).

Step 2.

Send $(\alpha_0, \alpha_1, \ldots, \alpha_{t-1})$ to transmitter and receiver by secure channel.

The number of bits sent is $t\lceil \log_2 q \rceil \approx \log_2 b$, or about $\frac{t}{q}$ bits of key for every bit of message.

The transmitter (encoding source state i).

Step 1.

If i = 0 then message := $(0,\alpha_0)$.

If i = 1 then message := $(1,\alpha_{t-1})$.

If $i \notin \{0,1\}$ then message := $\left(i , \sum_{j=0}^{t-1} \alpha_j \beta^{(i-2)j} \right)$.

Step 2.

Send message to receiver via open channel.

The receiver.

Step 1.

Put (m_1,m_2) = message.

Step 2.

If $m_1 = 0$ and $m_2 \neq \alpha_0$, then reject message.

If $m_1 = 1$ and $m_2 \neq \alpha_{t-1}$, then reject message.

If $m_1 \notin \{0,1\}$ and $m_2 \neq \sum_{j=0}^{t-1} \alpha_j \beta^{(m_1-2)j}$ then reject message.

Step 3.

If the message is acceptable then the source state is m_1.

Code 4.12

A perfect disclosure code with $\frac{q^d-1}{q-1}$ source states, having $\frac{q(q^d-1)}{q-1}$ messages and q^d encoding rules, for which $Pd_0 = Pd_1 = \frac{1}{q}$.

We construct a transversal design $TD_{qd-2}\ (2,\frac{q^d-1}{q-1},q)$.

To do this we consider the projective geometry PG(d,q) where the points are equivalence classes of (d+1)-dimensional vectors over GF(q) excluding the zero vector. The

equivalence classes are defined by $(x_1,x_2, \ldots ,x_{d+1}) \sim (y_1,y_2, \ldots y_{d+1})$ if there exists a $\lambda \in GF(q)\backslash\{0\}$ such that $x_n = \lambda y_n$, for all $n \in \{1, \ldots ,d+1\}$.

We choose a point $\mathbf{k} = (1,0,0, \ldots ,0)$, representing the one-dimensional subspace that it generates. We choose a hyperplane **H** not containing **k** for ease of recognizing elements, say $\mathbf{H} = \{(x_1,x_2, \ldots ,x_{d+1}) \mid x_1 = 0\}$. The columns of our transversal design are indexed by the lines in PG(d,q) through **k**. Since a line intersects any hyperplane in exactly one point, we can index these lines by the elements of **H**. The rows of our transversal design are indexed by the hyperplanes of PG(d,q) which do not contain **k**, that is, by those

$$\alpha_1x_1 + \alpha_2x_2 + \alpha_3x_3 + \ldots + \alpha_{d+1}x_{d+1} = 0,$$

where $\alpha_1 \neq 0$, and $\alpha_i \in GF(q)$, for all $i \in \{1, \ldots ,d+1\}$.

Since any multiple of the coordinate vector produces the same line, we set $\alpha_1 = 1$, and the rest of the α_n are arbitrary.

The source state is represented by an integer i, where $1 \leq i \leq \frac{q^d-1}{q-1}$.

There is an intrinsic lexicographical ordering on **H** (determined by the ordering on GF(q)).

That is,

the first q^{d-1} points are $(0,1,x_3,x_4,x_5, \ldots ,x_{d+1})$,

the next q^{d-2} points are $(0,0,1,x_4,x_5, \ldots ,x_{d-1})$,

the next q^{d-3} points are $(0,0,0,1,x_5, \ldots ,x_{d-1})$,

up to the last point , $(0,0, \ldots ,0,1)$.

We shall consider this order imposed, and refer to $\mathbf{H} = \{\mathbf{h}_i , 1 \leq i \leq \frac{q^{d-1}}{q-1}\}$. Given i we can talk about $\mathbf{h}_i$, and given $\mathbf{h}_i \in H$, we can talk about i.

The entry in the ith column of the row of the transversal design indexed by the hyperplane **G** is the point on the line through **k** and $\mathbf{h}_i$ that lies on **G**. This point is the message sent to communicate source state i.

The hyperplane **G** is selected with defining equation

$$x_1 + \alpha_2x_2 + \alpha_3x_3 + \ldots + \alpha_{d+1}x_{d+1} = 0.$$

Let the point of intersection between this plane and the indexing line be $\lambda \mathbf{k} + \mu \mathbf{h}_i$, for some $\lambda, \mu \in GF(q)$. Then the point is $(\lambda, \mu(\mathbf{h}_i)_2, \mu(\mathbf{h}_i)_3, \ldots, \mu(\mathbf{h}_i)_{d+1})$, where $(\mathbf{h}_i)_j$ is the jth coordinate of $\mathbf{h}_i$. Since it lies on **G**,

$$\lambda + \mu\left(\sum_{j=2}^{d+1} \alpha_j (\mathbf{h}_i)_j\right) = 0.$$

So if $\sum_{j=2}^{d+1} \alpha_j (\mathbf{h}_i)_j = 0$ then set $\lambda = 0$, $\mu = 1$.

Else put $\lambda = 1$, $\mu = -\left(\sum_{j=2}^{d+1} aj(\mathbf{h}i)j\right)^{-1}$.

This ensures a leading 1.

Example 4.12. Consider d = 2 and q = 3.

Then we have 4 source states, 9 encoding rules and 12 messages.

We consider PG(2,3) over GF(3) and further abbreviate (a,b,c) to abc.

Then $\mathbf{k} = 100$, $\mathbf{H} = \{\mathbf{h}_1=010, \mathbf{h}_2=011, \mathbf{h}_3=012, \mathbf{h}_4=001\}$. The array becomes

α_1	α_2	α_3	$\mathbf{h}_1$	$\mathbf{h}_2$	$\mathbf{h}_3$	$\mathbf{h}_4$
1	0	0	010	011	012	001
1	0	1	010	122	112	102
1	0	2	010	111	121	101
1	1	0	120	122	121	001
1	1	1	120	111	012	102
1	1	2	120	011	112	001
1	2	0	110	111	112	001
1	2	1	110	011	121	102
1	2	2	110	122	012	101

Note that the messages in one column do not occur in any other column. Each message in a column occurs the same number of times in that column, and any pair of messages from distinct columns occurs precisely once in a row of the array.

Algorithm 4.12.

The key source.

Step 1. {Choose a random hyperplane.}

$\alpha_1 := 1$

For all $i \in \{2, \ldots, d+1\}$,

$\alpha_i :=$ a random element of GF(q).

Step 2.

Send $(\alpha_2, \alpha_3, \ldots, \alpha_{d+1})$ to transmitter and receiver via a secure channel.

There are $d\lceil \log_2 q \rceil \approx \log_2 b$ bits sent by the key source, or about 1 bit of key for every bit of message.

The transmitter (encoding source state i).

Step 1.

$\mathbf{k} = (1,0,0, \ldots, 0)$.

Step 2.

Find $\mathbf{h}_i$ and let $(\mathbf{h}_i)_j$ be the jth coordinate of $\mathbf{h}_i$.

Step 3.

If $\sum_{j=2}^{d+1} \alpha_j (\mathbf{h}_i)_j = 0$ then message = $\mathbf{h}_i$,

$$\text{else message} = \mathbf{k} - \left(\sum_{j=2}^{d+1} \alpha_j (\mathbf{h}_i)_j \right)^{-1} . \mathbf{h}_i .$$

Step 4.

Send message to receiver.

The receiver.

If message is not a (d+1)-tuple of elements in GF(q), whose first non-zero coordinate is a 1, then reject message.

Step 1.

Let m_j be the jth coordinate of the message.

If $\sum_{j=2}^{d+1} \alpha_j m_j \neq 0$ then reject message.

Step 2.

If $m_1 = 1$ then $m_1 := 0$ (equivalent to message = message - **k**),

elt := (1st non-zero coordinate of message)$^{-1}$·message,

else elt := message. (the first entry of message is a 1)

Step 3.

Retrieve i from $\mathbf{h}_i$ = elt, noting that elt is some $\mathbf{h}_i$, since its leading non-zero element is a 1.

Chapter 5 :
Arithmetic Operations in Finite Fields

5.1 Motivation

The efficiency of the algorithms presented in the last chapter is dependent upon the performance of the algorithms that are employed to perform finite field arithmetic. So, in order to discuss the efficiency of the algorithms in Chapter 4, (in terms of time and space requirements), we need to explore finite field operations in some detail. In most applications, these would be the finite fields $GF(2^n)$, for various values of n, usually less than about 10^3, although other finite fields, especially GF(p), where p is a prime, (in particular a Mersenne or Fermat prime), must also be considered. We explore methods of facilitating these arithmetic operations. We need to be able to add, multiply, divide (or take inverses), exponentiate and take (finite) logarithms with respect to some primitive element, and generate a random element (from an equiprobable distribution of all elements in the field). For a detailed discussion of the theory of finite fields, see Lidl and Niederreiter [41] or McEliece [47], although most of the relevant properties will be mentioned as needed.

5.2 The Traditional Approach

The traditional approach to finite field algorithms is worth examining, both for motivation, and because it is still very useful, especially when the size of the field that we're working over is relatively small.

A *polynomial basis* for a field $GF(p^n)$ over $GF(p)$ is a basis of the form $\{1, \alpha, \alpha^2, ..., \alpha^{n-1}\}$, where α is an element of the field $GF(p^n)$, and the elements $1, \alpha, \alpha^2, ..., \alpha^{n-1}$ are linearly independent. The *minimal polynomial* for α, is the monic polynomial $f(x)$ of least degree such that $f(\alpha) = 0$. This $f(x)$ is irreducible over $GF(p)$ (by necessity), and is called the *generating polynomial* of the field. A field $GF(p^n)$ with a polynomial basis can be constructed by starting with a generating polynomial $f(x)$ of degree n, and considering the elements of $GF(p^n)$ as residue classes of polynomials with coefficients in $GF(p)$ modulo $f(x)$.

This section examines the usual techniques for field operations when the elements of the field are represented as coordinate vectors with respect to a polynomial basis. We discuss $GF(2^n)$, which is constructed as the residue classes of the ring of polynomials over $GF(2)$ with respect to some irreducible nth degree polynomial $f(x)$.

That is, $GF(2^n)$ is the quotient ring $GF(2)[x]/(f(x))$, where $GF(2)[x]$ is the ring of polynomials over $GF(2)$, and $(f(x))$ is the ideal generated by $f(x)$.

In section 5.2, unless otherwise stated,

$$\alpha = \sum_{i=0}^{n-1} a_i x^i,\ \beta = \sum_{i=0}^{n-1} b_i x^i \text{ and } \gamma = \sum_{i=0}^{n-1} c_i x^i$$

are elements of $GF(2^n)$.

5.2.1 Addition in a Polynomial Basis

To add two elements of $GF(2^n)$ represented as coordinate vectors with respect to a polynomial basis, the coordinates of the elements are simply added modulo 2 ("exclusive or"ed). The following adds α and β to get γ.

Algorithm traditional addition.
for i := 0 to n-1 do
$c_i := a_i + b_i \mod 2$.

If α and β are elements of $GF(p^n)$ then we simply perform the addition modulo p.
The algorithm can be performed as a loop, or, if more than one processor is available, the additions modulo 2 can be performed in parallel. It is also easy to build special purpose hardware to perform the additions.
Figure 5.1 is the logic diagram for the sequential adder. It uses 1 exclusive or gate and adds in time O(n), and the parallel adder uses O(n) gates and adds in time O(1).

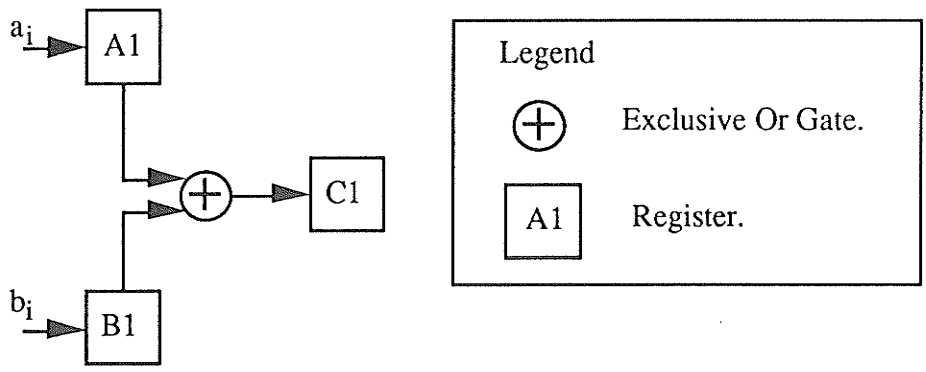

Figure 5.1. Serial adder.

Figure 5.2 is an example of a parallel adder over GF(16). It uses O(n) exclusive or gates and adds in time O(1)

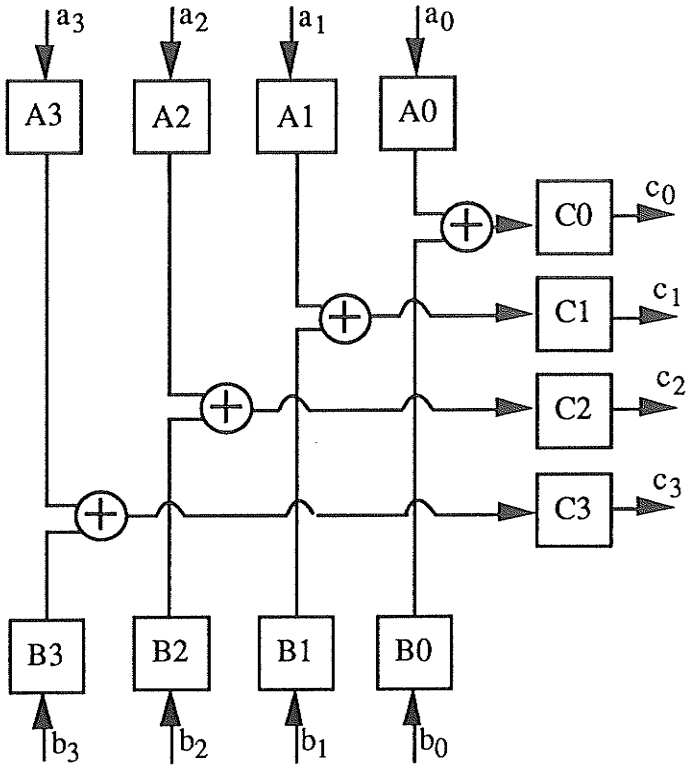

Figure 5.2. A Parallel Adder over GF(2^4).

To deal with addition in a polynomial basis in GF(p^n), where $p \neq 2$, we note that n additions in GF(p) need to be performed, and these may be performed in serial or parallel. To perform an addition modulo a prime p, we note that we are adding two $\log_2 p$ bit integers and subtracting zero or p (a fixed $\log_2 p$ bit integer.) This takes at most two $\log_2 p$ bitwise additions (noting that a bitwise addition and a subtraction are the same). So the addition takes $O(n\log_2 p)$ operations on one processor, or $O(\log_2 p)$ operation on n processors.

5.2.2 Multiplication in a Polynomial Basis

To multiply two elements, we note that a multiplication by x is a cyclic shift, with x^n replaced by $f(x) - x^n$. So we need an adaptation of a shift register with a feedback loop that replaces higher powers of x. We examine this by looking at an example, multiplying

two elements of GF(16) in which the generating polynomial is $f(x) = x^4 + x + 1$, and the polynomial basis is $\{1, \delta, \delta^2, \delta^3\}$, where δ is a root of f(x). To multiply $\alpha\beta = \gamma$, we load the bits b_i into registers Bi, so that B0, B1, B2 and B3 contain the bits of β, and the registers B4, B5 and B6 are zero Thus the register B is a shift register. The bit a_i is placed into the register A in clock cycle i. The answer c_i is in register Ci after n = 4 clock cycles. Each adder outputs the sum of its two or three inputs, and takes one or two exclusive or gates to implement (in general O(n) gates). So in this way, we can multiply in time O(n) using $O(n^2)$ gates. The left hand side of this circuit is dependant upon which f(x) is chosen as the generating polynomial of the field.

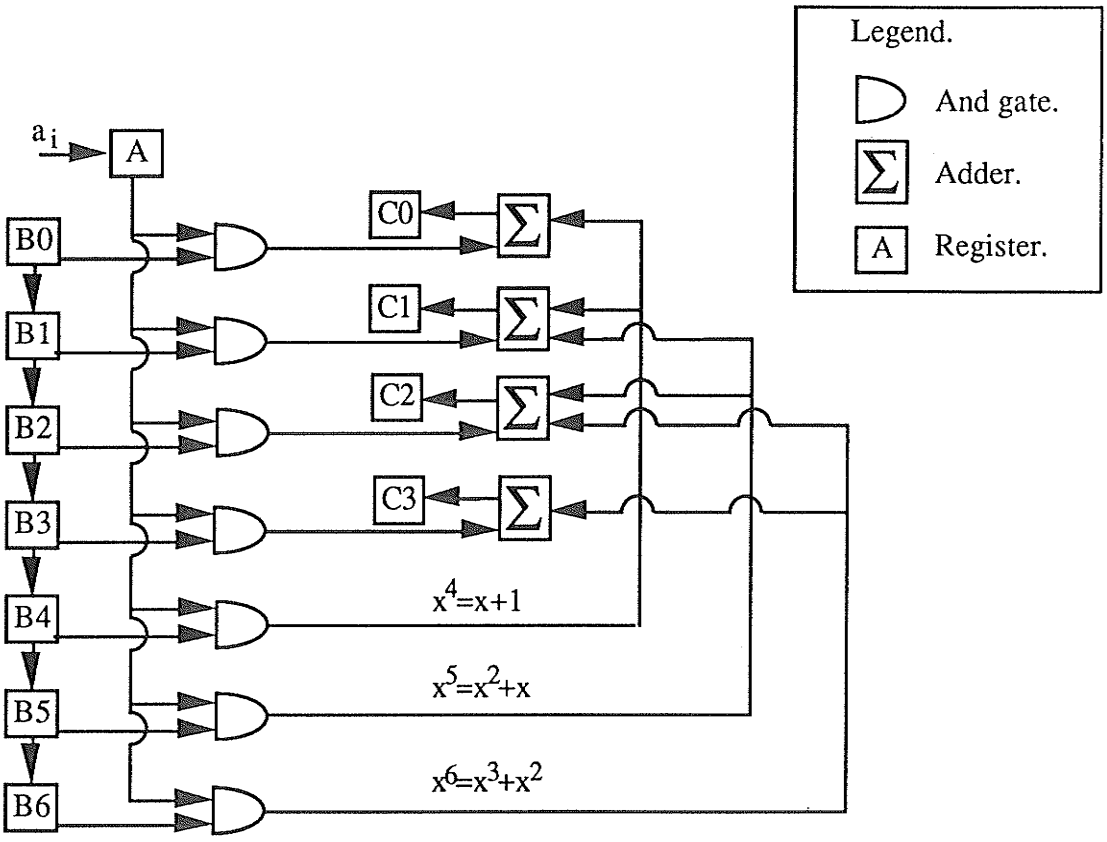

Figure 5.3. A polynomial basis multiplier over $GF(2^4)$.

Brickell [19] develops an algorithm to handle multiplication in GF(p) for prime p, (that is, modulo p) in $\log_2 p + 7$ clock pulses. Combining this and the above algorithm gives us an algorithm for multiplication in $GF(p^n)$ that takes time O(n logp).

Mastrovito [45] discusses this algorithm for multiplication in the field $GF(2^n)$. He claims that for 55% of the fields $GF(2^n)$, $2 \le n \le 1000$, we can chose the generating polynomial for the field to be a trinomial, so that in these cases, the adders in the above circuit all need but two Xor gates..

To implement this idea in VLSI see Rürer and Mehlhorn [27], Laws and Rushforth [39], Norris and Simmons [51], Wang et al [73], or Zak and Hwang [74], but their results are outside the scope of this document.

5.2.3 Exponentiation - Repeated Squaring

Firstly we note that to calculate the exponential α^e, it is always possible to multiply α by itself e times. If $0 \le e < 2^n-2$, this takes $O(2^n)$ multiplications, and there is an obvious vast improvement - using *repeated squaring*. If e is an integer such that $0 \le e < 2^n-2$, then we can express e in its binary form as

$$e = \sum_{i=0}^{n-1} a_i 2^i \,,\; a_i \in \{0,1\}.$$

Then for any element α of $GF(2^n)$, $\alpha^e = \prod_{i=0}^{n-1} \alpha^{a_i 2^i}$.

This computation requires m-1+n multiplications (m-1 multiplications of dissimilar terms, and n squarings), where $m = \sum_{i=0}^{n-1} a_i$ is the Hamming weight of (the number of ones in) the binary representation of e. Thus m = n-1 in the worst case (where, for example, $e = 2^n - 2$), and $m = \frac{n}{2}$ on average.

Repeated Squaring is also useful in $GF(p^n)$, where p is prime. This algorithm takes O(n) multiplications in GF(p).

In $GF(2^n)$, however, we know that squaring is a linear operation, so that we can express the square of some element

$$\gamma = \sum_{i=0}^{n-1} c_i x^i \qquad \text{where } \mathbf{c} \text{ is the vector of coefficients,}$$

as a matrix expression $S\mathbf{c}$. S is a binary n by n matrix whose entries depend on the specific generating polynomial chosen. Mastrovito [45] shows that if the generating polynomial is a trinomial, then this matrix can be realised in less than $\lceil \frac{3}{4}m \rceil$ gates.

5.2.4 Inverses - Euclid's Algorithm

If $\alpha \in GF(2^n)\backslash\{0\}$ is any non-zero element of the Galois field, then there is a unique element $\gamma \in GF(2^n)\backslash\{0\}$ such that $\alpha\gamma = 1$. This γ is called the inverse of α in $GF(2^n)$.
If the elements of $GF(2^n)$ are represented by the coordinate vectors of a polynomial basis, then we can use the extended Euclidean division algorithm for polynomials to find inverses in the field. That is, let u(x) be a generating polynomial for $GF(2^n)$ over GF(2). Let α be a root of u(x) that generates a polynomial basis for $GF(2^n)$ over GF(2). Given $v(x) \in GF(2^n)\backslash\{0\}$, we find inv(x) such that there is an $a(x) \in GF(2^n)$ which satisfies $v(x)\,inv(x) + u(x)\,a(x) = 1$. Note that u(x) is irreducible of degree n, and v(x) has degree less than n, so $gcd(u(x),v(x)) = 1$, and hence inv(x) must exist. Euclid's algorithm, or synthetic division of polynomials, is as follows.

Algorithm Euclid's Inverse.

Input: $v(x) \in GF(2^n)\backslash\{0\}$.

Output: $v^{-1}(x) \in GF(2^n)\backslash\{0\}$.

Step 0. {initialization}

$u'(x) := 0$

$u''(x) := u(x)$

$v'(x) := 1$

$v''(x) := v(x)$

Step 1.

While $v''(x) \neq 0$ do

$q(x) :=$ **div**$(u''(x), v''(x))$

$temp'(x) := v'(x)$

$temp''(x) := v''(x)$

$v'(x) := u'(x) - v'(x)\, q(x)$

$v''(x) := u''(x) - v''(x)\, q(x)$

$u'(x) := temp'(x)$

$u''(x) := temp''(x)$

Step 2.

$v^{-1}(x) := u'(x)$.

Algorithm **Euclid's Inverse** calls another algorithm, called **div** which performs polynomial division. Note that the degree of the polynomial 0 is $-\infty$ by convention. We let the sequence of degrees of the successive $v''(x)$ be

$$m = n_0, n_1, n_2, \ldots, n_t, -\infty, \qquad \text{where } n_t \geq 0.$$

Then in iteration i, for $0 \leq i \leq t$, a polynomial of degree n_{i-1} is divided by one of degree n_i, (where, for ease of expression, we call $n = n_{(-1)}$).

Algorithm div.

Input: $u(x) = \sum_{j=0}^{n_{i-1}} u_j x_j$, $u_{n_{i-1}}=1$;

$v(x) = \sum_{j=0}^{n_i} v_j x_j$, $v_{n_i}=1$.

Output: $q(x) = \sum_{j=0}^{n_{i-1}-n_i} q_j x_j$,

where $u(x) - q(x).v(x) = 0$ or $\deg (u(x) - q(x).v(x)) < n_i$.

Step 1.

For $k := n_{i-1}-n_i$ downto 0 do

$q_k := u_{n_i+k}$

For $j := n_i+k-1$ downto k do

$u_j := u_j - q_k v_{j-k}$.

The number of arithmetic operations required for algorithm **div** is essentially proportional to $n_{i-1}(n_i-n_{(i-1)}+1)$ (Knuth [37]).

Knuth [37] derives the average complexity for the algorithm **Euclid's Inverse** over all independently and uniformly distributed (monic) polynomials u(x) of degree n and v(x) of degree m. He calculates that we need a total of $2n^2 + 9n + 24 - \frac{3}{2^{n-2}} = O(n^2)$ (binary) operations, on average, to find an inverse using that algorithm.

This average is not taken over precisely the polynomials that arise in our problem, since u(x) is any polynomial, as opposed to a chosen irreducible one. It is actually an interesting question to ask if it is possible to choose an irreducible polynomial u(x), so as to minimize the average complexity of Euclid's division algorithm, but it is not one that shall be discussed here. The restriction of u(x) to irreducibles should not make any significant difference to the asymptotic behavior of the running time.

Euclid's algorithm does not lend itself well to LSI and VLSI implementation, since it is somewhat irregular. (It doesn't seem to have many repeatable steps at the hardware level.)

There have been a few reasonably sucessful attempts, for example Zak and Hwang [74], develop a systolic array for polynomial division that works in time O(n).

Over GF(p), Euclid's algorithm takes $O(n^2)$ field operations, on average to compute an inverse in $GF(p^n)$.

5.2.5 Logs - Tables

Given a generator α of $GF(2^n)$, and x a non-zero element of $GF(2^n)$, we define $\log_\alpha x$, the discrete logarithm of x with respect to α, to be the number y, $0 \le y \le 2^n-1$, such that $x = \alpha^y$ in $GF(2^n)$. This is regarded as a 'difficult' problem, and cryptosystems have been developed that rely on the fact that it is infeasible to calculate finite logarithms. The traditional method is that of a table look-up, storing each element as a power of α and as a vector of coordinates. If this table is stored (and it has the usually unacceptable size of $O(n2^n)$), it can also be used to facilitate multiplication.

Alternatively, we could use the powers of the primitive element α as the primary representation of the element of the field. This makes multiplication easy. To perform additions, we store a table of Zech's logarithms. The *Zech's log* of an integer i, $0 \le i \le 2^n-2$, is z(i), $0 \le z(i) \le 2^n-2$, where $1 + \alpha^i = \alpha^{z(i)}$. Then $\alpha^i + \alpha^j = \alpha^{i+z(i-j)}$. This table also has the size $O(n2^n)$, which is infeasible to store when n is large.

For example, consider $GF(2^3)$ generated by α where $f(x) = x^3+x+1$ and $f(\alpha) = 0$. Then the following table presents each element of $GF(2^3)$ as a vector of coordinates and a power of α and gives the Zech's logarithm of that power.

α^i	i	z(i)
000	∞	0
100	0	∞
010	1	3
001	2	6
110	3	1
011	4	5
111	5	4
101	6	2

5.3 Fast Fourier Transforms

Before we leave polynomial bases there are a couple of more sophisticated techniques worth examining. In this section and the next we examine the way these work. First, we examine the Fourier transform. Finite Fourier Transforms appear to be adaptable to perform multiplication in Galois fields (see Brassard And Bratley [17, Chapter 9], and Pollard [55]). A Fourier transform changes polynomials to n-tuples of points lying on the polynomial, which can then be manipulated more easily before being converted back.

Let us consider first this transformation, as outlined in Brassard and Bratley [17].

For convience, let m be a power of 2. Let $a = (a_0, a_1, ..., a_{m-1})$ be an m-tuple of elements in the field GF(q), where $m \mid (q-1)$. Define $p_a(x) = a_{m-1}x^{m-1} + ... + a_1x^1 + a_0$ be the polynomial whose coefficients are the vector a. Suppose ω is some constant element of GF(q) such that $\omega^{\frac{m}{2}} = -1$. This element must exist. The *discrete Fourier transform* of a with respect to ω is the m-tuple

$$F_\omega(a) = (p_a(1), p_a(\omega), ..., p_a(\omega^{n-1})).$$

At first glance, it appears the the transformation takes $O(m^2)$ operations to perform, but by using an algorithm called the *Fast Fourier Transform* (or just FFT), we can reduce this to O(m logm) operations.

The algorithm is not that difficult to understand. Basically, it is a divide-and-conquer method. We need a little more notation before we begin.

Put $t = \frac{m}{2}$. Then define the following two t-tuples:

$b = (a_0, a_2, ..., a_{m-2})$ (a_i where i is even)

and $c = (a_1, a_3, ..., a_{m-1})$ (a_i where i is odd)

Then we get the following equalities:

$$p_b(x^2) + xp_c(x^2) = p_a(x)$$

and $$p_b(\omega^2) + \omega p_c(\omega^2) = p_a(\omega).$$

Put $\alpha = \omega^2$; then

$$p_b(\alpha) + \omega p_c(\alpha) = p_a(\omega),$$

Now, $\alpha^{\frac{t}{2}} = \omega^t = \omega^{\frac{m}{2}} = -1$. So we can talk about $F_\alpha(b)$ and $F_\alpha(c)$, as Fourier transforms.

Also, $\alpha^t = 1$ and $\omega^t = -1$, so for any i, $0 \leq i \leq t$,

$$p_a(\omega^{t+i}) = p_b(\alpha^i) - \omega^i p_c(\alpha^i).$$

So we now have a method of breaking a Fourier transform down into two Fourier transfroms that are half the size, and recombining them to get our original.

This leads us to the following algorithm to compute the fast Fourier transform, which needs O(mlogm) space over GF(q).

Algorithm FFT (m,a,ω)

Input: m, a power of 2

$a = (a_0, a_1, ..., a_{m-1})$

$\omega \in GF(q)$ such that $\omega^{\frac{m}{2}} = -1$.

Output: $F_\omega(a) = (F_\omega(a)_0, F_\omega(a)_1, ..., F_\omega(a)_{m-1})$.

Step 1. If $m = 1$ then set $F_\omega(a)_0 = a_0$ and halt. If $m \neq 1$ then proceed.

Step 2. {initialization}

$t := \frac{m}{2}$.

$b = (b_0, b_1, ..., b_{t-1}) := (a_0, a_2, ..., a_{m-2})$

$c = (c_0, c_1, ..., c_{t-1}) := (a_1, a_3, ..., a_{m-1})$

Step 3. {recursion}

$F_\omega(b) = FFT\ (\frac{m}{2}, b, \omega^2)$

$F_\omega(c) = FFT\ (\frac{m}{2}, c, \omega^2)$

Step 4. {recombination}

$\alpha := 1$.

for $i := 0$ to $t-1$ do

{$\alpha = \omega^i$}

$F_\omega(a)_i := F_\omega(b)_i + \alpha F_\omega(c)_i$.

$F_\omega(a)_{t+i} := F_\omega(b)_i - \alpha F_\omega(c)_i$.

$\alpha := \alpha\omega$.

So, now we have the tool needed to convert efficiently polynomials of degree m-1 to m-tuples of points lying on the polynomial. Let's discuss conversions in the other direction.

Let γ be an element of GF(q) such that

(i) $\gamma \neq 1$

(ii) $\gamma^m = 1$

and (iii) $\sum_{j=0}^{m-1} \omega^{ji} = 0$, for all $1 \leq i \leq m$.

Then γ is called a *principal mth root of unity.*

Let m be a power of 2. Then the following results hold.

Theorem 5.3.1. ω is a principle mth root of unity if and only if $\omega^{\frac{m}{2}} = -1$. ■

Theorem 5.3.2. If ω is a principle mth root of unity then $\omega^{-1} = \omega^{m-1}$. ■

Theorem 5.3.3. If 1+1+...+1 (m times) $\neq 0$ then $\omega^0, \omega^1, ..., \omega^{m-1}$ are all distinct. These are all the principal mth roots of unity. ■

Define m = 1+1+...+1 (m times) in our field GF(q).

Theorem 5.3.4. The following statements are equivalent.

(i) m^{-1} exists in GF(q).

(ii) 1+1+...+1 (m times) $\neq 0$ in GF(q)

(iii) if $q = p^n$, p prime, then p does not divide m, that is, $p \neq 2$. ■

Let $a = (a_0, a_1, ..., a_{m-1})$ be an m-tuple of elements of the field GF(q), where q is not a power of 2, and let ω be a primitive mth root of unity. Then the *inverse Fourier transform* of a with respect to ω is the m-tuple

$$F_\omega^{-1}(a) = (m^{-1}p_a(1), m^{-1}p_a(\omega^{-1}), ..., m^{-1}p_a(\omega^{-(n-1)})).$$

Then the following theorem holds, and is possible to prove by substituting in the definitions of the Fourier transforms.

Theorem 5.3.5. $F_\omega^{-1}(F_\omega(a)) = F_\omega(F_\omega^{-1}(a)) = 0$. ■

This leads us to the following algorithm.

Algorithm FFTinv

Input: m, a power of 2
$a = (a_0, a_1, ..., a_{m-1})$
$\omega \in$ GF(q) such that ω is a primitive mth root of unity.

Output: $F^{-1}{}_\omega(a) = (F^{-1}{}_\omega(a)_0, F^{-1}{}_\omega(a)_1, ..., F^{-1}{}_\omega(a)_{n-1})$.

Step 1. $F^{-1}{}_\omega(a) = m^{-1}\mathbf{FFT}(m,a, \omega^{m-1})$.

If we are working over a field of characteristic 2, then the algorithms FFT and FFTinv can be modified so that m is a power of some other number, (presumably 3, as it is the next smallest). This means that the exponentiations are not merely shifts in base two representation, but this should not affect the asymptotic running time of the algorithm (see Pollard [55] for a detailed explanation of the changes that need to be made for the cases where m a power of a small prime, where m is "highly composite", and where m is "not highly composite").

This leaves us only with the question of manipulating the Fourier transforms of the polynomials. If $a(x)b(x) = c(x)$, and m and ω are as above, then $F_\omega(c)_i = F_\omega(a)_i \, F_\omega(b)_i$, from the definition of the Fourier transform.

So we have the following algorithm to multiply two elements of $GF(p^n)$.

Algorithm FFTmultiply

Input: $a = (a_0, a_1, ..., a_{n-1})$.
$b = (b_0, b_1, ..., b_{n-1})$.
$\omega \in GF(p^n)$ such that ω is a primitive mth root of unity.

Output: $c = (c_0, c_1, ..., c_{n-1})$.

Step 1. **Precomputation.**
Find m such that m is a power of 2 and $m \geq 2n$.
Pad a and b with zeros to get m-tuples.
Find $\omega \in GF(p^n)$ such that ω is a primitive mth root of unity.

Step 2. $A := FFT(m,a,\omega)$.
$B := FFT(m,b,\omega)$.

Step 2. $C_i := A_iB_i$, for all $0 \leq i \leq m-1$.

Step 3. $d := FFTinv(m,C,\omega)$.

Step 4. Reduce d modulo g(x) (the generating polynomial) to get c.

Steps 1 to 3 can be performed in O(n logn) field operations in GF(p).

However, computing step 4 with the algorithm div discussed in section 5.2.4 takes $O(n^2)$ field operations. So we need a more efficient algorithm for dividing a polynomial of degree 2n-2 by one of degree n. If the irreducible generating polynomial has only a very few terms (for instance if it is a trinomial)then the algorithm div in section 5.2.4 would only take O(nlogn) field operations in GF(p). So, if a suitable generating polynomial exists, the algorithm can be executed in O(n logn) field operation in GF(p).

Now, to add or subtract two polynomials, the Fourier transforms of those polynomials are merely added or subtracted. This is not a particularly efficient thing to do for itself, but if we have an equation involving additions, subtractions and multiplications, then we need only transform the domain once in each direction.

Reif [56, §2.2] discussed the parallel adaptation of the above algorithm. To multiply d polynomials together we compute the Fast Fourier Transform for each polynomial in parallel, and then compute, in parallel, the products obtained in step 2 of the algorithm FFT multiply. Then the inverse transform is performed. This reduces the time required for steps 1 to 3 to O(log(dn)) = O(logd + logn) using O(dn) gates. This is somewhat better than the naive method of multiplying two polynomials together, which takes time O(logd logn) with the same order of magnitude number of gates.

The other thing we'd like to be able to do in the Fourier transformed domain (and don't appear to be able to) is to divide polynomials (modulo the generating polynomial g(x)), or to find inverses by dividing our polynomial into the polynomial f(x) = 1. This does not seem to be something that can be done efficiently. This algorithm is purely for addition and multiplication.

5.4 A Parallel Algorithm for Exponentiation.

We can use the idea of a Fourier transform to develop an efficient parallel algorithm for exponentiating field elements represented as the sum of elements of a polynomial basis. Von zur Gathen [69] adapts Eberly's algorithm "Iterated Product of Polynomials" [25] to exponentiate in $GF(2^n)$ in time O(logn) and using space polynomial in n.

We wish to raise

$$a(x) = \sum_{i=0}^{n-1} u_i x^i$$

to the power e, where $0 \le e \le 2^n-1$.

Let $\quad e = \sum_{j=0}^{n-1} e_j 2^j, \qquad e_j \in \{0,1\}.$

Then $\quad [a(x)]^e = \prod_{j=0}^{n-1} a(x)^{e_j 2^j}$ (the same as our algorithm for repeated squaring)

$$= \prod_{j=0}^{n-1} \left(\sum_{i=0}^{n-1} u_i x^i \right)^{e_j 2^j}$$

Now, noting that squaring is linear in $GF(2^n)$, and $(u_i)^2 = u_i$, we get

$$[a(x)]^e = \prod_{j=0}^{n-1} \left(\sum_{i=0}^{n-1} u_i x^{i2^j} \right)^{e_j}$$

so that $[a(x)]^e$ is the product of the polynomials

$$\sum_{i=0}^{n-1} u_i x^{i2^j} \qquad \text{for all } 0 \le j \le n-1,\ e_j \neq 0.$$

So our algorithm has three steps, as follows:

Algorithm Parallel Exponentiate

Input: $a(x) = \sum_{i=0}^{n-1} u_i x^i$

$e = \sum_{j=0}^{n-1} e_j 2^j, \quad e_i \in \{0,1\}$

f(x), the generating polynomial of the field.

Output: $[a(x)]^e \bmod f(x)$

Step 1. Calculate $pj(x) = \sum_{i=0}^{n-1} u_i x^{i2^j}$ for every j in parallel, $0 \le j \le n-1$, $e_j = 1$.

Step 2. Find $[a(x)]^e$ by multiplying together pj(x) calculated in **Step 1.**

Step 3. Reduce $[a(x)]^e \bmod f(x)$.

Now, **Step 1** can be performed efficiently provided that x^{i2^j} has already been calculated for each $0 \le i \le n-1$ and each $0 \le j \le n-1$. Then, for each j we need to add (up to) n polynomials each of degree (up to) n-1. If we perform this addition for each coefficient in parallel, it takes time O(logn) if we have $O(n^2)$ gates.

Step 3 can be performed efficiently provided that x^k has already been calculated for $n \le k \le n^2-n$. We note that $[a(x)]^e$ has degree at most n^2-n, and that the powers of x less than n are elements of our basis. This step, then involves adding (up to) n^2-n polynomials of degree at most n-1. We do this for each coefficient in parallel and take time $O(\log(n^2)) = O(\log n)$ with $O(n^3)$ gates.

Step 2 is calculated using a variant of the Fast Fourier Transform. We need to be able to choose n^2+1 distinct elements of the underlying field. Unfortunately, our underlying field is GF(2), so we need to work in an extension field of GF(2), say $GF(2^r)$, where $2^r \ge n^2+1$. So we choose r to be about $2\log_2 n$. **Step 2**, then, is implemented as follows. Assume that n^2+1 constants $\gamma_0, \gamma_1, \ldots, \gamma_{n-1}$ are chosen from $GF(2^r)$.

Algorithm Step 2.

Step 2a. For all $0 \le k \le n^2$ and all $p_j(x)$ in parallel, calculate $p_j(\gamma_k)$

Step 2b. For all $0 \le k \le n^2$ in parallel, calculate $g(\gamma_k) = \prod_j p_j(\gamma_k)$

Step 2c. Interpolate the n^2+1 points to get the coefficients of $g(x)$.

To calculate $p_j(\gamma_k)$ in **Step 2a**, we need to add (up to) n field elements. To do this in time O(logn), we merely add the coefficients in parallel, as in **Step 1**. We do this for (up to) n^3 things in parellel, so we need $O(n^4)$ gates.

In **Step 2b**, we need to multiply (up to) n field elements in $GF(2^r)$. This can be done in time O(logn) only if multiplication in $GF(2^r)$ can be done in constant time. This is achieved by storing a multiplication table of elements of $GF(2^r)$. Note that this table will have about n^2 entries, and hence need storage $O(n^2 \log n)$.

In **Step 3b**, we need to do an interpolation. This is achieved by precomputing the entries of the inverse of the Vandermonde matrix $V(\gamma_0, \gamma_1, ..., \gamma_{n^2})$ of order n^2+1 defined by $V(\gamma_0, \gamma_1, ..., \gamma_{n^2})_{ij} = (\gamma_{i-1})^{j-1}$, for $1 \le i,j \le n^2+1$. Then the coefficients $g_0, g_1, ..., g_{n^2}$ of $g(x)$ are computed in parallel by the matrix multiplication

$$\begin{pmatrix} g_0 \\ g_1 \\ \cdot \\ \cdot \\ \cdot \\ g_{n^2} \end{pmatrix} = V(\gamma_0, \gamma_1, ..., \gamma_{n^2})^{-1} \begin{pmatrix} g(\gamma_0) \\ g(\gamma_1) \\ \cdot \\ \cdot \\ g(\gamma_{n^2}) \end{pmatrix}.$$

So the algorithm takes time O(logn) if we precompute

(i) a multiplication table in $GF(2^r)$

(ii) $x^{i2^j} \bmod f(x)$ for each $0 \le i \le n-1$ and each $0 \le j \le n-1$,

(iii) $x^k \bmod f(x)$ for each $n \le k \le n^2-n$,

and (iv) the entries of the inverse of the Vandermonde matrix $V(\gamma_0, \gamma_1, ..., \gamma_{n^2})$ of order n^2+1 defined by $V(\gamma_0, \gamma_1, ..., \gamma_{n^2})_{ij} = (\gamma_{i-1})^{j-1}$, for $1 \le i,j \le n^2+1$.

The amount of storage required in $O(n^4 \log n)$ and we need $O(n^4)$ gates.

5.5 Dual Bases

Before dual bases can be examined, we need to define the concept of a *trace*. If $\gamma \in GF(2^n)$, then the trace of γ is defined as $tr(\gamma) = \gamma + \gamma^2 + ... + \gamma^{2^{n-1}}$. Trace is a linear operator on $GF(2^n)$, considered as a vector space over GF(2). That is, $tr(\gamma) \in GF(2)$, for all $\gamma \in GF(2^n)$, and $tr(\gamma+\delta) = tr(\gamma) + tr(\delta)$, for all $\gamma,\delta \in GF(2^n)$. Also, $tr(\gamma) = tr(\gamma^{2^i})$, for $1 \le i \le n-1$.

Let $\{\alpha_i\}$ and $\{\beta_i\}$ be bases for $GF(2^n)/GF(2)$. Then $\{\alpha_i\}$ and $\{\beta_i\}$ are *dual bases* if $tr(\alpha_i\beta_j) = \delta_{ij}$, where δ_{ij} is the Kroneker delta (that is, $\delta_{ij} = 1$ if $i = j$, and $\delta_{ij} = 0$, otherwise). Dual bases are a useful way to represent elements of $GF(2^n)$, if multiplication is to be performed efficiently (by a process due to Berlekamp [8]), and are discussed in McEliece [46]. If $\{\alpha_i\}$ is a basis, a dual basis exists and is unique, (see Lidl and Niederreiter [41].)

McEliece presents Berlekamp's bit serial multiplication algorithm as follows.

Suppose that $\gamma \in GF(2^n)$. Then $\gamma = (trace(\gamma\beta_0), trace(\gamma\beta_1), ... ,trace(\gamma\beta_{n-1}))$ with respect to the basis $\{\alpha_i\}$, and conversely, $\gamma = (trace(\gamma\alpha_0), trace(\gamma\alpha_1), ... ,trace(\gamma\alpha_{n-1}))$ with respect to the basis $\{\beta_i\}$.

We call $\{\alpha_i\}$ the *primal* basis, and $\{\beta_i\}$ the *dual* basis. Let $\alpha_i = \alpha^i$ for each i, and for some primitive element α. That is, let $\{\alpha_i\}$ be a polynomial basis. Then multiplying any element x by α is easy in the dual coordinate system, as follows.

Let $x = x_{n-1}\beta_{n-1} + ... + x_1\beta_1 + x_0\beta_0$.

Now $\alpha x = \sum_{i=0}^{n-1} Tr(x\alpha^{i+1})\beta_i$.

Then $x_i = Tr(x\alpha^i)$.

So $\alpha x_i = Tr(\alpha x\alpha^i) = Tr(x\alpha^{i+1}) = x_{i+1}$, for $0 \le i \le n-2$.

Also, if $f(z) = \sum_{i=0}^{n-1} f_i z^i$ is the minimal polynomial for α, then

$$\begin{aligned} \alpha x_{n-1} &= Tr(x\alpha^n) \\ &= Tr\left(x\sum_{i=0}^{n-1} f_i\alpha^i\right) \\ &= \sum_{i=0}^{n-1} f_i Tr(x\alpha^i) \\ &= \sum_{i=0}^{n-1} f_i x_i. \end{aligned}$$

So multiplication by α is simply a cyclic shift, with $x_{n-1} = Tr(x\alpha^n)$. Figure 5.4 represents a circuit that multiplies x by the constant α, where x is represented in the primal basis, and in which representation the ith bit is loaded into the register Xi. The gates in the top of the figure calculate $Tr(x\alpha^n)$, and the ith bit of $x\alpha$ is contained in the register Xi after one clock cycle. If we tap the register Xi, the bits of $x\alpha^i$ appear serially. That is, after the kth clock cycle, Xi contains the kth component of $x\alpha^i$ in dual coordinates.

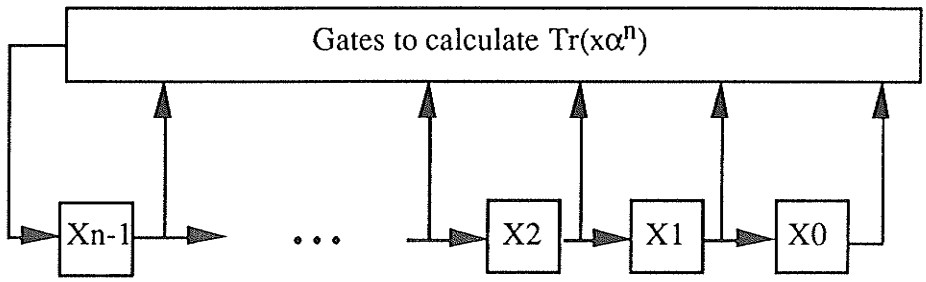

Figure 5.4. A "multiply by α" circuit in $GF(2^n)$.

Example 5.5.1. Suppose GF(8) has $f(z) = z^3+z+1$ as a generating polynomial, and a basis $\{1, \alpha, \alpha^2\}$, where α is a primitive element of the field and $f(\alpha) = 0$.

The dual basis of $\{1, \alpha, \alpha^2\}$ is $\{1, \alpha^2, \alpha\}$.

Now $\alpha x_2 = Tr(x\alpha^3) = Tr(x\alpha + x) = Tr(x\alpha) + Tr(x) = x_1 + x_0$.

That is, if $x = x_0 + x_1\alpha^2 + x_2\alpha$ then $\alpha x = x_1 + x_2\alpha^2 + (x_1 + x_0)\alpha$, so the "gates to calculate $Tr(x\alpha^n)$ " in figure 5.4 become the gate at the top of figure 5.5.

Figure 5.5 represents a circuit to perform bit-serial multiplication in GF(8) as outlined in Example 5.5.1. Now x (in dual coordinates) is loaded into registers 0, 1, and 2, and after one clock pulse, αx is contained in those registers.

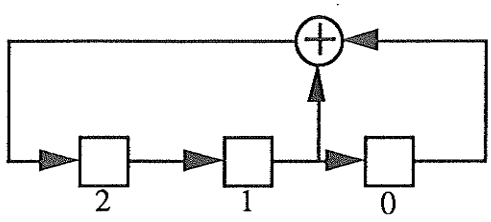

Figure 5.5. A "multiply by α" circuit in GF(8)

We can extend our multipliers to handle any specific constant. We consider the contents of register i at time t. We note that after t shifts, the shift register contains $x\alpha^t$. So, after 0 shifts, register i will contain $Tr(x\alpha^i)$. After 1 shift, register i will contain $Tr(x\alpha^{i+1})$, and after t shifts, register i contains $Tr(x\alpha^{i+t})$. Now any element of $GF(2^n)$ can be written as a sum of elements of the primal basis. If we load the registers with i, initially, sum the registers corresponding to terms in that sum, then we will get the coefficients of the product (in the dual basis) serially.

For example, in GF(8) as above, suppose that we want to multiply by α^4. We note that $\alpha^4 = \alpha^2 + \alpha$. So we adapt the circuit in figure 5.5 to become figure 5.6. We load x into registers 0, 1 and 2 at clock = 0, and after i shift cycles, the output is the ith component of $\alpha^4 x$, in the dual coordinate system. That is, if $x = x0 + x_1\alpha^2 + x_2\alpha$, then after 0 shifts, x_1+x_2 is output, after 1 shift , $x_0+x_1+x_2$, and after 2 shifts x_0+x_1 is output.

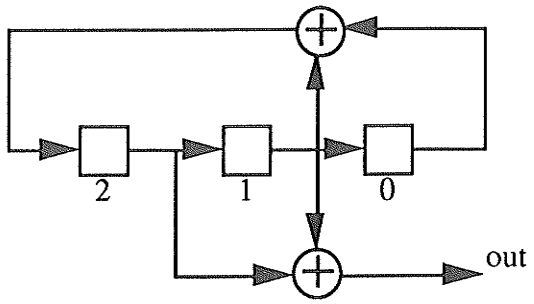

Figure 5.6. A "multiply by α^4" circuit in GF(8)

This provides us with the motivation behind the multiplier. Instead of having a constant "hardwired" into the circuit, the multiplicand is loaded into a register, and the elements of the shift register corresponding to the terms used to express the multiplicand as a sum of elements of the primal basis are added to produce the serial output. That is, the ith coordinate of the product is $\sum \mathrm{Tr}(\alpha^j x)$, where y is $\sum \alpha^j$. So, to multiply the element y expressed in the primal basis by the element x expressed in the dual basis, a circuit is constructed to the specifications of Figure 5.7. The coordinates of y are placed in the (static) Register A, the coordinates of x in the Shift Register B, in their respective bases. Again, the gates at the top of the diagram calculate $\mathrm{Tr}(x\alpha^m)$, and the adder calculates the (mod 2) sum of its n inputs. The ith bit of the product xy, in the dual coordinate system, is the result of the adder after the ith clock cycle.

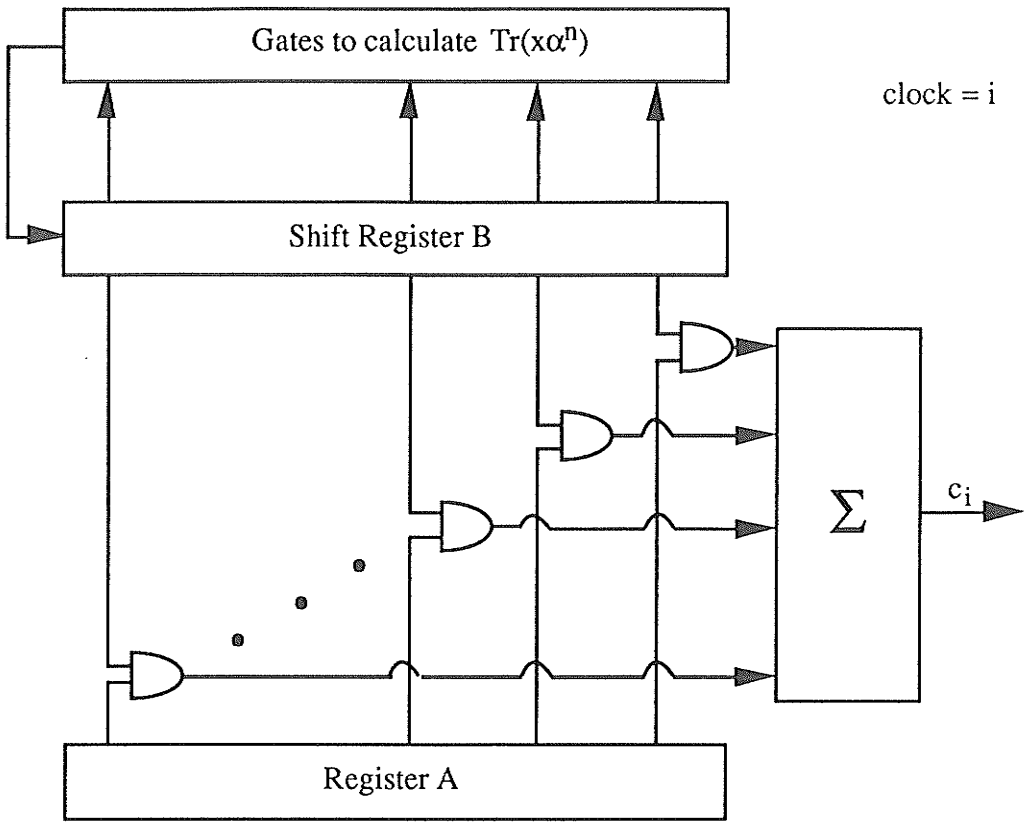

Figure 5.7. A dual basis multiplier over GF(2^n).

Lets look back at the field in Example 5.5.1. We wish to multiply $x = x_0 + x_1\alpha^2 + x_2\alpha$ (in dual coordinates) by $y = y_0 + y_1\alpha + y_2\alpha^2$ (in primal coordinates). We multiply x by the values y_0, $y_1\alpha$, and $y_2\alpha^2$ and add these to get the answer, the bits of which are output serially in dual coordinates. See Figure 5.8.

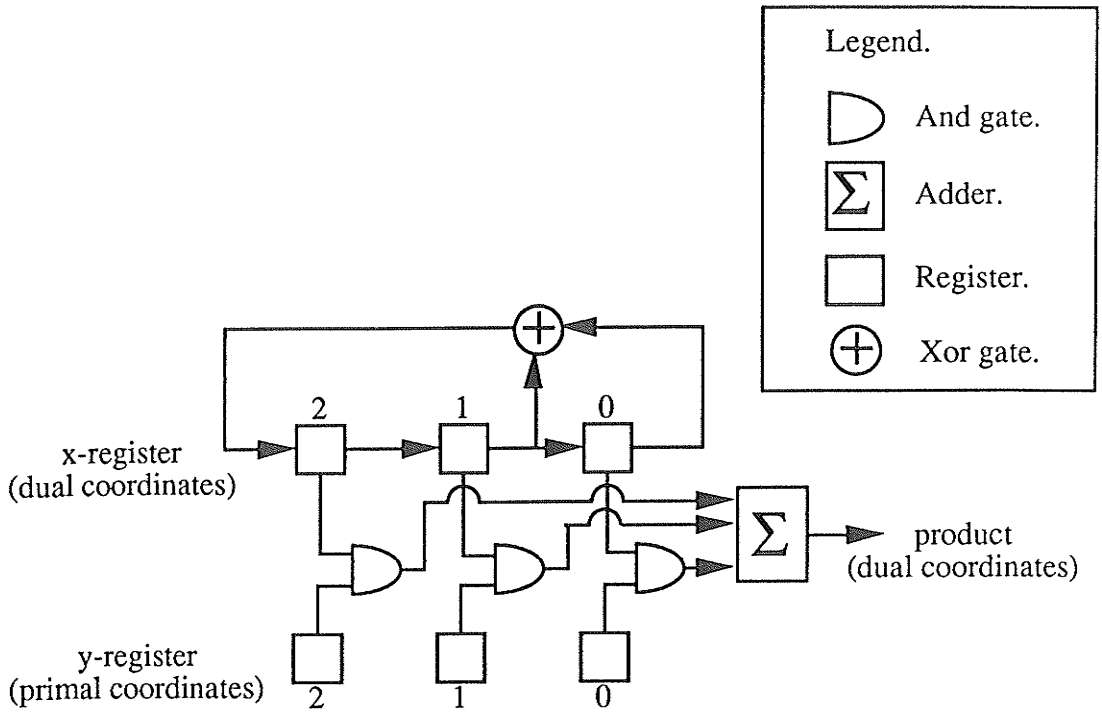

Figure 5.8. A dual basis multiplier over GF(8).

This multiplier contains O(n) gates and performs a multiplication in time O(nlogn) - there are n bits output, and the adder can have depth logn. However, the earlier bits of the product can be used before the multiplication is completely finished. That is, if the next operation that the product is used for is something in which the bits are input serially, that process can begin after only 1 shift cycle. However, if the next operation is the same type of multiplication, (say we were exponentiating by repeated squaring) then we need all the bits before we can begin and we have essentially no time saving.

The only problem that has not yet been addressed is the issue of converting from one basis to another. This can be partially solved by precalculating the dual basis representation of the constant 1, and loading this into Register A, in Figure 5.7. Then to calculate the dual basis representation of any element B represented in the primal basis, merely multiply B by the dual basis 1. This takes time O(nlogn), and can be performed on the existing hardware. To convert the dual basis element B back into the primal basis,

however, is not as simple, since the dual basis is not necessarily a polynomial one. If $\gamma \in GF(2^n)$, then $\gamma = (\text{trace}(\gamma\beta_0), \text{trace}(\gamma\beta_1), \ldots, \text{trace}(\gamma\beta_{n-1}))$ with respect to the basis $\{\alpha_i\}$. So in order to express B in terms of $\{\alpha_i\}$, we need circuitry to calculate n traces. This takes $O(n^2)$ gates, in general, and time O(logn). A better approach may be to do a straight matrix conversion from one basis to the other. If this matrix can be contrived to be sparse, the necessary number of gates may be reduced.

So what we do is look for polynomial bases whose duals are merely permutations, or whose duals are simple linear combination of the primal basis elements. In Example 5.5.1, the dual basis contained exactly the same elements as the primal basis, and hence it is an easy job to rewire the circuit so the only one basis is represented. See figure 5.9.

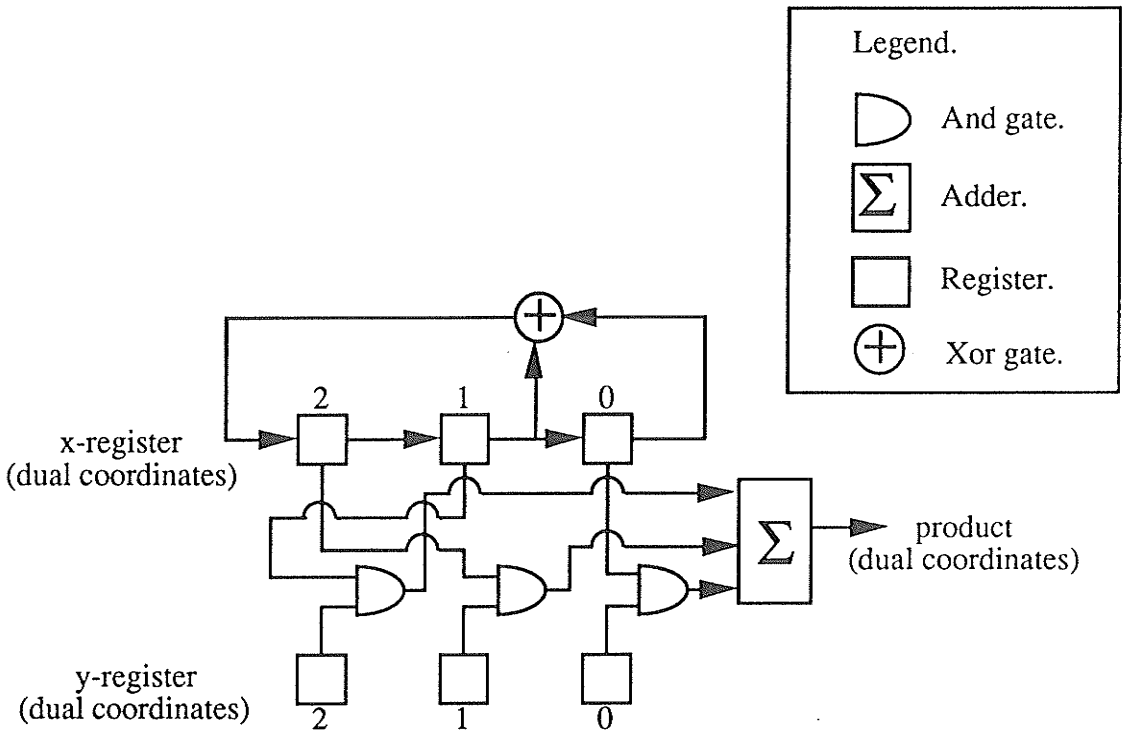

Figure 5.9. A dual basis multiplier over GF(8) with conversion.

Morii and Kasahara [48] discuss the question of converting from a dual of a polynomial basis back to that basis. They modify the bit-serial multiplier above, and call it the

generalized bit-serial multiplier (GBSM). The GBSM uses a pair of bases, a polynomial basis and a basis that can be chosen arbitrarily in a class of 2^n-1 elements. By choosing such a basis wisely, there may be a simple transform between the bases. Indeed, when the primitive polynomial is a trinomial (that is, has only three terms) the transform turns out to be a simple cyclic permutation.

They proceed as follows. For a fixed nonzero element $\beta \in GF(2^n)$, and arbitrary element $x \in GF(2^n)$, define the quantities $\mathbf{x}_i = Tr(\beta\alpha^i x)$ for i = 0, 1, ..., 2n-2 and denote $\mathbf{x} = (\mathbf{x}_i)_i$. Now, $\mathbf{x} = Tx$, where $T = (\varepsilon_{ij})$, $\varepsilon_{ij} = Tr(\beta\alpha^{i+j})$, for i,j = 0, 1, ..., m-1.

The basis $A = \{\alpha_0, \alpha_1, \alpha_2, ..., \alpha_{n-1}\}$ is said to be a *permutation dual* to the basis $B = \{\beta_0, \beta_1, \beta_2, ..., \beta_{n-1}\}$ if the matrix $\left(Tr(\alpha_i\beta_j)\right)$ is a permutation matrix.

Then the quantities $\mathbf{x}_i$ play the role of the dual coordinates in the previous scheme. This β can be chosen to be optimal easily in the case that the generating polynomial for the primal basis A is a trinomial. Indeed, β is chosen so that T is a *back circulant permutation matrix* (that is, one "1" per column in a matrix in which every entry in each back diagonal is identical) and that $T^{-1} = T$. Then, the coefficients of the primal basis may be obtained from the dual basis by a cyclic shift.

Lemma 5.5.1. (Wang and Blake [72, Lemma 1]) There exists an element $\beta \in GF(2^n)$ such that $B = \{\beta, \beta\alpha, \beta\alpha^2, ..., \beta\alpha^{n-1}\}$ is permutation dual to $A = \{1, \alpha, \alpha^2, ..., \alpha^{n-1}\}$ if and only if the minimal polynomial of α is a trinomial.

Proof. We outline the sufficiency proof only. Firstly note that if $\beta \in GF(2^n)$ is non-zero, and if A is a basis, then B is also one. Suppose that the minimal polynomial of α is the trinomial $x^n + x^k + 1$, for some $1 \le k \le n-1$. Since A is a basis, we can choose a unique β such that $Tr(\beta\alpha^{k-1}) = 1$, and $Tr(\beta\alpha^i) = 0$, for all $i \ne k-1$. Then T is a back circulant permutation matrix, as required. ■

It should be noted here that not every finite field $GF(2^n)$ can be generated by an element whose minimal polynomial which is a trinomial. Wang and Blake [72] go on to develop another form of the matrix T that is not as elegant as the trinomial case but always works

and results in a fairly efficient multiplication scheme. T is contrived to be a lower triangular matrix whose entries along the back diagonals are equal. They develop an explicit form for the inverse of T and use this to perform the basis conversions.

Also, many of these results may be extended to the general case of $GF(q^n)$ over GF(q).

5.6 Normal Bases

A *normal basis* is one of the form $\{\beta, \beta^2, \beta^4, \beta^8, \ldots, \beta^{2^{n-1}}\}$, (as opposed to the more commonly discussed polynomial basis) for some β in $GF(2^n)$. β is referred to as the *generator* of the normal basis. A normal basis in the field $GF(2^n)$ can be shown to exist for all n. In fact, a primitive normal basis (one in which all elements are primitive) exists for every finite field, (see Lenstra and Schoof [40]).

5.6.1 Multiplying in a Normal Basis - the Basic Approach

Suppose $a = \sum_{i=0}^{n-1} a_i\beta^{2^i}$, $b = \sum_{i=0}^{n-1} b_i\beta^{2^i}$ and $c = \sum_{i=0}^{n-1} c_i\beta^{2^i}$ are elements of $GF(2^n)$.

So $(a_0,a_1, \ldots,a_{n-1})$ and $(b_0,b_1, \ldots,b_{n-1})$ are the coordinate vectors for a and b in $GF(2^n)$ with respect to a normal basis, $N = \{\beta,\beta^2,\beta^4,\beta^8, \ldots ,\beta^{2^{n-1}}\}$. Let c = ab.

Define λ_{ij} by

$$c_0 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\lambda_{ij}a_ib_j .$$

Now, λ_{ij} is the coefficient of β in the normal basis representation of $\beta^{2^i+2^j}$, so the coefficient of β^{2^k} in $\beta^{2^{i+k}+2^{j+k}} = \left(\beta^{2^i+2^j}\right)^{2^k}$ is also λ_{ij}. (Since raising something to the 2^k is merely shifting it cyclically k places). Hence we calculate that

$$c_k = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\lambda_{ij}a_{i+k}b_{j+k},$$

for each $0 \le k \le n$, where the subscripts are taken modulo n.

Let $\lambda = (\lambda_{ij})$, the matrix of multiplication coefficients.

So $c_k = (a_k, a_{k+1}, \ldots, a_{k-1})\,\lambda \begin{pmatrix} b_k \\ b_{k+1} \\ \vdots \\ b_{k-1} \end{pmatrix}$, for each k.

To calculate any coordinate of c then, we cyclically shift the vectors representing a and b and use the matrix λ to calculate the bilinear form for c_k.

So we can implement a multiplier with two cyclic shift registers for a and b, and a gate-array realizing the matrix λ and an adder for the summation process. See Figure 5.10.

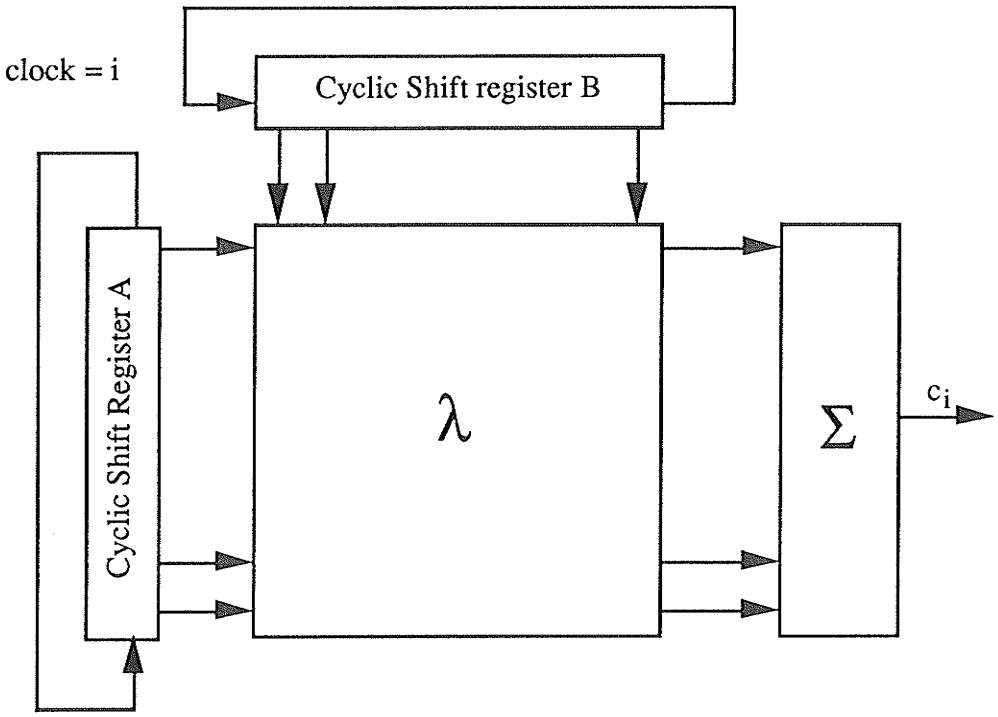

Figure 5.10 A normal basis multiplier.

Example 5.6.1. Consider the field GF(8). Let α be an element of GF(8) such that $\alpha^3 = \alpha^2+1$. Then $\{\alpha, \alpha^2, \alpha^4\}$ is a normal basis for GF(8). The matrix λ, as defined above is as follows

$$\lambda = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

Note that the (i,j)th element of λ is the jth coordinate of α^{2^i+1}. This multiplier is demonstrated in figure 5.11.

Example 5.6.2. Consider GF(2^5) generated by $f(x) = x^5+x^2+1$. Consider a primitive element α such that $f(\alpha) = 0$. Put $\beta = \alpha^3$. Then $\{\beta, \beta^2, \beta^4, \beta^8, \beta^{16}\}$ is a normal basis for GF(2^5). The corresponding λ is

$$\lambda = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix},$$

and could be hardwired into a similar circuit.

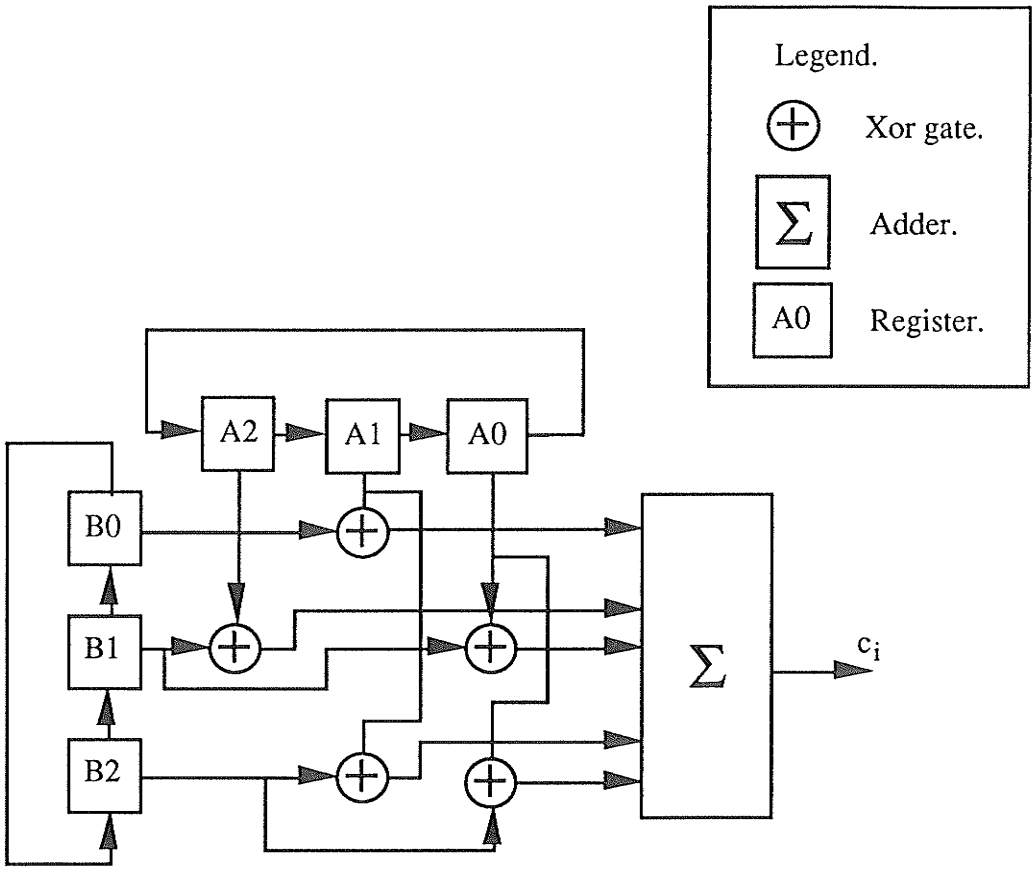

Figure 5.11.

This is sometimes called a Massey-Omura multiplier (see Massey [44]).

λ contains $O(n^2)$ gates on average. In §5.6.2, we discuss a way to reduce this to O(n) gates in some cases using Optimal Normal Bases. There must be $\log_2 n$ levels of Xor gates in the summation of the n terms, so the multiplication takes time O(nlogn).

If we have n processors (i.e., n copies of the gates realizing M and the adder), then we can replace the shift registers A and B by static registers. Connect the jth bit of the register to the (j-i)th corresponding input of the processor calculating c_i. (taking the (j-i) mod n). In this way all the bits of the product are produced in parallel. So, if n processors are used, the algorithm takes time O(logn). The n processors (sets of gates) are identical, so n identical chips could be used (this is good - it cuts down on development cost).

Beth [9] discusses the case where a normal basis is also a polynomial basis, and introduces an algorithm to expedite multiplication in this case. However, this basis is also an optimal normal basis, an idea that is explored in §5.6.2, so that the techniques developed there may be used, These are actually more efficient than the polynomial normal basis techniques.

5.6.2 Optimal Normal Bases

To expedite multiplication in $GF(2^n)$, when the elements are represented as coordinate vectors of a normal basis, Mullin, Onyszchuk, Vanstone and Wilson [49] introduce the concept of an optimal normal basis.

Consider the matrix λ, defined in §5.6.1, above. Let $C_N = |\{(i,j) \mid \lambda_{ij} = 1\}|$ be the number of ones in the matrix λ, for the normal basis N. The number of operations needed to multiply in $GF(2^n)$, or alternatively, the number of gates required in a hardware implementation increases as C_N gets bigger, therefore, it would be nice to have a small value for C_N. Clearly $C_N \leq n^2$ for all normal bases N.

Theorem 5.6.1. $C_N \geq 2n-1$.

Proof. Let $d = (d_0, d_1, \ldots, d_{n-1}) = \beta.\beta^{2^c}$.

Then $d_k = (0,...,0,1,0,...,0)\ \lambda \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$,

for each k, where the 1 in the first vector is in the (n-k)th position, and the one in the second vector is in the (c-k)th position. That is, $d_k = \lambda_{(-k)(c-k)}$. Let $D = (\lambda_{(-k)(c-k)})$, the matrix which has as rows the coordinates of $\beta.\beta^{2^c}$ for each i=0,..n-1. Then D has precisely the number of 1 entries as λ.

Next we note that $\text{trace}(\beta) = \beta + \beta^2 + ... + \beta^{2^{n-1}}$ is a linear operator, and hence is equal to 0 or 1. Now $\{\beta, \beta^2, ..., \beta^{2^{n-1}}\}$ is a basis, so that the sum $\beta + \beta^2 + ... + \beta^{2^{n-1}} \neq 0$, so $\text{trace}(\beta) = 1$.

So the sum of the rows of D is $\beta\ (\beta + \beta^2 + ... + \beta^{2^{n-1}}) = \beta\ \text{trace}(\beta) = \beta$. So there are an even number of ones in every column of D except the one corresponding to the coefficients of β. D does not have a column containing only zeros since $\{\beta,\beta^2,\beta^4,\beta^8, ... ,\beta^{2^{n-1}}\}$ is a basis. So D has at least 2n-1 ones in it . So $C_N \geq 2n-1$. ■

If $C_N = 2n-1$ for some basis N, then N is referred to as an *optimal normal basis*. In an optimal normal basis, the complexity of a multiplication is O(n).

Optimal normal bases do not always exist. $GF(2^n)$ has an optimal normal basis if one of the following holds:

(a) n+1 is a prime and 2 is primitive in GF(n+1).

(b) 2n+1 is a prime and 2 is primitive in GF(2n+1).

or (c) $2n+1 \equiv 3 \bmod 4$ is a prime and 2 generates the quadratic residues mod 2n+1.

In case (a), the basis consists of the non-unit (n+1)st roots of unity in $GF(2^n)$. In cases (b) and (c), the basis is generated by $\beta = \gamma + \gamma^2$, where γ is a primitive (2n+1)st root of unity in $GF(2^{2n})$.

Mullin, Onyszchuk, Vanstone, and Wilson [49] describe the above constructions in greater detail and conjecture that the above three conditions are the only ones under

which an optimal normal basis in $GF(2^n)$ exists. They also include a list of numbers $n \leq 1200$ for which an optimal normal basis is known to exist, that is, which satisfy one of these conditions, and have done a computer search through all normal bases of each $GF(2^n)$ for $n \leq 30$, and found no other optimal normal bases besides types (a), (b), and (c). Optimal normal bases have more than mere academic and asymptotic efficiency. Actual chips have been built to exploit their properties.

Optimal normal bases also exist for certain $GF(p^n)$. In particular, if n+1 is a prime and p is primitive in GF(n+1), then $GF(p^n)$ has an optimal normal basis consisting of the non-unit (n+1)st roots of unity in GF(n+1).

5.6.3 Duals of Normal Bases

Geiselmann and Gollmann [28] derive several serial input / parallel output architectures for multiplication of elements in a normal basis and the dual of the normal basis that are analogous to standard polynomial basis multipliers. These multipliers decompose the multiplication uv so that the factor v may be entered serially (starting with either the most or the least significant bit), and in which any bit of the product uv is available only at the end of the computation. Jungnickel, Menzies, and Vanstone [36] prove that a self-dual normal basis for $GF(2^n)$ over GF(2) exists when $n \equiv 0 \bmod 4$, so the conversion between one basis and the other in that case is not a problem, as it was in §5.5. However, the combination of dual basis and normal basis techniques does not yield the same efficient designs as in the case of the dual of a polynomial basis (the complexity of the multiplier is about $O(n^2)$).

5.6.4 Exponentiating in a Normal Basis

If elements of a finite field are represented as coordinate vectors with respect to a normal basis, then squaring an element is simply a cyclic shift of that coordinate vector. So it

takes one clock cycle and will hence be ignored in time calculations. So to perform an exponentiation in $GF(2^n)$ utilizing the properties of a normal basis, it seems logical to use *repeated squaring*. That is, if

$$e = \sum_{i=0}^{n-1} a_i 2^i \,,\ a_i \in \{0,1\},$$

then $$\alpha^e = \prod_{i=0}^{n-1} \alpha^{a_i 2^i} \text{ in } GF(2^n).$$

This computation requires m-1 multiplications, where $m = \sum_{i=0}^{n-1} a_i$ is the Hamming weight of (the number of ones in) the binary representation of e. Thus m = n-1 in the worst case, (where, for example, $e = 2^n - 2$), and $m = \frac{n}{2}$ on average.

Agnew, Mullin and Vanstone [2] improve the above method somewhat by selecting a positive integer k and writing e as the sum of powers of 2^k. That is, for $d = \lceil \frac{n}{k} \rceil$,

$$e = \sum_{i=0}^{d-1} b_i 2^{ki} \,,\ \text{where } b_i \text{ is a binary k-tuple.}$$

The equal values of b_i are collected together, so

$$e = \sum_{\omega} \omega\lambda(\omega) \,,\ \text{where } \omega \text{ runs over all non-zero binary k-tuples}$$

$$\text{and } \lambda(\omega) = \sum_{i=0}^{d-1} C_{i,\omega} 2^{ki} \,,\ \text{for some } C_{i,\omega} \in \{0,1\}.$$

Then $\alpha^e = \alpha^{\sum \omega\lambda(\omega)} = \prod_{\omega} (\alpha^{\omega})^{\lambda(\omega)}$.

So the algorithm can be summarized as follows.

> **Algorithm Exponentiate.**
> **Stage I.**
> compute α^{ω}, $1 \le \omega \le 2^k - 1$.
> **Stage II.** multiply $(\alpha^{\omega})^{\lambda(\omega)}$ together.

Stage I is further refined by Stinson [62] as follows.

Algorithm Stage I.
Step 1. $\alpha^1 = \alpha$
Step 2. for $\omega := 2$ to 2^k-1 do

If ω is even then $\alpha^{\omega} := \left(\alpha^{\frac{\omega}{2}}\right)^2$ {a cyclic shift !}
If ω is odd then $\alpha^{\omega} := \alpha^{\omega-1}\,\alpha$.

So then, to calculate α^{ω} for $1 \le \omega \le 2^k-1$, by the algorithm **Stage I** takes at most $2^{k-1}-1$ field multiplications. This gives the worst case complexity of the algorithm **Exponentiate** as $M(k) = 2^{k-1} + \left\lceil \frac{n}{k} \right\rceil - 2$ multiplications.

Example 5.6.3. Suppose $e = 1499_{10} = 10111011011_2$, and $k = 2$ then

$$e = (1)\,2^{10} + (1)\,2^8 + (1+2)\,2^6 + (1)\,2^4 + (2)\,2^2 + (1+2)\,2^0$$

$$= (1)\,(2^{10} + 2^8 + 2^4) + (2)\,(2^2) + (1+2)\,(2^6 + 2^0).$$

So for $\omega = 1$, $\lambda(\omega) = 2^{10} + 2^8 + 2^4$.

$\omega = 2$, $\lambda(\omega) = 2^2$.

$\omega = 3$, $\lambda(\omega) = 2^6 + 2^0$.

Hence $\alpha^e = (\alpha)^{2^{10}}(\alpha)^{2^8}(\alpha)^{2^4}\left(\alpha^2\right)^{2^2}\left(\alpha^3\right)^{2^2}\left(\alpha^3\right)^{2^0}$.

So to calculate α^{1499} takes 5 (field) multiplications.

To make this algorithm highly parallel, the following definition is needed.

Suppose that we have at least $\left\lfloor \frac{m}{2} \right\rfloor$ processors and we wish to multiply m objects together. If they are multiplied together in pairs, after one round there are $\left\lceil \frac{m}{2} \right\rceil$ objects remaining. If this process is iterated, the m objects can be multiplied in $\lceil \log_2 m \rceil$ rounds. This process is called *binary fan-in multiplication*. If we only have $p < \left\lfloor \frac{m}{2} \right\rfloor$ processors available to multiply the m objects, then the obvious adaptation of binary fan-in multiplication takes $\left\lceil \frac{m-2p_0}{p} \right\rceil - \log_2 p_0 + 1$ rounds, where $p_0 = \lfloor \log_2 p \rfloor$.

If we have $\lfloor \frac{s}{2} \rfloor$ processors, where $s = \lceil \frac{n}{k} \rceil$, then we can perform **Stage II** in $\lceil \log_2 s \rceil$ rounds.

Stage I is a little more complicated. If we have $(k-1)2^{k-2}$ processors, then the algorithm would take k rounds if each α^{ω} is calculated simultaneously using binary fan-in multiplication. However, we can improve this.

Algorithm Parallel Stage I.
Step 1. Compute α^{ω} for $1 \le \omega \le 2^j - 1$.
Step 2. For each α^{ω}, $1 \le \omega \le 2^{k-j} - 1$, find $(\alpha^{\omega})^{2^j}$
{cyclic shifts}
Step 3. Multiply all terms found in **Step 1**
by all terms found in **Step 2.**

This algorithm takes time $\lceil \log_2 k \rceil$ on $P(k) = (2^j - 1)(2^{k-j} - 1)$ processors.

If k is even,

$$\text{then } P(k) = \left(2^{\frac{k}{2}} - 1\right)^2,$$

and if k is odd,

$$\text{then } P(k) = \left(2^{\frac{k+1}{2}} - 1\right)\left(2^{\frac{k-1}{2}} - 1\right)$$

So if we have available $\max\left\{\lfloor \frac{s}{2} \rfloor, P(k)\right\}$ processors, then the algorithm **Exponentiate** can be performed in $S(k) = \lceil \log_2 k \rceil + \lceil \log_2 s \rceil$ rounds of multiplications. We note that $\lfloor \log_2 n \rfloor \le S(k) \le \lceil \log_2 n \rceil$. So instead of being interested in k such that S(k) is minimal, we are instead concerned with finding k such that the number of processors required is minimal.

If k is $\log_2 n - \log_2 \log_2 n$, then

$$M(k) \text{ is } O\left(\frac{n}{\log_2 n - \log_2 \log_2 n}\right),$$

and the number of processors required for the algorithm **Parallel Stage II** is also

$$O\left(\frac{n}{\log_2 n - \log_2 \log_2 n}\right).$$

Much improvement may be made if we have 2^k processors, for some $2^k < \left\lfloor \frac{s}{2} \right\rfloor$. Then the algorithm takes $T(k) = \lceil \log_2 k \rceil + \left\lfloor \frac{s}{2^k} \right\rfloor + k - 1$ rounds.

Some values for M(k), S(k), the minimum number of processors for S(k), and T(k) are tabulated in Appendix 2 for various values of n.

5.6.5 A Normal Basis Inverse

Any non-zero element α of $GF(2^n)$ has an inverse

$$\alpha^{-1} = \alpha^{2^n - 2} = \alpha^{\sum_{i=1}^{n-1} 2^i} = \prod_{i=1}^{n-1} \alpha^{2^i}.$$

This can be computed in n - 2 multiplications in a normal basis using repeated squaring. Agnew, Mullin, and Vanstone [2] propose the following method to reduce this.

Suppose that n-1 = gh, for some integers g and h. Then

$$2^{n-1} - 1 = 2^{gh} - 1 = (2^g - 1)\left(\sum_{i=0}^{h-1} 2^{gi}\right) \text{ and } \alpha^{-1} = \alpha^{2^n - 2} = \left(\alpha^2\right)^{(2^g - 1)(\sum_{i=0}^{h-1} 2^{gi})}.$$

Now $\gamma = \alpha^2$ is calculated "free". Then $\omega = \gamma^{(2^g - 1)}$ takes g-1 multiplications to calculate.

Also $\alpha^{-1} = \omega^{\sum_{i=0}^{h-1} 2^{gi}}$ takes h-1 multiplications to calculate.

So computing α^{-1} takes g+ h - 2 multiplications. This process may be iterated if g is composite. So if $n-1 = \prod_{i=1}^{a} p_i$ is the prime factorization of n-1 then we require $(\sum_{i=1}^{a} p_i) - a$ steps.

Example 5.6.4. For example, in $GF(2^{361})$,

$2^{360} - 1 = (2^{72} - 1)\sum_{i=0}^{4} 2^{72i}$, since $360 = 72 \cdot 5$.

$2^{72} - 1 = (2^{24} - 1)\sum_{i=0}^{2} 2^{24i}$, since $72 = 24 \cdot 3$.

$2^{24} - 1 = (2^8 - 1)\sum_{i=0}^{2} 2^{8i}$, since $24 = 8 \cdot 3$.

$2^8 - 1 = (2^2 - 1)(2^2 + 1)(2^4 + 1)$, since $8 = 2 \cdot 2 \cdot 2$.

So raising any element to the power $2^{361} - 2$ takes

0 multiplications to square,

3 multiplications to raise the result to $2^8 - 1$,

2 multiplications to raise the result to $\sum_{i=0}^{2} 2^{8i}$,

2 multiplications to raise the result to $\sum_{i=0}^{2} 2^{24i}$,

and 4 multiplications to raise the result to $\sum_{i=0}^{4} 2^{72i}$,

for a total of 11 multiplication steps.

Compared this to the 360 multiplications required to calculate an inverse using repeated squaring.

Algorithm (n-1) Small Primes Inverse.
Input: $g \in GF(2^n)\backslash\{0\}$.
Output: $g^{-1} \in GF(2^n)\backslash\{0\}$.
Given: $n-1 = \prod_{i=1}^{a} p_i$ is the prime factorization of n-1.
Step 0. {initialization.}
g^{-1} := **Raise**(a,g).

(n-1) Small Primes Inverse calls the recursive algorithm **Raise.**

Algorithm Raise.

Input: a, a non-negative integer,

$\alpha \in GF(2^n)\backslash\{0\}$.

Output: $\beta \in GF(2^n)\backslash\{0\}$.

Step 1. {terminating case}

If $a = 0$ then $\beta = \alpha$. End.

Step 2. {recursive call}

$k := \prod_{j=1}^{a-1} p_i$

$\gamma := \alpha^{\sum_{i=0}^{a-1} 2^{ki}}$

$\beta :=$ **Raise**$(a-1,\gamma)$

The algorithm can be modified when n-1 does not factor into only small primes. We find an x such that m-x factors into small primes. Then $2^{m-1} - 1 = 2^{x-1}(2^{m-x} - 1) + 2^{x-1} - 1$.

Let $m-x = \prod_{i=1}^{a} p_i$ and $x-1 = \prod_{i=1}^{b} q_j$.

So we compute α^{-1} in $\sum_{i=1}^{a} p_i + \sum_{i=1}^{b} q_j - a - b + 1$ multiplication steps.

Example 5.6.5. For example, in $GF(2^{1279})$,

$2^{1279} - 1 = 2^6(2^{24.53} - 1) + 2^6 - 1$.

Noting that $2^{24} - 1 = (2^{12} + 1)(2^6 + 1)(2^3 + 1)(2^3 - 1)$ requires 5 multiplications to evaluate,

$\sum_{i=0}^{52} 2^{24i}$ contributes 52 multiplications,

$2^6 - 1 = (2^3 + 1)(2^3 - 1)$ requires 3 multiplications,

and we need one multiply to put it all together, α^{-1} is computed in 61 multiplications.

This is equal to $(2 + 2 + 2 + 3 + 53) - 5 + (2 + 3) - 2 + 1$, as expected.

Note that this algorithm seems inherently sequential.

Algorithm (n-x) Small Primes Inverse

Input: $g \in GF(2^n)\backslash\{0\}$.

Output: $g^{-1} \in GF(2^n)\backslash\{0\}$.

Given: an integer $x \geq 1$.

$n\text{-}x = \prod_{i=1}^{a} p_i$ is the prime factorization of n-x.

$x\text{-}1 = \prod_{i=1}^{b} q_i$ is the prime factorization of x-1.

Step 0. {initialization.}

$g^{-1} := (\textbf{Raise}(a,g))^{2^{x-1}}.\textbf{Raisex}(b,g)$

This algorithm requires **Raise** again, and a **Raisex** very much like **Raise**, but with b replacing a and q_i replacing p_i.

5.6.6 Divide and Conquer - Another Normal Basis Inversion

Consider the scheme proposed by Itoh, Teechai and Tsujii, (originally in Japanese, so as quoted in [2]).

Note that $2^{m-1} - 1 = \begin{cases} (2^{\frac{m-1}{2}} - 1)(2^{\frac{m-1}{2}} + 1), \text{ when m is odd,} \\ 2^{m-2} + (2^{\frac{m-2}{2}} - 1)(2^{\frac{m-2}{2}} + 1), \text{ when m is even.} \end{cases}$

For m odd then, we require one multiply to compute $\alpha^{2^{m-1}-1}$,

given that we have computed $\alpha^{2^{\frac{m-1}{2}}-1}$.

For m even, we require two multiplications to compute $\alpha^{2^{m-1}-1}$,

given that we have computed $\alpha^{2^{\frac{m-2}{2}}-1}$.

Using this procedure recursively, it is a simple inductive proof to show that the number of multiplications required to find an inverse in $GF(2^n)$ (that is, to raise an element to 2^n-2) is $\log_2(n-1) + d - 2$, where d is the Hamming weight of the binary representation of n-1.

Example 5.6.6. For example,

$2^{360} - 1 = (2^{180} - 1)(2^{180} + 1)$

$2^{180} - 1 = (2^{90} - 1)(2^{90} + 1)$

$2^{90} - 1 = (2^{45} - 1)(2^{45} + 1)$

$2^{45} - 1 = 2^{44} + (2^{22} - 1)(2^{22} + 1)$

$2^{22} - 1 = (2^{11} - 1)(2^{11} + 1)$

$2^{11} - 1 = 2^{10} + (2^{5} - 1)(2^{5} + 1)$

$2^{5} - 1 = 2^{4} + (2^{2} - 1)(2^{2} + 1)$

$2^{2} - 1 = (2 + 1)$

So raising anything to the $2^{361} - 2$, and hence finding an inverse in $GF(2^{361})$ takes 11 operations using this algorithm (since there are 11 addition signs in the final expression for $2^{361} - 2$.

We summarize the algorithm as ITT inverse, as follows.

Algorithm ITT Inverse.
Input: $g \in GF(2^n)\backslash\{0\}$.
Output: $g^{-1} \in GF(2^n)\backslash\{0\}$.
Step 0. {initialization}
g^{-1} = **ITT Raise** (g,n-1).

ITT Inverse needs a recursive algorithm **ITT Raise** to make it work.

Algorithm ITT Raise.

Input: i, a positive integer,

$\alpha \in GF(2^n)\backslash\{0\}$.

Output: $\beta \in GF(2^n)\backslash\{0\}$.

Step 1. {terminating case}

If i = 1 then $\beta := \alpha$. End.

Step 2. {recursive call}

If i is even then $\beta := \alpha^{2^{m-2}}$.**ITT Raise**$(\frac{i}{2}, \alpha^{2^{\frac{i}{2}}+1})$.

If i is odd then $\beta :=$ **ITT Raise**$(\frac{i-1}{2}, \alpha^{2^{\frac{i-1}{2}}+1})$

5.7 New Algorithms to Find Discrete Logarithms

Given a generator α of $GF(2^n)$, and x a non-zero element of $GF(2^n)$, we define $\log_\alpha x$, the discrete logarithm of x with respect to α, to be the number y, $0 \le y \le 2^n-1$, such that $x = \alpha^y$ in $GF(2^n)$. Many heuristic and probabilistic algorithms have been proposed to calculate $\log_\alpha x$ for a fixed α and arbitrary x, with variable results in terms of efficiency. The most successful logarithm algorithms are discussed.

5.7.1 The Index-Calculus Algorithm

In 1979, Aldeman proposed the first sub-exponential time log algorithm, the index-calculus algorithm. This was improved by Coppersmith [21] in 1984 to one that runs in time $O\left(e^{c\sqrt[3]{n\log^2 n}}\right)$ where c is a small constant. We present the algorithm as it applies to $GF(2^n)$, but it can be modified to work for any Galois field.

The index-calculus algorithm is a probabilistic method. It relies on setting up a data base of logarithms for some subset of S of $GF(2^n)\backslash\{0\}$, where the elements of S have some reasonably easy to determine property, and then using heuristics to reduce the element of which the logarithm is being found to a combination of elements of S.

Let $GF(2^n)$ be considered as the ring of polynomials modulo some irreducible g(x) over GF(2). Let f(x) be a generator of the multiplicative group $GF(2^n)\backslash\{0\}$. We note that polynomials over GF(2) are easy to factor. Algorithms are known that factor a polynomial of degree m over GF(2) in time O(m). S is usually chosen to be the set of all, (or most) irreducible polynomials over GF(2) with degree less than some number m.

Algorithm Index-Calculus
Input: $\alpha \in GF(2^n)\backslash\{0\}$
Output: $a = \log_\beta \alpha$.
Stage I. Compute the logarithms of all the elements in S.
Stage II. Reduce the element α to a combination of elements in S.
Combine the logarithms of these elements to calculate a.

We describe the standard algorithm without the Coppersmith variations first, and then sketch an outline of those changes.

Algorithm Stage I.
Step 1. Choose a random s, $1 \le s \le 2^n-1$.
Step 2. Set $\alpha^* \equiv \beta^s \pmod{f(x)}$.
If α^* factors into irreducibles from S, then insert

$$s \equiv \sum_{v \in S} b_v(\alpha^*)\log_\beta v \pmod{2^n-1}$$

into the set of congruences to solve
Step 3. If we have determined more than |S| congruences then continue to **Step 4**, else return to **Step 1**.
Step 4. Solve the set of congruences to determine $\log_\beta v$, for each v in S.

Note that 2^n-1 may not be prime, in which case the congruences are solved for each prime power divisor and then combined using the Chinese Remainder Theorem. This would actually speed up the computation since smaller primes are being worked with, so we assume for algorithm analysis purposes that 2^n-1 is prime.

If **Step 4** is done with straight Gaussian elimination it takes $O(|S|^3)$ steps, but if the fact that the set of congruences is a sparse system is exploited, then the time can be reduced to $O(|S|^2)$. The number of irreducible polynomials over GF(2) of degree less than m is approximately $\frac{2^{m+1}}{m}$, so **Step 4** takes time

$$O\left(\left|\frac{2^{m+1}}{m}\right|^2\right)$$

The polynomials α^* behave like random polynomials. Define p(k,m) to be the probability that a polynomial over GF(2) of degree k has all its irreducible factors of degree less than m. Then the running time of the first three steps of **Stage I** is $\frac{|S|}{p(n,m)}$. Odlyzko, in his definitive survey paper [52] analyses p(k,m) in terms of N(k,m), where N(k,m) is the number of polynomials over GF(2) of degree k which have all its irreducible factors of degree less than m. He determines the recurrence relation

$N(0,0) = 1,$

$N(k,0) = 0,$ if $k \neq 0,$

$N(k,m) = 0,$ if $k,0$ and $m \geq 0,$

and for all $n,m>0,$

$$N(n,m) = \sum_{k=1}^{m} \sum_{r \geq 1} N(n-rk,k-1)\binom{r+I(k)-1}{r}.$$

He then goes on to determine that, if $n^{\frac{1}{100}} \leq m \leq n^{\frac{99}{100}}$, then

$$N(n,m) \rightarrow 2^n \left(\frac{m}{n}\right)^{(1+o(1))\frac{n}{m}} \text{ as } n \rightarrow \infty,$$

so that $p(n,m) \rightarrow \left(\frac{m}{n}\right)^{(1+o(1))\frac{n}{m}}$ in the same range.

Odlyzko [52] also has a tabulation of p(n,m) for small values of n and m.

This gives us the average running time of (the unmodified version of) **Stage I** as

$$O\left(2^m\left(\frac{n}{2m}\right)^{\frac{n}{m}} + 2^{2m}\right).$$

Algorithm Stage II.
Input: $\alpha \in GF(2^n)\backslash\{0\}$
Output: $a = \log_\beta \alpha$.
Step 1. Choose a random s, $1 \le s \le 2^n-1$.
Step 2. Set $\alpha^* \equiv \alpha\beta^s \pmod{f(x)}$. ($\deg \alpha^* \le n$)
If α^* factors into irreducibles from S, then

$$a \equiv \sum_{v \in S} b_v(\alpha^*)\log_\beta v \; - s \qquad (\mathrm{mod}\ 2^n-1)$$

else repeat from **Step 1.**

The probability that α^* will factor as required is

$$\sum_{k=1}^{n} 2^{-k} p(n-k,m),$$

which is asymptotically equivalent to p(n,m) as $n \to \infty$, if $n^{\frac{1}{100}} \le m \le n^{\frac{99}{100}}$, (see Odlyzko [52]). So the expected running time of **Stage II** is

$$\frac{1}{p(n,m)} = \left(\frac{n}{m}\right)^{(1+o(1))\frac{n}{m}}.$$

There are ways to speed up this algorithm. Blake, Fuji-Hara, Mullin, and Vanstone [14] replace factoring α^* in each stage by finding two polynomials w_1 and w_2 such that $\alpha^* = \frac{w_1}{w_2}$ and the degree of each w_i is about $\frac{n}{2}$. This is done using information gained by applying the extended Euclidean algorithm. Each w_i is then factored as in the standard algorithm. This adaptation is faster than the standard version by a factor of about $2^{\frac{n}{m}}$, which is a great practical saving, but not an asymptotic one.

Other improvements are discussed in Odlyzko [52]. The one that has had the greatest impact is the Coppersmith algorithm, presented by Coppersmith [21], and analyzed in great detail by Odlyzko.

The Coppersmith algorithm begins by choosing the generating polynomial f(x) in such a way that $f(x) = x^n + f'(x)$, where the degree of f'(x) is small. It relies on the factorization of two polynomials of degree $\sqrt[3]{n^2}$.

Algorithm Coppersmith Stage I.

Step 1. Choose a real number $k > 0$ such that 2^k is about $\sqrt[3]{\frac{n}{\log_e n}}$.

$h := \lfloor \frac{n}{2^k} \rfloor + 1$.

Choose B such that B is about $\sqrt[3]{n \log_2 n}$.

Step 2. Choose $u_1(x)$ and $u_2(x)$, relatively prime and of degree less than B.

$w_1 := u_1(x)x^h + u_2(x)$

$w_2 := (w_1(x))^{2^k} \quad \text{mod}(f(x))$

If w_1 and w_2 have all their linear factors in S, then we get a linear equation in $\log_\beta v$, for the v in S.

Step 3. Repeat **Step 1** and **Step 2** until we have slightly more than |S| equations.

Algorithm Coppersmith Stage II.

Step 1. Construct successive α^* as above, until one is found whose irreducible factors u_i are all of degree $\leq m_1 = \sqrt[3]{n^2 \log n}$.
(or alternatively, until the Blake et al modification produces w_1 and w_2 with irreducibles u_i of degree $\leq m_1$).

Step 2. Choose a positive real number k such that 2^k is about $\sqrt[3]{\frac{n}{B}}$.

$d := \left\lfloor \frac{n}{2^k} \right\rfloor + 1$.

Choose B such that B is about $\sqrt[3]{n \log_2 n}$.

Step 2. Choose $v_1(x)$ of degree $\leq B$, find a relatively prime $v_2(x)$ of degree $\leq B$, such that $u_{i(}x)$ divides $w_1(x)$, where

$w_1 := v_1(x)x^d + v_2(x)$

$w_2 := (w_1(x))^{2^k} \mod(f(x))$

If w_1 and w_2 factor into polynomials of low degree, then find the logarithms of these factors using the algorithm **Stage II**, above, else repeat **Step 2**.

We note that $w_2 = v_1(x^{2^k})x^{d2^k - n}f_1(x) + v_2(x)$, so that the degrees of both w_1 and w_2 are $O\left(n^{\frac{2}{3}}\right)$. There are improvements to the Coppersmith algorithm that speed up the first stage some constant number of times, and are outlined in Odlyzko [52] but these improvements do not affect the asymptotic running time of the algorithm.

5.7.2 The Pohlig-Hellman Algorithm

If p is a prime such that p-1 has only small prime factors, then there is an algorithm that can compute logarithms in GF(p) in time $O(\log_2 p)$. It was first published by Pohlig and Hellman in 1978 [54], who credit the earlier independent discovery to Ronald Silver. The algorithm was later generalized to the field GF(q) where q is any prime power and q-1 has only small prime factors (see Odlyzko [52] or van Tilborg [67] for a fairly good

description). Let $GF(q) = GF(p^n)$, for some prime p. Let $p^n - 1 = \prod_{i=1}^{k} p_i^{n_i}$, where p_i are distinct primes. If $r_1,...,r_k$ are any real numbers with $0 \le r_i \le 1$ then logarithms over $GF(p^n)$ can be calculated in time

$$O\left(\sum_{i=1}^{k} n_i\left(\log_2 q + p_i^{1-r_i}(1+\log_2 p_i^{r_i})\right)\right),$$

using

$$O\left(n \sum_{i=1}^{k}(1+p_i^{r_i})\right)$$

bits of memory, provided that a precalculation requiring time

$$O\left(\sum_{i=1}^{k}\left(p_i^{r_i}\log_2 p_i^{r_i} + n\right)\right)$$

is performed.

For example, if the time taken for the algorithm is proportional to $\sqrt{\max}$, where max the largest prime factor of p^n-1, and the amount of storage required is also proportional to $\sqrt{\max}$.

To sketch an outline of the algorithm, let β be a primitive element of $GF(p^n)$, and let x be any non-zero element of $GF(p^n)$. The aim is to find $y = \log_\beta x$, that is, where, $\beta^y = x$. The algorithm can be summarized as follows.

Algorithm Pohlig-Hellman.

Input: x, β, n, where we want to find $\log_\beta x$ in GF (2^n).

$$2^n - 1 = \sum_{i=1}^{k} p_i^{n_i}.$$

r_i , the weight used in the step corresponding $p_i^{n_i}$, $0 \le r_i \le 1$.

Step 1. For $1 \le i \le k$ **determine y mod** $p_i^{n_i}$.

Step 2. Use the Chinese Remainder Theorem to find y mod $\sum_{i=1}^{k} p_i^{n_i}$.

Which leaves the problem of determining y mod $p_i^{n_i}$, for each i. We put

$$y \equiv \sum_{j=0}^{n-1} b_j p_i^j \pmod{p_i^{n_i}},$$

and set about determining b_j. To find b_0, calculate

$$a = x^{\frac{q-1}{p_i}} = \beta^{y\frac{q-1}{p_i}} = \left(\beta^{\frac{q-1}{p_i}}\right)^{b_0}, \text{ since } y \equiv b_0 \bmod p_i.$$

Note that a can only be one of p_i elements. Set $\alpha = \beta^{\frac{q-1}{p_i}}$. The baby steps-giant steps technique pioneered by Shanks is used to find b_0 given $a = \alpha^{b_0}$. Given a real number r, $0 \le r \le 1$, find $u = \lceil (p_i)^r \rceil$. Then there must be a unique pair of integers c and d, $0 \le d \le u-1$, $0 \le c \le \frac{p_i}{u}$, such that $y = cu + d$. So then $a = \beta^{b_0}$ means that $\alpha^{-cu}a = \alpha^d$. So α^d is precomputed for each d, $0 \le d \le u-1$, and these values are then sorted. This step can be done in $O\left((p_i)^2 \log_2 p_i\right)$.

Compute $a\alpha^{-cu}$ for c = 0, 1, ... (up to $\left\lfloor \frac{p}{u} \right\rfloor$ if necessary), and check each result for a match with the α^d values. There are

$$O\left((p_i)^{1-r}\right)$$

of these $a\alpha^{-cu}$ to be computed. So once b_0 has been found, set

$$\left(x\beta^{-b_0}\right)^{\frac{q-1}{p_i^2}} = \alpha^{b_1}, \text{ and iterate.}$$

It is easy to see that if q-1 has a large prime factor, the Pohlig-Hellman algorithm becomes infeasible in terms of precalculation time and storage requirements.

Algorithm determine y mod $p_i^{n_i}$.

Input: x, β, n, where we want to find $\log_\beta x$ in GF (2^n).

p_i, n_i

r_i, determines the size-storage tradeoff in the computation.

Step 1a. {initialization}

$a := x^{\frac{q-1}{p_i}}$

$\alpha := \beta^{\frac{q-1}{p_i}}$

$u := \lceil (p_i)^r \rceil$

$\gamma := \alpha^{-u}$

$j := 0$

Step 1b. {precomputation}

for $0 \le d \le u-1$ find α^d.

Sort these.

Step 1c. {find $b_i = \log_\alpha a$}

for $0 \le c \le \lfloor \frac{p}{u} \rfloor$,

calculate $a\gamma^c$

if $a\gamma^c = \alpha^d$ for any calculated in **Step 1b**, then $bj := cu + d$

and go to **Step 1d.**

Step 1d. {iterate}

$j := j+1$

if $j = n_i$ then go to **Step 1e.**

else $a := \left(x\beta^{-b_0}\right)^{\frac{q-1}{(p_i)^j}}$ and go to **Step 1c.**

Step 1e. {summary}

$$y := \sum_{j=0}^{n-1} b_j (p_i)^j \pmod{(p_i)^{n_i}},$$

Various steps of the Pohlig-Hellman algorithm can be executed in parallel. If we have k processors, where there are k distinct primes dividing q-1, and letting $r_i = \frac{1}{2}$, for all i, then the algorithm can be performed in time $O(\log_2^2 q + \sqrt{P} \log_2^2 q \log_2 P)$, where

$P = \max \{p_i \mid 1 \le i \le k\}$ is the largest prime dividing q-1, with a precalculation of time $O(\log_2 q + \sqrt{P} \log_2 P)$ using memory $O(k \sqrt{P} \log_2 q)$. For more details of the derivation of these numbers, see von Tilborg [67] and adapt his theorems to the multiprocessor case.

5.8 Generating Random Elements

The most obvious way to select a random element of GF(q) from an equiprobable distribution is to select a random integer r, $0 \le r \le q$, and if $r = q$ then the random element is 0, else the random element is α^r, where α is a fixed primitive element of GF(q). Then the complexity of finding a random element is the complexity of exponentiating.

Alternatively, if $GF(q) = GF(p^n)$, for some prime p is represented as an n-vector of elements of GF(p) with respect to some basis, then an alternate approach is to choose n random elements of GF(p), that is n integers between 0 and p-1 (inclusive). This algorithm has complexity O(n).

5.9 Comparisons and Conclusions

Many of the most efficient algorithms that have been discussed only work in specific fields. For example, the Pohlig-Hellman logarithm algorithm and the efficient normal basis multipliers (the optimal normal bases and the self-dual normal bases). Appendix 2 shows that it is reasonable to discuss a case where both may be implemented for the same field.

The following table summarizes the algorithms for manipulation of elements in the finite fields $GF(2^n)$ that have been discussed in this chapter.

operation	algorithm	time	space	special requirements
add	traditional	O(n)	O(1)	
add	parallel traditional	O(1)	O(n)	
multiply	traditional	O(n)	$O(n^2)$	polynomial basis
multiply	traditional	O(n)	O(n)	polynomial basis, trinomial generating polynomial
exponent	repeated squaring	O(n) mults		
inverses	Euclid	$O(n^2)$		polynomial basis
logarithm	Table lookup	O(n)	$O(n2^n)$	
multiply	FFT	O(nlogn)	O(1)	polynomial basis, trinomial generating polynomial
exponent	von zur Gathen	O(logn)	$O(n^4 \log n)$	polynomial basis
multiply	self-dual basis	O(n)	O(n)	self-dual basis
multiply	normal basis	O(n)	$O(n^2)$	normal basis
multiply	parallel normal	O(1)	$O(n^3)$	normal basis
multiply	optimal normal	O(n)	O(n)	optimal normal basis
multiply	parallel ONB	O(1)	$O(n^2)$	optimal normal basis
exponent	divide & conquer	$O\left(\frac{n}{\log n}\right)$		normal basis
exponent	parallel d & c	O(logn)	$O\left(\frac{n}{\log n}\right)$	normal basis
inverse	normal basis	O(P)		normal basis, 2^n-1 has largest prime factor P.
exponent	divide & conquer	O(logn)		normal basis.
logarithm	index-calculus	$O\left(e^{c\sqrt[3]{n\log^2 n}}\right)$	$O\left(e^{c\sqrt[3]{n\log^2 n}}\right)$	
logarithm	Pohlig-Hellman	O(n)		2^n-1 has only small prime factors.
logarithm	parallel P-H	O(n)	O(k)	2^n-1 has k small prime factors.
random	exponentiation	as exp.		
random	lexicographical	O(n)		

The following table summarizes the complexities of the (single processor) algorithms to perform operations in $GF(p^n)$, where $p \geq 3$.

operation	algorithm	time	special requirements
add	traditional	O(nlogp)	
multiply	traditional	O(nlogp)	polynomial basis
exponent	repeated squaring	O(n) mults	
inverses	Euclid	$O(n^2 \log p)$	polynomial basis
logarithm	Table lookup	O(n)	
multiply	FFT	O(nlognlogp)	polynomial basis
multiply	self-dual basis	O(nlogp)	self-dual basis.
multiply	normal basis	O(nlogp)	normal basis
multiply	optimal normal basis	O(nlogp)	optimal normal basis.
exponent	divide & conquer	$O\left(\frac{n \log p}{\log n}\right)$	normal basis
inverse	normal basis	O(Plogp)	normal basis, p^n-1 has largest prime factor P.
logarithm	index-calculus	$O\left(e^{c\sqrt[3]{n \log^2 n}}\right)$	
logarithm	Pohlig-Hellman	O(nlogp)	2^n-1 has only small prime factors.
random	exponentiation	as exp.	
random	lexicographical	O(n)	

The following table summarizes the complexity of the most efficient algorithm to perform each operation in $GF(2^n)$, for a single processor, and the multiprocessor cases.

operation	single processor	multiple processor
add	O(n)	O(1)
multiply	O(n)	O(1)
exponent	$O\left(\frac{n}{\log n}\right)$	O(logn)
logarithm	O(n)	O(n)
inverse	$O\left(\frac{n}{\log n}\right)$	O(logn)
random	$O\left(\frac{n}{\log n}\right)$	O(logn)

The following table summarizes the complexity of the most efficient algorithm to perform each operation in $GF(p^n)$, (for a single processor).

operation	complexity
add	O(nlogp)
multiply	O(nlogp)
exponent	$O\left(\frac{n^2 \log p}{\log n}\right)$
logarithm	$O(n^2 \log p)$
inverse	$O\left(\frac{n^2 \log p}{\log n}\right)$
random	$O\left(\frac{n^2 \log p}{\log n}\right)$

Chapter 6 : Conclusions

The following table summarizes the field operations required for the algorithms in Chapter 4.

Algorithm	field	exp	log	add	mult	rand	inv
4.2.	GF(q)	1	1	2		1	
4.4.	none.						
4.5.	GF(q)	1	1	2			1
4.6.	GF(q)	1	1	1	2	3	
4.7.	GF(q)	1	1				
	$GF(q^d)$	1		6	11	3	
4.8.	GF(q)	1	1	4	4	2	1
4.9.1.	GF(q)	1	1	4	4	3	
4.9.2.	GF(q)	1	1	1			1
	$GF(q^3)$	1	1				
4.10.	GF(q)	1	1				
	$GF(q^d)$			4	4		2
	GF(q+1)	2	2	2	1	2	1
4.11.	GF(q)	2t		2t	2t	t	
4.12.	$GF(q^d)$			2d	2d	d	2

The following table summarizes the complexity for the algorithms in Chapter 4 which may be implemented over GF(q) where $q = 2^n$ for the single and multiprocessor case as outlined in §5.9.

Algorithm 4.10 is listed only for the operations in GF(q+1), where $q+1 = 2^n$.

Algorithm	Single processor complexity	Multiprocessor complexity
4.7	$O\left(n + \frac{nd}{\log nd}\right)$	O(n + lognd)
4.8	O(n)	O(n)
4.9.1	O(n)	O(n)
4.9.2	O(n)	O(n)
4.10	O(n)	O(n)
4.11	$O\left(\frac{nt}{\log n}\right)$	O(tlogn)
4.12	$O\left(\frac{nd}{\log n}\right)$	O(dlogn)

The following table summarizes the complexity for the algorithms in Chapter 4 which are implemented over $GF(p^n)$, where $p \neq 2$. In algorithm 4.10, operations are performed over GF(p), $GF(p^d)$, and $GF(2^n)$, and logp is O(n)

Algorithm	Complexity
4.2	$O(n^2 \log p)$
4.5	$O(n^2 \log p)$
4.6	$O(n^2 \log p)$
4.10	$O(d^2 n)$

Appendix 1 :
The best Choice of Irreducible generating Polynomial for GF(2^n), $2 \leq n \leq 16$

When multiplications are performed in a field represented as coefficients of a polynomial basis, the algorithm is more efficient if the generating polynomial has as few terms as possible. Mastrovito [45] lists the following table for the best generating polynomial for GF(2^n), for small n.

n	powers of x in the generating polynomial.
2	2,1,0
3	3,1,0
4	4,1,0
5	5,2,0
6	6,3,0
7	7,1,0
8	8,7,5,1,0
9	9,1,0
10	10,3,0
11	11,2,0
12	12,3,0
13	13,7,6,1,0
14	14,5,0
15	15,1,0
16	16,11,6,5,0

Appendix 2 :
Optimal Normal Bases and Pohlig-Hellman logarithms: Can we have both in GF(2^n)?

Two of the most efficient finite field algorithms that were discussed in Chapter 5 only work in GF(2^n) for certain values of n. Optimal normal bases exist when n+1 is a prime and 2 is primitive in GF(n+1), when 2n+1 is a prime and 2 is primitive in GF(2n+1), or when 2n+1 is a prime congruent to 3 modulo 4 and 2 generates the quadratic residues mod 2n+1. Luckily finding which of the small integer n ($n < 1000$) satisfy these rather complicated conditions is tabulated for us by Mullin, Onyszchuk, Vanstone and Wilson [49]. The Pohlig-Hellman algorithm is efficient when 2^n-1 has no large prime factors. Brillhart, Lehmer, Selfridge, Tuckerman and Wagstaff [20] list the prime factors of 2^n-1 for many values of n. So if we are to justify the claim that both algorithms can be used on certain Galois fields of characteristic 2, then the task remains to compare and correlate these two lists of numbers.

We note also, that in implementing the (1,1)-code constructed using PG(q), we needed field operations in PG(q^3) as well as those in GF(q), and in implementing the (3,2)-code and the (2,2)-code we used both GF(q) and GF(q^d). So a search is also conducted for small n and 3n which both satisfy the above two conditions. Actually, since 2^n-1 divides 2^{dn}-1, if 2^{dn}-1 has only small prime factors, so too does 2^n-1. Some of the work in the precalculation stage of the Pohlig-Hellman algorithm need not be duplicated for the second.

Small values of n, ($n < 1000$), for which an optimal normal basis exists for $GF(2^n)$, and for which 2^n-1 has only small prime factors.

Note: d is the number of decimal digits in the largest prime factor of 2^n-1.

C before a number signifies that it is the number of digits in a number that is known to be composite but for which the factorization is not known.

n	d	n	d	n	d
4	1	273	20	585	C78
6	1	292	22	586	C77.C89
12	2	299	C60	615	C89
18	2	316	24	618	C62
28	3	323	C80	645	C88
36	3	326	24	5658	8.C81
52	4	348	17	660	C90
60	4	354	19	4726	5.C63
66	6	372	19	774	C56.C76
70	6	388	26	810	C60
72	5	411	C63	820	49
81	8	420	16	828	40
90	8	429	C63	852	34
100	6	438	28	876	32
105	6	441	C59	930	65
148	9	460	21	940	35
172	13	470	35	1060	56.C63
210	7	483	C62	61108	70.C78
230	16	495	C70	1116	48
243	19	540	C80	1122	C92
268	13	546	30	1170	C78
270	15	558	48		

Small values of n, ($n < 400$) for which $GF(2^n)$ and $GF(2^{3n})$ have optimal normal bases and $2^{3n} - 1 = (2^n - 1)(2^{2n} + 2^n + 1)$ has only small prime factors.

Note: d is the number of decimal digits in the largest prime factor of $2^{3n}-1$.

C before a number signifies that it is the number of digits in a number that is known to be composite but for which the factorization is not known.

n	3n	d
2	6	1
4	12	2
6	18	2
12	36	3
30	90	8
35	105	6
60	180	8
70	210	7
81	243	19
90	270	15
146	438	28
180	540	C80
186	558	48
270	810	C60
292	876	32
338	1014	94
372	1116	48
378	1134	98
398	1194	C113

Examples.

$GF(2^{210})$ has an optimal normal basis.

$2^{210} - 1 = 3^2 \cdot 7^2 \cdot 11 \cdot 31 \cdot 43 \cdot 71 \cdot 127 \cdot 151 \cdot 211 \cdot 281 \cdot 331 \cdot 337 \cdot 5419 \cdot 29191 \cdot$

$86171 \cdot 106681 \cdot 122921 \cdot 152041 \cdot 664441 \cdot 1564921.$

$GF(2^{70})$ has an optimal normal basis, also.

$2^{70} - 1 = 3 \cdot 11 \cdot 31 \cdot 43 \cdot 71 \cdot 127 \cdot 281 \cdot 86171 \cdot 122921.$

Note that, as expected, $2^{70} - 1$ divides $2^{210} - 1$.

So the Pohlig - Hellman logarithm algorithm can be used on both, and the first part of the precalculation step for $GF(2^{210})$ is the precalculation step for $GF(2^{70})$.

Similarly,

$GF(2^{180})$ and $GF(2^{60})$ both have optimal normal bases.

$2^{60} - 1 = 3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321.$

$2^{180} - 1 = (2^{60} - 1) \cdot 3 \cdot 19 \cdot 37 \cdot 73 \cdot 109 \cdot 181 \cdot 631 \cdot 23311 \cdot 54001 \cdot 18837001 \cdot$

$29247661.$

Appendix 3 : Values of M(k), S(k), and T(k), for Exponentiating in a Normal Basis

In §5.5.4, we calculated the value $M(k) = 2^{k-1} + \lceil \frac{n}{k} \rceil - 2$ as the worst case complexity for the algorithm **Exponentiate**. If we have available

$$\max \left\{ \lfloor \tfrac{s}{2} \rfloor, P(k) \right\} \text{ processors,}$$

then the algorithm **Exponentiate** can be performed in $S(k) = \lceil \log_2 k \rceil + \lceil \log_2 s \rceil$ rounds of multiplications.

Much improvement may be made if we have 2^k processors, for some $2^k < \lfloor \frac{s}{2} \rfloor$. Then the algorithm takes $T(k) = \lceil \log_2 k \rceil + \lfloor \frac{s}{2^k} \rfloor + k - 1$ rounds.

For $n = 2^{10}$ we compute

k	M(k)	S(k)	processors for S(k)
5	219	11	102
6	201	11	85
7	209	11	105

So M(k) is minimized when k = 6 and the number of processors required for S(k) when k = 7.

For $n = 2^{16}$ we compute

k	M(k)	S(k)	processors for S(k)
7	9630	17	4681
8	8318	16	4096
9	7535	17	3641
10	7064	17	3277
11	6980	17	2979
12	7508	17	3969

M(k) is minimized by k = 11 and the number of processors required for S(k) by k = 11.

Values of k which minimize M(k) and T(k) for various values of n where n is a power of 2.

n	k_M	$M(k_M)$	k_S	$S(k_S)$	processors for S(k)
16	3	8	3	5	3
32	3	13	3	6	5
64	4	23	4	6	9
128	4	39	4	7	16
256	5	66	5	8	26
512	6	116	5	10	49
1024	6	201	6	11	85
2048	7	355	7	12	146
4096	8	639	8	12	256

A look at T(k), for various values of k such that the number of processors required for T(k) is less than that for $S(k_S)$, where n is a power of two.

n	k	processors needed	T(k)
16	2	4	6
	3	8	6
32	2	4	8
	3	8	7
64	2	4	12
	3	8	8
128	2	4	20
	3	8	11
256	2	4	36
	3	8	16
	4	16	11
512	2	4	68
	3	8	27
	4	16	15
	5	32	12
1024	2	4	132
	3	8	48
	4	16	23
	5	32	15
	6	64	12

n	k	processors needed	T(k)
2048	2	4	260
	3	8	91
	4	16	39
	5	32	21
	6	64	15
	7	128	13
4096	2	4	516
	3	8	176
	4	16	71
	5	32	34
	6	64	20
	7	128	15

Bibliography

[1] G. B. Agnew, R. C. Mullin and S. A. Vanstone, "Fast exponentiation in GF(2^n)", *Advances in Cryptology - Eurocrypt 88*, Lecture Notes in Computer Science 330, Springer-Verlag, Berlin, 1988, pp. 251-255.

[2] G. B. Agnew, R. C. Mullin, S. A. Vanstone, "Arithmetic Operations in GF(2^n)", *Journal of Cryptology*, to appear.

[3] David W. Ash, Ian F. Blake and Scott A. Vanstone, "On the Construction of Low Complexity Normal Bases", Research Report CORR 86-21, University of Waterloo, 1986.

[4] David W Ash, Ian F. Blake and Scott A. Vanstone, "Low Complexity Normal Bases", *Discrete Applied Mathematics*, vol. 25, 1989, pp 191-210.

[5] Lynn M. Batten, *Combinatorics of Finite Geometries*, Cambridge University Press, Cambridge, 1986.

[6] Leonard D. Baumert, *Cyclic Difference Sets*, Lecture Notes in Mathematics 182, Springer-Verlag, Berlin, 1971.

[7] Paul W. Beame, Stephen A. Cook and H. James Hoover, "Log Depth Circuits for Division and Related Problems", *Siam Journal on Computing*, vol 15, no 4, November 1986, pp. 994-1003.

[8] Elwyn R. Berlekamp, "Bit-Serial Reed-Solomon Encoders", *IEEE Transactions on Information Theory*, vol IT-28, 1982, pp. 869-874.

[9] Thomas Beth, "On The Arithmetics of Galoisfields and The Like", *3rd International Conference AAECC-3*, Lecture Notes in Computer Science 229, Springer-Verlag, Berlin, 1986, pp. 2-16.

[10] T. Beth, B.M.Cook and D Gollmann, "Architectures for exponentiation in GF(2^n)", *Proceedings CRYPTO 86*, Lecture Notes in Computer Science, 1986, pp. 302-310.

[11] T. Beth, D. Jungnickel, and H. Lenz, *Design Theory*, Bibliographisches Institut, Zurich, 1985.

[12] Dario Bini and Victor Pan, "Algorithm for Polynomial Division", *EUROCAL '85*, volume 2, Lecture Notes in Computer Science 204, Springer-Verlag, Berlin, 1985, pp. 1-3.

[13] N.L. Biggs and A.T. White, *Permutation Groups and Combinatorial Structures*, London Mathematical Society Lecture Notes Series 33, Cambridge University Press, Cambridge, 1979.

[14] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone, "Computing Logarithms in Finite Fields of Characteristic Two", SIAM Journal of Algebraic and Discrete Mathematics, vol. 5, no.2, 1984, pp. 276-285.

[15] I. F. Blake, R. C. Mullin, and S. A. Vanstone, "Computing Logarithms in GF(2^n)", *Advances in Cryptology: Proc of Crypto '84*, Lecture Notes in Computer Science 196, Springer-Verlag, Berlin, 1984, pp. 73-82.

[16] Ian F. Blake, Paul C. Oorschot and Scott A. Vanstone, "Complexity Issues for Public Key Cryptography", *Performance limits in communications theory and practice*, ed. J. K. Skwirzynski, NATO ASI Series, Kulwer Academic Publishers, Vol. 142, 1988, pp 75-97.

[17] Gilles Brassard and Paul Bratley, *Algorithmics: Theory and Practice*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[18] Ernest F. Brickell, "A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography", *Advances in Cryptography, Proceedings CRYPTO 82*, Plenum Press, New York, 1983, pp. 51-60.

[19] E. F. Brickell and J. H. Moore, "Some Remarks on the Herlestam-Johannesson Algorithm for Computing Logarithms over GF(2^p)", *Advances in Cryptology: Proceedings of Crypto '82*, Planun Press, New York, 1982, pp. 15-19.

[20] John Brillhart, D. H. Lehmer, J. L. Selfridge, Bryant Tuckerman and S. S. Wagstaff, Jr., *Factorizations of $b^n \pm 1$ b=2,3,5,6,7,10,11,12 up to high powers*, Contemporary Mathematics vol. 22, American Mathematical Society, Providence, Rhode Island, 1983.

[21] Don Coppersmith, "Fast Evaluation of Logarithms in Fields of Characteristic Two", *IEEE Transactions on Information Theory*, vol. IT-30, no. 4, 1984, pp. 587-594.

[22] P. Dembowski, *Finite Geometries*, Springer-Verlag, New York, 1968.

[23] Marijke De Soete, "Some Constructions for Authentication - Secrecy Codes", *Advances in Cryptology: Eurocrypt '86*, Lecture Notes in Computer Science 330, Springer-Verlag, Berlin, 1986, pp. 57-75.

[24] Wayne Eberly, "Very Fast Parallel Matrix and Polynomial Arithmetic", Technical report 178/85, Department of Computer Science, University of Toronto; extended abstract in *Proceedings 25th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Society, Long Beach, California, 1984, pp. 21-30.

[25] Wayne Eberly, "Very Fast Parallel Polynomial Arithmetic", *Siam Journal on Computing*, vol. 18, no. 5, October, 1989, pp. 955-976.

[26] Faith E. Fich and Martin Tompa, "The Parallel Complexity of Exponentiating Polynomials over Finite Fields", *Journal of the Association of Computing Machinery*, vol. 35, no. 3, July, 1988, pp. 651-667.

[27] Martin Fürer and Kurt Mehlhorn, "AT^2-Optimal Galois Field Multiplier for VLSI", *IEEE Transactions on Computers*, Vol 38, no.9, September 1989.

[28] Willi Geiselmann and Dieter Gollmann, "Symmetry and Duality in Normal Basis Multiplication", preprint.

[29] Philippe Godlewski and Chris Mitchell, "Key-minimal cryptosystems for unconditional secrecy.", *Hewlett-Packard Technical Manual*, memo no. HPL-ISC-TM-89-031, 1989.

[30] A. Granville, A. Moisadis and R. Rees, "Nested Steiner n-gon systems and perpendicular arrays", *Journal of Combinatorial Mathematics and Combinatorial Computing*, vol.3, 1988, pp. 163-167.

[31] H. Hanani, "On Transversal Designs", *Mathematical Centre Tracts*, vol. 55, 1974, pp 42-52.

[32] Martin E. Hellman and Justin M. Reyneri, "Fast Computation of Discrete Logarithms in GF(q)", *Advances in Cryptology: Proceedings of Crypto '82*, Planun Press, New York, 1982, pp. 3-13.

[33] Tore Herlestam and Rolf Johannesson, "On computing logarithms over $GF(2^p)$", *BIT 21*, 1981, pp. 326-334.

[34] J. W. P. Hirschfeld, *Projective Geometries over Finite Fields*, Clarendon Oxford Press, 1979.

[35] D.R. Hughes and F.C. Piper, *Design Theory*, Cambridge University Press, Cambridge, 1985.

[36] Deiter Jungnickel, Alfred J. Menzies and Scott Vanstone, "On the Number of Self-Dual Bases of $GF(q^m)$ over GF(q)", *Proceedings of the American Matematical Society*, to appear.

[37] Donald E. Knuth, *The Art of Computer Programming*, vol. 2, Addison-Wesley, Reading, Massachusetts, 1981.

[38] E. S. Kremer, D. L. Kreher, R. Rees and D. R. Stinson, On perpendicular arrays with $t \geq 3$, *ARS Combinatoria*, to appear.

[39] B. A. Laws, Jr. and C. K. Rushforth, "A Cellular-Array Multiplier for GF(2^m)", *IEEE Transactions on Computing*, vol. C-20, 1971, pp. 1573-1578.

[40] H.W. Lenstra and R. J. Schoof, "Primitive normal bases for finite fields", *Mathematics of Computing*, vol 48., 1978, pp 217-231.

[41] Rudolf Lidl and Harold Niederreiter, *Finite Fields*, Addison-Wesley, Reading, Massachusetts, 1983.

[42] C. C. Lindner and D. R. Stinson, "Steiner pentagon systems", *Discrete Mathematics*, vol. 52, 1984, pp. 67-74.

[43] Bruce E. Litlow and George I. Davida, "O(log(n)) Parallel Time Finite Field Inversion", *Proceedings of the Third Aegean Workshop on Computing, Corfu,* Lecture Notes in Computer Science 319, Springer, 1988, pp.74-80.

[44] James L. Massey, "Cryptology - A Selective Survey", in *Digital Communications*, E. Biglieri and G. Prati (eds), Elsevier Science Publishers, North Holland, 1986.

[45] Edoardo D. Mastrovito, "Exponentiation over Finite Fields GF(2^m). A Comparison of Polynomial and Normal Basis Designs with Respect to VLSI Implementation.", *Fourth Joint Swedish-Soviet International Workshop on Information Theory*, Gotland Sweden, August 27 – September 1, 1989

[46] Alfred Menezes, *Representations in Finite Fields*, Master's thesis, University of Waterloo, 1989.

[47] Robert J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Kluwer, Boston

[48] Masakatu Morii and Masao Kasahara, "Efficient Bit-Serial Multiplier", submitted to *IEEE Transactions of Information Theory.*

[49] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone and R. M. Wilson, "Optimal Normal Bases in GF(p^n)", *Discrete Applied Mathematics*, vol. 22, 1988-89, pp. 149-161.

[50] R. C. Mullin, P. J. Schellenberg, G. J. H. van Rees and S. A. Vanstone, "On the Construction of Perpendicular Arrays", *Utilitas Mathematica*, vol. 18, 1980, pp. 141-160.

[51] Michael J. Norris and Gustavus J. Simmons, "Algorithms for High-Speed Modular Arithmetic", *Congresses Numerantium*, vol. 31, 1981, pp. 153-163.

[52] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance", *Advances in Cryptology*, Lecture Notes in Computer Science 209, Springer-Verlag, Berlin, 1984, pp. 224-260.

[53] Din Y. Pei, Charles C. Wang, and Jim K. Omura, "Normal Basis of Finite Field GF(2^m)", *IEEE Transactions on Information Theory*, vol. IT-32, no. 2, 1986, pp. 285-287.

[54] Stephen C. Pohlig and Martin E. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance", *IEEE Transactions on Information Theory*, vol. IT-24, no. 1, 1987, pp. 106-110.

[55] J.M. Pollard. "The Fast Fourier Transform in a Finite Field", *Mathematics of Computation*, vol. 25, no. 114, 1971.

[56] John H. Reif, "Logarithmic Depth Circuits for Algebraic Functions", *SIAM Journal on Computing*, vol. 15, no. 1, February 1986, pp. 231-242.

[57] C. E. Shannon, "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, vol. 28, 1949, pp. 656-715.

[58] Gustavus J. Simmons, "A Natural Taxonomy for Digital Information Authentication Schemes", in *Advances in Cryptology*, Lecture Notes in Computer Science, vol. 293, Springer-Verlag, Berlin, 1988, pp. 269-288.

[59] Gustavus J. Simmons, "A Survey of Information Authentication", *Proceedings of the IEEE 76*, 1988, pp. 603-620.

[60] D. R. Stinson, "The combinatorics of authentication and secrecy codes", *Journal of Cryptography*, submitted.

[61] D. R. Stinson, " A Construction for Authentication/Secrecy Codes From Certain Combinatorial Designs", *Journal of Cryptology*, vol. 1, 1988, pp. 119-127.

[62] D. R. Stinson, " Some observations on parallel algorithms for fast exponentiation in GF(2^n)", *SIAM Journal on Computing*, to appear.

[63] D .R. Stinson, "Some Constructions and Bounds for Authentication Codes", *Journal of Cryptology*, vol.1, 1988, pp. 37-51.

[64] D. R. Stinson and L. Teirlinck, "A construction for authentication / secrecy codes from 3-homogeneous permutation groups", *European Journal of Combinatorics*, to appear.

[65] Anne P. Street and Deborah J. Street, *Combinatorics of Experimental Design*, Clarendon Press, Oxford, 1987.

[66] J. J. Thomas, J. M. Keller, and G.N. Larsen, "The calculation of Multiplicative Inverses Over GF(P) Efficiently Where P is a Mersenne Prime", *IEEE Transactions on Computers*, vol. C-35, no. 5, 1986, pp. 478-482.

[67] Henk C. A. van Tilborg, *An Introduction to Cryptology*, Kluwer Academic Press, Boston.

[68] Joachim von zur Gathen, "Computing Powers in Parallel", *SIAM Journal on Computing*, vol. 16, no. 5, October, 1987, pp. 930-945.

[69] Joachim von zur Gathen, "Inversion in finite fields using logarithmic depth", *Journal of Symbolic Computing*, to appear.

[70] Joachim von zur Gathen, "Parallel Arithmetic Computations: A Survey", *Proceedings 13th International Symposium on the Mathematical Foundations of Computer Science, Bratislava*, Lecture Notes in Computer Science 223, 1986, pp. 432-452.

[71] Joachim von zur Gathen, "Parallel Algorithms for Algebraic Problems", *SIAM Journal on Computing*, vol. 13, no. 4, November 1984.

[72] Muzhong Wang and Ian F. Blake, "Bit Serial Multiplication in Finite Fields", preprint.

[73] Charles C. Wang, T. K. Truong, Howard M. Shao, Leslie J. Deutsch, Jim K. Omura, and Irving S. Reed, "VLSI Architectures for Computing Multiplications and Inverses in GF(2^m)", *IEEE Transactions on Computers*, vol.C-34, no. 8, 1985, pp. 709-717.

[74] Stanislaw H. Zak and Kai Hwang, "Polynomial Division on Systolic Arrays", *IEEE Transactions on Computers*, vol. C-34, no. 6, 1985, pp. 577-578.

[75] Neal Zierler, "A Conversion Algorithm for Logarithms on GF(2^n)", *Journal of Pure and Applied Algebra*, vol. 4, 1974, pp. 353-356.