1986

An Implementation of the Chor-Rivest Knapsack Type Public Key Cryptosystem

Robert Jr T. Salamone

Rochester Institute of Technology
School of Computer Science and Technology

**An Implementation of the
Chor-Rivest Knapsack Type
Public Key Cryptosystem**

by

Richard Thomas Salamone, Jr.

December 16, 1986

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

Approved By:

2/6/1986

Dr. Donald L. Kreher (Advisor)

Dr. Stanislaw P. Radziszowski

Dr. John L. Ellis

Title of Thesis: An Implementation of the Chor-Rivest Knapsack Type Public Key Cryptosystem

I Richard Thomas Salamone, Jr. hereby grant permission to the Wallace Memorial Library, of R.I.T., to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Abstract

The Chor-Rivest cryptosystem is a public key cryptosystem first proposed by MIT cryptographers Ben Zion Chor and Ronald Rivest [Chor84]. More recently Chor has implemented the cryptosystem as part of his doctoral thesis [Chor85]. Derived from the knapsack problem, this cryptosystem differs from earlier knapsack public key systems in that computations to create the knapsack are done over finite algebraic fields. An interesting result of Bose and Chowla supplies a method of constructing higher densities than previously attainable [Bose62]. Not only does an increased information rate arise, but the new system so far is immune to the low density attacks levied against its predecessors, notably those of Lagarias-Odlyzko and Radziszowski-Kreher [Laga85, Radz86].

An implementation of this cryptosystem is really an instance of the general scheme, distinguished by fixing a pair of parameters, p and h, at the outset. These parameters then remain constant throughout the life of the implementation (which supports a community of users). Chor has implemented one such instance of his cryptosystem, where p =197 and h =24. This thesis aspires to extend Chor's work by admitting p and h as variable inputs at run time. In so doing, a cryptanalyst is afforded the means to mimic the action of arbitrary implementations.

A high degree of success has been achieved with respect to this goal. There are only a few restrictions on the choice of parameters that may be selected. Unfortunately this generality incurs a high cost in efficiency; up to thirty hours of (VAX11-780) processor time are needed to generate a single key pair in the desired range (p = 243 and h = 18).

Table of Contents

CHAPTER 1

Introduction and Background

1. Overview

A *Public Key Cryptosystem* is implemented in software to run on the UNIX† operating system. Fundamentally a cryptosystem should provide users a means to exchange secret information such that eavesdroppers cannot understand the communication. A public key cryptosystem provides this service with an added bonus: The correspondents do not exchange any secret information, or keys, before establishing (encrypted) communication. Rather, the sender consults a directory of all participants in the system and obtains the information (the *public key*) associated with the desired recipient. This information is passed to a system provided facility (the *enciphering facility*) along with the message to be disguised. The resultant message is unintelligible, except to the intended recipient. With other information (his *private key*) and the garbled message, the receiver may call a second system provided facility (the *deciphering facility*) to recover the original message. While it is not necessary for the users to have had previous contact, it prerequisite that the keys be precomputed for each participant (in the *key generation facility*).

The particular cryptosystem implemented was proposed by Ben-Zion Chor and Ronald Rivest in 1984 [Chor84]. In this scheme, encryption and decryption are based on the knapsack problem from complexity theory (see § 1.2.3). Thus we shall refer to this scheme as the *Chor-Rivest Knapsack Type Public Key Cryptosystem*, or the *Chor-*

†UNIX is a trademark of Bell Laboratories.

Rivest system for short.

As a public key cryptosystem our Chor-Rivest system implementation naturally provides the above facilities. However, our goal was not to develop a secure public key system. Instead we provide a tool for cryptanalysts who wish to break the system. More specifically, there has been research at Rochester Institute of Technology which may be applied to breaking knapsack based cryptosystems [Radz86]. The Chor-Rivest system has not yet been broken but there is some confidence that this research may be extended to include this scheme. This implementation will provide the data necessary for this task.

Because we provide a system for experimentation with different cryptanalytic attacks, flexibility is required of the system. In particular, the Chor-Rivest scheme is based on arithmetic in finite fields. Typically, a given implementation would fix certain parameters defining the specific field used by all of the system facilities. This greatly simplifies the implementation without compromising the integrity of the system. Here, these same parameters are variables, which the cryptanalyst may alter to try his attack on arbitrary Chor-Rivest implementations. Furthermore, the system facilities allow easy disclosure or modification of important intermediate results. Alternatively, maintenance of the directory of users and protocols for message passing are not a concern of this project. Hence, although we provide all functions as prescribed this is not a ready to install user oriented system [Chor84].

2. Previous Work

2.1. General Background

In this thesis we are primarily interested in public key cryptology. However an overview of the techniques and terminology of classical (or conventional) cryptology is prerequisite to the ensuing discussion. The field of *cryptology* embraces the disciplines

of *cryptography* and *cryptanalysis*. Cryptography is the designing of cipher systems, while cryptanalysis is concerned with breaking ciphers. Intuitively, a *cipher* is a means of disguising a message such that (hopefully) only the intended recipient can recover its meaning.

Cryptology has been practiced throughout much of history. Julius Caesar (100-44 B.C.) and Thomas Jefferson (1743-1826) are notable examples of prominent cryptographers [Beke82]. Thus cryptology is a very old science, but it was not until the 1940's that it was formalized.

Shannon defines a *cipher system* (also called a cryptosystem) as a three-tuple: These are the set of messages, the set of cryptograms (or ciphertext messages), and a set of invertible transformations [Shan49]. The sender disguises a message, called the *plaintext*, by applying one of the transformations to the message to yield the *ciphertext*. The recipient of the message performs the inverse of the transformation to recover the original message. These operations are called encryption and decryption respectively. A *key* is associated with each element of the set of transformations, thus the sender and receiver must agree upon a key prior to exchanging messages.

The cryptanalyst attempts to break the system. That is, given only publicly available information and a portion of ciphertext, the cryptanalyst hopes to find a way to deduce the plaintext or better yet, the key used. To attain this goal, the cryptanalyst will often begin by trying to break the system given much or all of the secret information. He then proceeds by polishing his technique to require progressively less privileged information, ultimately rendering the system useless. Using this tact a partially successful attack often results. Perhaps an attack that works only for a subset of the message space or when certain additional secret information is compromised.

It is always assumed that the cryptanalyst knows the entire cipher system, including the set of transformations, but not the particular key used. We may then categorize

cryptanalysis into three levels of attack upon the system [Diff76]. The weakest attack occurs when the interceptor has only encrypted messages to work with. This is a *ciphertext only* attack. A stronger attack, the *known plaintext* attack, occurs when the interceptor has at least one encrypted message along with its plaintext translation. Finally, the *chosen plaintext* attack is the strongest attack to which a cipher system may be subjected. Here, the cryptanalyst has the means to encrypt messages of his own choosing.

Now we can assess the security of a given cryptosystem. This is a measure of its resilience to the various levels of cryptanalytic attacks. A cipher system is called *unconditionally secure* if it can withstand cryptanalysis with unlimited computation. A system called the one-time pad is the only unconditionally secure cipher system in common use. Unconditional security results only if several meaningful solutions exist for a given cipher [Diff76]. In general it is necessary to use inordinately long keys (as in the one-time pad) or extremely complicated algorithms to attain unconditional security. These approaches are undesirable due to the increased cost or time requirements. Therefore, for practical (economic) reasons we are interested rather in systems that are *computationally secure*. Here the system is considered secure if the computational cost of cryptanalysis is prohibitive, even though the system might not resist an unbounded attack. Secure, throughout the remainder of this paper, will mean "computationally secure". We may further classify the security of a system relative to the levels of attack to which it will succumb. So while a system may be secure under a ciphertext only attack, it may be considerably less secure under a chosen plaintext attack.

There are other factors of import in the evaluation of cipher systems in addition to security. Perhaps the most important of these is the information rate R provided by a particular system [Chor84, Simm79]:

$$R = \log_2 |M| / N$$

where M is the message space and N is the number of bits in the ciphertext. Thus the

information rate is an efficiency concern: If a system is secure, but encrypted messages are many times longer than the original message, one may want to choose a slightly less secure system having a higher information rate. For instance, the one-time pad mentioned above, requires a key as long as all of the messages to ever be encrypted. Clearly this is impractical in most applications!

2.2. Public Key Cryptosystems

In 1976 Diffie and Hellman sparked a revolution in cryptology with their idea for *public key* encryption [Diff76]. The outstanding feature of this class of cipher systems is that the communicants do not exchange keys prior to exchanging messages. Instead, a key is associated with each user in a public directory. In this section we outline how this is accomplished.

Recall that in conventional cipher systems, the sender and receiver must agree upon then hold in secret the key. This is necessary because the encryption and decryption functions are closely related to each other. Such a scheme is termed *symmetric* if it is easy to deduce the enciphering transformation knowing the deciphering transformation and vice versa [Simm79].

If on the other hand it is *difficult*[1] to deduce the decryption function knowing the encryption function an *asymmetric* scheme results. In asymmetric schemes there are two keys, one associated with encryption and the other associated with decryption. Since it is not feasible to compute one key from the other, there is no loss in security if one of the keys is revealed. Thus a public key system is an asymmetric scheme in which an encryption/decryption key pair is found for each participant's enciphering (public) key is posted in a public directory along with his name, but his deciphering (private) key is

[1]By difficult we mean computationally infeasible, given modern equipment and technology.

held in secret. To send messages one simply obtains the recipient's public key then encrypts the message using it. The recipient deciphers the message using his private key.

To construct a public key system one needs a way to find matched key pairs satisfying the above constraint. This is done by choosing a computationally difficult problem from complexity theory as the basis of the system. We will digress briefly to introduce some terms from this field which will be useful to us.

Complexity Theory is a collection of results that attempts to explain what we mean when we say "problem A is harder than problem B".[2] A *problem* is a general question which may be described by specificing its input parameters and stating the properties required of the solution. If we are given all of the input parameters then we have an instance of the general problem. An *algorithm* is a detailed procedure for solving a problem; For our purposes, algorithms are equivalent to computer programs. The approach taken is to determine the complexity of individual problems, then problems with similar complexity are placed into one of the several equivalence classes constituting the complexity hierarchy.

We now describe the criteria for membership in the predominant equivalence classes. Say f is some function and α is an algorithm that computes f (i.e. $\alpha(x)$ computes $f(x)$ on input x). We count the number of "primitive instructions" required by α to compute $f(x)$. This value is called the *running time* $RT(\alpha, x)$. Now the complexity $T_\alpha(n)$ of α on input of size n is:

$$T_\alpha(n) = max\{RT(\alpha, x) : size\ of\ x\ is\ n\}.$$

If there is a polynomial $p(n)$ such that $T_\alpha(n) \leq p(n)$ for all n, we say that α computes f in *polynomial time* and write:

---

[2]No attempt is made to be rigorous in this discourse; only an intuitive understanding is necessary. The following results largely follow Garey and Johnson [Gare79].

$$T_\alpha(n) = O(p(n)).$$

The set P = {all deterministic polynomial time problems} is the resulting equivalence class of problems. If on the other hand $T_\alpha(n) = O(2^{|p(n)|})$ for some polynomial $p(n)$ then α is called an *exponential time* algorithm and is a member of EXP = {all exponential time problems}. EXP encompasses more difficult problems than P. Also included in EXP is the class NP = {all *nondeterministic polynomial time* problems}. More precisely, $P \subseteq NP \subseteq Exp$, and $P \subset Exp$.[3]

Finally, the set of NP problems has a subset called the NP-complete problems which are in some sense the hardest problems in NP. If it is possible to adapt an algorithm that solves problem B to obtain one that solves problem A then there is a function $\tau: A \rightarrow B$. Furthermore, if τ is computable in polynomial time then τ is said to be a *polynomial-time transformation from A to B.* A problem A is called NP-complete if and only if

1) $A \in NP$,
2) for every $B \in NP$, there is a polynomial-time transformation from B to A.

It is members of this class of problems that have been suggested for use as the basis of public key cryptosystems[4] [Diff76].

Notice that the classification of problems presented above describes the worst case complexity of a given problem. Consider a problem X which is a member of NP. In general, with sufficiently large input we could not expect to solve X on any known computing machine. But since we have a nondeterministic algorithm to solve X, there may be several instances of X which can be solved on a computer, even for very large input. This observation is crucial to our application, providing a means to construct matched

[3]There are also an infinite number of complexity levels above EXP in the complexity hierarchy: However, we are able to exhibit only somewhat artificial problems as witnesses to this hierarchy [Lewi81].

[4]Wagner has also proposed the use of undecidable problems from the theory of computability, but this idea has not yet been generally accepted [Wagn84].

public/private key pairs.

The following terms were defined by Diffie and Hellman to augment complexity theory for applications to public key cryptography [Diff76]. We say that a function f is *easy* to compute if there is a known algorithm α in the complexity class P which computes f. Otherwise, f is *hard* to compute. If a bijection b is easy to compute and b^{-1} is hard to compute, then b is said to be a *one way function.* Finally, a function t is a *trapdoor function* if

(1) t is easy to compute;
(2) there exists some side information (the key) without which t is a one way function, but given the key t^{-1} is easy to compute.

To generate key pairs then we select a problem in NP (preferably in NP-complete) believed to be not in P as the basis of the system. An instance of this problem is construed that may be deterministicly solved with information about the design. This version of the problem becomes the private key. This instance of the problem is then disguised to appear to be the general problem, hence the public key. In other words a trapdoor function is constructed based on the chosen problem. Thus the technique used to embed a trapdoor in a given problem defines a public key scheme.

To break the system, the cryptanalyst has two options. One option is to try to convert the public key back to the private key. To resist this attack, the cryptographer must insure that the only way to do this is by exhaustive search over a prohibitively large set of transformations. His only other option is to directly solve the general NP-complete problem given by the public key. Notice that by their very nature, public key systems will be subject to chosen plaintext attack.

Shortly after Diffie and Hellman introduced the public key idea two schemes were proposed. One was the Merkle-Hellman system based on the knapsack problem [Merk78]. The other, set forth by Rivest, Shamir and Adleman (called the RSA method)

is based on the difficulty of factoring very large numbers [Rive78]. The RSA method is generally believed to be the best public key technique in existence, and has the honor of being the first one implemented in an LSI chip [Bric83]. The history of knapsack based schemes is much more complicated as several early models have been broken leading to subsequent improvements. We shall examine these developments in the next section, culminating with the Chor-Rivest knapsack scheme.

2.3. The Knapsack Problem in Cryptography

The *knapsack problem* is a very well known *NP*-complete problem from complexity theory. Given a knapsack with fixed capacity and several items of various weights, the knapsack problem is to determine how to completely fill the knapsack (if possible). For our purposes, we are actually concerned with the closely related *subset sum* problem [Gare79, Radz86]:

The Subset Sum Problem
Given a vector of positive integers $\vec{a} = (a_1 a_2, \cdots, a_n)$ and a positive integer M find, if it exists, a (0,1)-vector $\vec{x} = (x_1, x_2, \cdots, x_n)$ such that

$$\sum_{i=1}^{n} a_i \cdot x_i = M.$$

To adapt this problem to cryptology we pick an $\vec{a}$ such that the subset sum problem can be solved easily for different M's, then we scramble the system giving $\vec{b} = (b_1, b_2, \cdots, b_n)$. An easy way to do this is by performing a modular multiplication of $\vec{a}$ by some integer, v, which is relatively prime with respect to the modulus, m:

$$\vec{b} = \vec{a} \cdot v \ (mod\ m).$$

The resultant random-looking vector, $\vec{b}$, is published as the public key while $\vec{a}$, m and v comprise the private key [Merk78]. This process is called the *key generation phase*. To encrypt a message, P, we merely take its binary representation to be a vector $\vec{p} = (p_1, p_2, \cdots, p_n)$, then perform the computation:

$$C = \sum_{i=1}^{n} p_i \cdot b_i,$$

where C is the ciphertext to be sent. Note that if the message length $|\vec{p}|$ exceeds the length of $\vec{b}$, we merely break the message into blocks of length n.

Decryption is only slightly more complicated. Because v and m are relatively prime, there exists v^{-1} modulo m such that $vv^{-1} \equiv 1 \ (mod\ m)$. Thus we can find $N = C \cdot v^{-1}$, then solve the easy knapsack:

$$N = \sum_{i=1}^{n} p_i \cdot a_i,$$

to recover the binary message $\vec{p}$.

The collection of the key generation, encryption, and decryption algorithms together define the system. In the description above, we have avoided the topic of how to create $\vec{a}$. The method chosen identifies the type of knapsack system we are using. For instance, the Merkle-Hellman system requires that $\vec{a}$ be a superincreasing sequence, that is

$$a_i > \sum_{j=1}^{i-1} a_j \qquad \text{for } i = 2, 3, \cdots, n.$$

This was the first and simplest knapsack system proposed. A solution is found by successive subtractions.

An important measure of a knapsack system is its *density*:

Knapsack Density
The density $d(\vec{a})$ of a knapsack $\vec{a} = (a_1, a_2, \cdots, a_n)$ is $d(\vec{a}) = n / \log_2(max\ a_i)$.

This property represents the information rate of the system [Chor84, Laga85, Radz86]. For instance, consider the knapsack vectors $\vec{r} = (19, 14, 31, 15, 26)$ and $\vec{s} = (4, 9, 16, 32, 64)$. Both $\vec{r}$ and $\vec{s}$ have an average weight of 25, but each of the weights in $\vec{r}$ can be represented in six bits whereas seven bits are required to represent the elements of $\vec{s}$. This relationship is reflected in the densities, $d(\vec{r}) = \frac{1}{6} > d(\vec{s}) = \frac{1}{7}$. There is

also a relationship between the density of a knapsack and its security that will become apparent in later sections. In this context we will speak in terms of *high* and *low* density knapsacks where the cutoff is roughly 1.0. For now however, notice that superincreasing knapsacks like $\vec{s}$ are generally low density since a_n must be large relative to n.

2.4. Recent Developments in Knapsack Cryptology

Having described the important properties of public key cryptosystems in general and knapsack systems in specific, we now review their historical development. Essentially, each time a knapsack system has been proposed, its cryptanalysis has prompted advances in solving the subset sum problem, which in turn has caused cryptographic improvements in knapsack design! The current state of affairs finds the ball in the court of the cryptanalysts due to the appearance of the Chor-Rivest system.

The Merkle-Hellman knapsack (described above) was developed at Stanford in 1978 as a direct result of Merkle's doctoral work [Merk78, Merk82]. Since this introduction, the security of trapdoor knapsacks have been regarded with suspicion by the cryptographic community [Bric83]. Indeed, in 1978 Adi Shamir (one of the inventors of the RSA scheme) and Zipple showed that if the modulus m were known the system could certainly be broken [Bric83]. Thus although our example showed only one modular multiplication in the key generation phase, it was realized that the system could be strengthened by iterating this process several times with a different v and m at each iteration. In 1982 Shamir found a polynomial time algorithm to completely brake the single-iteration method [Sham82b]. Brickell soon discovered a technique to break the multiple-iteration technique [Bric83].

Shamir, working at the Weizmann Institute in Israel, used the knowledge that the trapdoor was a superincreasing sequence in his attack. It turns out that the superincreasing nature of the trapdoor imposes too much structure on the system. The wor-

khorse in this attack is the Lovasz algorithm in which basis reduction is performed on a lattice [Lens83]. The goal of this operation is to find the basis with the shortest basis vectors. For low-density knapsacks, the shortest vector found by the algorithm contains the solution to the subset sum problem.

Coincidently with the cryptanalysis of the Merkle-Hellman scheme, Shamir published an improved knapsack system [Sham82a]. In the new scheme the trapdoor knapsack need not be superincreasing, so the weights can be closer together, yielding a higher density. Without going into details, the keys are formed via modular multiplication of easy (but not necessarily superincreasing) knapsacks. The Merkle-Hellman knapsacks form a subset of these newer knapsacks, thus an attack on this improved system would apply to its predecessors.

Such attacks appeared very shortly as expected. Unfortunately, the Lovasz algorithm is not guaranteed to find the shortest vector in the lattice. This does not appear to be a problem in the attack on the Merkle-Hellman system due to its very low density. Adleman (again of RSA fame) noted that if the basis reduction algorithm came closer to finding the shortest vector in the lattice it could be used to break the new Shamir system [Adle83]. Also in 1982, the so called L^3 Algorithm, an improvement on the Lovasz method above, was introduced [Lens82]. This algorithm improves upon its predecessor in that it is slightly faster and many low density (density < 0.6) knapsacks could be solved with it, including the Shamir scheme.

At this point one might assume that knapsack systems were doomed. This was not the case for several reasons. As of 1982, the L^3 algorithm had not been applied to the subset sum problem; This happened only recently when Lagarias and Odlyzko used it in an algorithm to solve all sufficiently low density problems [Laga85]. Another, perhaps more important, reason for not dismissing knapsack based cryptosystems has to do with our definition of security. Recall that for a public key system to work it must be

infeasible to break the system. The aforementioned attacks have polynomial time complexity, thus a sufficiently large knapsack (i.e. several hundred weights) is not feasibly broken [Bric83]. The final and most important reason that knapsacks have not been dismissed is (you guessed it!), a system has been devised to deliberately foil these low density attacks with high density knapsacks. This is the Chor-Rivest scheme [Chor84]. Before proceeding to this latest knapsack system we must explore a very recent development in solving the subset sum problem.

Radziszowski and Kreher, based at Rochester Institute of Technology, proposed an algorithm superior to L^3 [Radz86]. Although the core of their algorithm is again the lattice basis reduction technique of L^3, several enhancements are added. Because they avoid many of the multiprecision operations of the former method, their algorithm runs an order of magnitude faster. Another major improvement in this algorithm is a technique the authors call weight reduction which causes shorter vectors to be found in the basis reduction operation. Due to these improvements, this algorithm can solve knapsacks of density less than 1.0 while L^3 works reliably for knapsacks with density less than 0.6. It is noteworthy that neither algorithm makes any assumption about how the knapsack was constructed.

At the present time, the Chor-Rivest scheme appears to be secure against this algorithm, producing knapsacks with density exceeding 1.0. However, Radziszowski and Kreher report that further gains are to be had from their algorithm, hopefully to the point of successful attack on the Chor-Rivest scheme [Radz86a]. This thesis is a sophisticated implementation of the Chor-Rivest scheme expressly designed to aid these cryptanalysts. Prior to the current project, the only implementation of Chor-Rivest, was a limited version produced by Chor himself as part of his doctoral thesis [Chor85]. His implementation produces knapsacks of 197 elements, with a message space of binary vectors having weight 24. The general scheme is not limited to these parameters, there

are many suitable possibilities. The current implementation works for nearly all of the suggested parameters, not only 197 and 24. The remainder of this chapter provides the mathematical background needed to understand this most recent knapsack cryptosystem.

3. Theoretical and Conceptual Development

A result from Bose and Chowla is the cornerstone of the Chor-Rivest cryptosystem, providing the means to construct dense knapsacks [Bose62]. However this construction takes place in finite algebraic structures called *Galois fields*. In this section we first orient the reader to Galois fields, then we examine how to represent and manipulate them. Finally, the Bose-Chowla theorem is proved. The cryptographic usefulness of this theorem will become apparent in chapter 2 where we show in detail how Chor and Rivest exploit it. We provide a running example of these concepts as they apply to the current problem (summarized in appendices A and B for easy reference).

3.1. Galois Fields

This treatment of Galois Fields is neither rigorous nor exhaustive, we begin by stating several useful results from Gilbert [Gilb76]. The interested reader is referred to the source for proof of the theorems.

3.1.1. Definitions and Theorems

Definition 1: **Commutative Ring**

> A *commutative ring* $(R, +, \cdot)$ is a set R, together with two binary operations $+$ and $\cdot$ on R satisfying the following axioms. For any $a, b, c \in R$,
>
> (i) associativity of addition - $(a+b)+c = a+(b+c)$;
> (ii) commutativity of addition - $a+b = b+a$;
> (iii) existence of additive identity - there exists $0 \in R$ called the *zero* such that $a+0 = a$;
> (iv) existence of additive inverse - there exists $-a \in R$ such that $a+(-a) = 0$;
> (v) associativity of multiplication - $(a \cdot b) \cdot c = a \cdot (b \cdot c)$;
> (vi) commutativity of multiplication - $a \cdot b = b \cdot a$;
> (vii) existence of multiplicative identity - there exists $1 \in R$ such that

$a \cdot 1 = a$;
(viii) mutual distributivity of multiplication and addition - $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$.

Definition 2: **Field**

A *field* is a commutative ring having more than one element and satisfying the following additional axiom,

(ix) existence of multiplicative inverse - for each $a \in R$ there exists $a^{-1} \in R$ such that $a \cdot a^{-1} = 1$.

Definition 3: **Polynomial Ring**

The set of all polynomials in x with coefficients from the commutative ring R is denoted by $R[x]$. That is,

$$R[x] = \{a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n : a_i \epsilon R, n \geq 0\}.$$

This forms a commutative ring $(R[x], +, \cdot)$ called the *polynomial ring with coefficients from R* where addition and multiplication of the polynomials

$$p(x) = \sum_{i=0}^{n} a_i x^i \text{ and } q(x) = \sum_{i=0}^{m} b_i x^i$$

are defined by

$$p(x) + q(x) = \sum_{i=0}^{max\,(m,n)} (a_i + b_i) x^i$$

and

$$p(x) \cdot q(x) = \sum_{k=0}^{m+n} c_k x^k \text{ where } c_k = \sum_{i+j=k} a_i b_j.$$

Definition 4: **Reducible Polynomial**

A polynomial $f(x)$ is said to be *reducible* over the ring (or field) F if it can be factored into two (non-constant) polynomials in $F[x]$, otherwise $f(x)$ is called *irreducible*.

Theorem 5

The ring $F[x]/(p(x))$ is a field if and only if $p(x)$ is irreducible over the field F, where the notation $F[x]/(p(x))$ denotes the set of residues $\{f(x) \bmod p(x) : f(x) \in F[x]\}$.

Theorem 6

If F is a finite field, it has p^m elements for some prime p and some integer m. The prime p is called the *characteristic* of F and is the smallest nonzero integer such that $p \cdot 1 = 0$. Furthermore, all fields having p^m elements are isomorphic to each other.

Definition 7: **Galois Field**

A finite field with p^m elements is called a *Galois field* of order p^m and is denoted by $GF(p^m)$; they are unique up to isomorphism.

3.1.2. Representation of a Galois Field

The Galois field $GF(p^n)$ may be represented as the set of polynomials over $\mathbb{Z}_p$ modulo an irreducible polynomial $f(x)$ of degree n. That is

$$GF(p^n) = \mathbb{Z}_p[x]/(f(x))$$

Thus we can represent the elements of the field by choosing the polynomials of degree $\leq n-1$ with coefficients from $\mathbb{Z}_p$ as equivalence class representatives. In the following we will use "polynomial" to mean "polynomial modulo some irreducible polynomial". For instance our representation of the field $GF(5^3)$ will include the polynomials

$$x^2 + 4x + 2, \tag{1}$$

or

$$2x, \tag{2}$$

but not:

$$2x^3 + 8x^2 + 4x. \tag{3}$$

A degree m polynomial is called *monic* (of degree m) if the coefficient of the x^m term is 1, so (1) above is monic of degree 2.

The operations of "+" and "·" are similar to polynomial addition and multiplication learned in elementary algebra: Addition consists of adding the coefficients of corresponding terms and multiplication is accomplished by multiplying each term of the first polynomial by each term of the second, then combining like terms. For example, if we multiply (1) and (2) above, (3) results. Note however that these operations are performed modulo some $f(x)$. So, for instance if we take (3) modulo $f(t) = x^3 + x + 1$ then the resulting polynomial, $3x^2 + 2x + 4$, is a valid equivalence class representative. This amounts to dividing the result of the elementary operations above by $f(x)$ and taking the remainder as the final result, a member of $GF(p^n)$. If we can find such an $f(x)$, we

will have defined a field satisfying all of the necessary properties. Thus we write $GF(p^n) = \mathbb{Z}_p[x]/(f(x))$.

3.1.3. Primitive Elements

A (multiplicative) *generator* of a Galois field, $\alpha \in GF(p^n)$, has the following special property: α^i takes on the value of each nonzero field element exactly once as i ranges through the natural numbers $1 \cdots p^n-1$. Thus we denote the elements of $GF(p^n)$ as $\{0, \alpha, \alpha^2, \cdots, \alpha^{p^n-1}\}$. There may be more than one generator for a field, these elements are collectively called *primitive elements* of the field[5]. This representation allows us to easily multiply in the field. For instance, in $GF(c)$,

$$\alpha^a \cdot \alpha^b = \alpha^{a+b \pmod{c-1}}.$$

The following theorem is the basis of our algorithm to construct fields.

Theorem 8

> Given any n-degree monic polynomial f over $GF(p)$ and $\alpha \in GF(p^n)$, then α is primitive, and f is irreducible over $GF(p)$ if and only if
>
> (1) $\alpha^k \pmod f \neq 1$ for all $k < p^h-1$;
> (2) $\alpha^{p^h}-1 \pmod f = 1$.

These criteria are to be used by this implementation to decide whether a randomly selected pair f and α define a field.

3.1.4. Discrete Logarithms

Say we have selected a primitive element $\alpha \in GF(p^n) = \mathbb{Z}_p[x]/(f(x))$. Then clearly as m ranges through the integers $1, 2, \cdots, p^n-1$, $\alpha^m \pmod f$ takes on the value of each nonzero field element exactly once. Conversely, to each nonzero field element A we associate the integer m, and say that m is the *logarithm to the base* α of A. So the

[5] In fact there are always $\phi(p^n-1)$ primitive elements in $GF(p^n)$, where ϕ is the Euler totient.

notation

$$m = \log_\alpha(A)$$

is used to express the inverse of $A = \alpha^m \pmod{f(x)}$, the exponentiation function. We consider the logarithm to be an integral value in the range $1, 2, \cdots, p^n-1$, and use $p^n-1 = \log_\alpha(1)$.

The calculation of m is called the *discrete logarithm problem.* When the order of $GF(p^n)$ is small one may tabulate the field elements with their associated logarithms, then perform a search. If however the field is large enough, this will not be possible. In this case the discrete logarithm problem is relatively difficult: No polynomial time algorithm is known for arbitrary fields. On the other hand it is relatively easy (polynomial time) to exponentiate. Despite this, there are practical algorithms for fields having special properties. Odlyzko has recently cataloged and critiqued many of these techniques [Odly86a, Odl86b]. One is the Coppersmith algorithm for fields $GF(2^n)$ (where $n <\sim 400$) [Copp84]. Another is the Pohlig-Hellman algorithm which is applicable to $GF(q)$ when $q-1$ has small prime factors only [Pohl78]. We chose the latter algorithm for the current project since it is more general in scope.

3.1.5. Construction of a Galois Field

To illustrate the above concepts we give a simple example. We wish to construct $GF(4)$, using our polynomial representation. Since $4 = 2^2$, we can construct such a field if we can find an irreducible polynomial of degree 2 over $\mathbb{Z}_2$.

By trial and error, we find the polynomial $p(x) = x^2 + x + 1$ is irreducible because $p(0) = 1$ and $p(1) = 3 \equiv 1 \pmod 2$, which implies that p has no factors in $\mathbb{Z}_2$.

Next we try to find a generator of the field, α. In this case the polynomials x and $x + 1$ are both primitive elements. If we select $\alpha = x$ as our generator the following representation of $GF(4)$ results:

$$0 = 0x + 0$$

$$\alpha = x$$

$$\alpha^2 = x + 1$$

$$\alpha^3 = 1$$

Figure 1.1. $GF(4) = \mathbb{Z}_2[x]/(x^2 + x + 1)$

3.1.6. Field Extensions

As a result of the previous sections, we have a field $GF(q)$ where $q = p^n$ for some prime p, represented as $\{0, \alpha, \cdots, \alpha^{q-1}\}$. We now wish to construct a field $GF(q^m)$ for some $m > 1$, called the *m-degree extension* of $GF(q)$. We will call $GF(q)$ the *base* field. The field extension will be represented as the set of polynomials over $GF(q)$ modulo an irreducible polynomial $f(t)$ of degree m. Thus we can represent the elements of $GF(q^m)$ by choosing the polynomials of degree $\leq m$ with coefficients from the base field, $\{0, \alpha, \alpha^2, \cdots, \alpha^{q-1} \equiv 1\}$, as equivalence class representativities.

We proceed in a manner analogous to the one used to construct the base field. That is, generator g is found and the nonzero "polynomials" comprising the field extension are represented as powers of g. Notice that this representation of $GF(q^m)$ is isomorphic (by Theorem 6) to the field $GF(p^{n \cdot m})$, so p is again the characteristic of the field. This relationship is shown pictorially in figure 1.2.

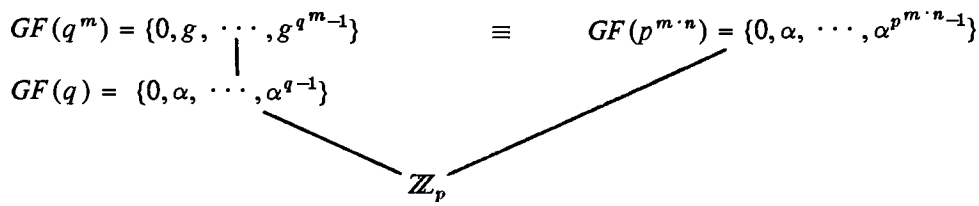

Figure 1.2. Relationship between Base Field and Field Extension.

Continuing with our example, we now construct $GF(4^3)$, an extension of $GF(4)$. First we find an irreducible polynomial, $f(t)$, of degree 3 over $GF(4)$; $f(t) = \alpha^3 t^3 + \alpha^3$ works. Next we find a generator which we will now call g. The polynomials (modulo $f(t)$) comprising the field will have degree ≤ 2, and we find that $g = \alpha^3 t + \alpha^3$ is indeed a primitive element. The entire field is given in appendix A, due to its size, and for convenience in subsequent examples. Notice that $g^{63} \equiv \alpha^3 \equiv 1$, while no other elements are 1, so g is a generator.

We need one more concept to prove the Bose-Chowla theorem in the next section. If E is a field extension of F, the element $e \in E$ is *algebraic of degree n* over F if n is the smallest integer and there exist $a_0, a_1, \cdots, a_n \in F$, not all zero, such that

$$a_0 + a_1 e + \cdots + a_n e^n = 0.$$

In other words, the minimal polynomial in F having e as its root is of degree n.

3.2. The Bose-Chowla Theorem

We have now provided the necessary background to provide a proof of the Bose-Chowla theorem [Bose62]. This version of the theorem was slightly modified by Chor from the original to better suit the application, and we have further extended Chor's version to suit our implementation of the Chor-Rivest system (we have allowed p to be a prime power). This theorem demonstrates that we can form a p element knapsack vec-

tor $\vec{a}$ whose h-fold sums are unique. At the same time the magnitude of each knapsack elements is confined to a polynomial (p^h) range, so high density is insured. For contrast, consider a superincreasing knapsack of p elements where successive elements are growing exponentially, rather than polynomially.

The Bose-Chowla Theorem
Let p be a *prime power*, $h \geq 2$ an integer. Then there exists a vector $\vec{a} = \{a_i : 0 \leq i \leq p-1\}$ of integers such that

1. $1 \leq a_i \leq p^h - 1 \quad (i = 0,1, \cdots, p-1)$;

2. If $(x_0, x_1, \cdots, x_{p-1})$ and $(y_0, y_1, \cdots, y_{p-1})$ are two distinct vectors with non-negative integral coordinates and

$$\sum_{i=0}^{p-1} x_i = \sum_{i=0}^{p-1} y_i \leq h,$$

then

$$\sum_{i=0}^{p-1} x_i a_i \neq \sum_{i=0}^{p-1} y_i a_i.$$

PROOF

Consider the Galois field $GF(p)$ having distinct elements $\alpha_0 = 0, \alpha_1, \cdots, \alpha_{p-1}$.

Let g be a primitive element of $GF(p^h)$, an h-degree extension of the field $GF(p)$. That is g is a generator of the multiplicative group of nonzero elements in $GF(p^h)$.

Let t be algebraic of degree h over $GF(p)$, and consider $GF(p^h) = GF(p)(t)$.

Construct vector $\vec{a} = (a_0, a_1, \cdots, a_{p-1})$ where

$$a_i = \log_g(t + \alpha_i), \quad i = 0, \cdots, p-1 \tag{1}$$

and suppose that

$$\sum_{i=0}^{p-1} x_i a_i = \sum_{i=0}^{p-1} y_i a_i. \tag{2}$$

Then in $GF(p^h)$ we have

$$g^{\sum_{i=0}^{p-1} x_i a_i} = g^{\sum_{i=0}^{p-1} y_i a_i},$$

so,

$$\prod_{i=0}^{p-1} (g^{a_i})^{x_i} = \prod_{i=0}^{p-1} (g^{a_i})^{y_i}.$$

Expanding these products, we obtain

$$(t+\alpha_0)^{x_0}(t+\alpha_1)^{x_1}\cdots(t+\alpha_{p-1})^{x_{p-1}} = (t+\alpha_0)^{y_0}(t+\alpha_1)^{y_1}\cdots(t+\alpha_{p-1})^{y_{p-1}}$$

Since $\vec{x}$ and $\vec{y}$ are distinct, and the sum of their elements is h, both sides of the above equation represent distinct monic polynomials of degree h. We now subtract these to yield a nonzero polynomial of degree $<h$ with coefficients in $GF(p)$ that equals zero.

Hence, t is a root of this polynomial, contradicting the fact that t is algebraic of degree h over $GF(p)$.

Therefore, assumption (2) leads to a contradiction.

QED

Our method of constructing knapsacks in this scheme is to mimic this construction of $\vec{a}$. For example, in the field $GF(4^3)$ tabulated in appendix A we obtain $\vec{a} = (56, 1, 58, 25)$, which may be easily verified for compliance with the Bose-Chowla theorem. That is, any 3-fold sum (repetitions allowed) is unique.

CHAPTER 2

Project Description

1. How the Chor-Rivest Cryptosystem Works

1.1. Key Generation

Creation of the key pair, the most demanding phase of this public key cryptosystem, is outlined in figure 2.1. The first step is to find a prime power p and a positive integer $h < p$ such that discrete logarithms over $GF(p^h)$ may be easily computed (i.e.

---

(1) Let p be a *prime power*, $h < p$ an integer such that discrete logarithms in $GF(p^h)$ can be efficiently computed (i.e. $p^h - 1$ factors into small factors only).

(2) Construct the Galois field $GF(p)$, then compute and tabulate $\alpha + \beta$, $\alpha \cdot \beta$, $\alpha - \beta$ and α / β for all $\alpha, \beta \in GF(p)$.

(3) Find a monic degree h polynomial $f(t)$ that is irreducible over $GF(p)$, and $g \in GF(p^h)$, a multiplicative generator of $GF(p^h)$. We thus define $GF(p^h)$, an h-degree extension of $GF(p)$, where arithmetic is done modulo $f(t)$ over $GF(p)$ (using the tables created in step 2). This is done by randomly selecting candidates for $f(t)$ and g until the properties $g^s \neq 1$ for all $s \mid p^h - 1$ and $g^{p^h - 1} = 1$ are satisified.

(4) Construct the knapsack vector $\vec{a}$ following the Bose-Chowla theorem: Compute $a_i = \log_g(t + i)$ for $i = 0, 1, \cdots, p-1$.

(5) Scramble the knapsack $\vec{a}$ from the original ordering: Let $\pi: \{0,1, \cdots, p-1\} \rightarrow \{0,1, \cdots, p-1\}$ be a randomly chosen permutation. Set $b_i = a_{\pi(i)}$.

(6) Add noise to the knapsack: Pick $0 \leq d \leq p^h - 1$ at random. Set $c_i = b_i + d$.

(7) Public key - to be published: $c_0, c_1, \cdots, c_{p-1}; p, h$.

(8) Private key - to be kept secret: $f(t), g, \pi^{-1}, d$.

Remark: Every user can use the same p and h. The probability of collisions (two users having the same keys) is negligible.

Figure 2.1. Outline of key generation phase of Chor-Rivest [Chor85].

p^h-1 factors into small factors only). Throughout the remainder of this paper, p and h are reserved for these parameters. The desired span for these parameters is $p \leq 256$ and $h \leq 25$. In steps 2 and 3 of figure 2.1 a field $GF(p)$ and its h-degree extension are defined as described last chapter. However, the orders of field extensions we are interested in, precludes computing and saving the entire field, so Theorem 8 is applied.

Step 4 is the Bose-Chowla construction, also covered previously, yielding a dense knapsack vector, $\vec{a}$. The Pohlig-Hellman algorithm (§ 4.3) is used to extract logarithms.

This knapsack is scrambled using a random permutation, and a noise factor is added to each element. This disguised knapsack $\vec{c}$ along with p and h form the public encryption key. The elements t, g and the permutation applied to the knapsack constitute the private decryption key.

1.2. Encryption

Encryption, figure 2.2, proceeds as in previous knapsack systems. To encrypt a binary message add the knapsack elements corresponding to a 1 bit in the plaintext. Notice that this operation is very fast since only addition (modulo p^h-1) is required. The only complication here is that we must add very large numbers, exceeding the capacity of a machine integer.

1.3. Decryption

Decryption, figure 2.3, based on exponentiation of the generator g, is more complex than encryption. In step 2, the noise factor d is removed from the ciphertext C giving c. The generator is raised to the power c, resulting in degree $h-1$ polynomial $q(t)$. Recall that field elements are equivalence class representatives. In step 4 we are simply finding the h-degree member $s(t)$ of the equivalence class denoted by $q(t)$ (that is, $q(t) \equiv s(t) \pmod{f(t)}$).

To encrypt a binary message $\vec{m}$ of length p and weight (number of 1's) *exactly* h, into the ciphertext C:

$$C = \sum_{i=0}^{p-1} m_i \cdot c_i \pmod{p^h - 1},$$

where $\vec{c}$ is the knapsack contained in the public key.

Figure 2.2. Outline of encryption phase of Chor-Rivest [Chor85].

(1) Let $r(t) = t^h \bmod f(t)$, a polynomial of degree $\leq h - 1$.

(2) Given the ciphertext C, compute $c = C - hd \pmod{p^h - 1}$.

(3) Compute $q(t) = g^c \bmod f(t)$, a polynomial of degree $h - 1$.

(4) Add $t^h - r(t)$ to $q(t)$ to get $s(t) = t^h + q(t) - r(t)$, a polynomial of degree h in $GF(p)[t]$.

(5) We now have

$$s(t) = (t + i_1) \cdot (t + i_2) \cdots (t + i_h)$$

namely $s(t)$ factors to linear terms over $GF(p)$. By successive substitutions, we find the h roots i_j's. Apply π^{-1} to recover the coordinates of the original plaintext message having the bit 1.

Figure 2.3. Outline of decryption phase of Chor-Rivest [Chor85].

Since $s(t)$ has degree h, we expect h roots for $s(t)$.
Observe that

$$s(t) \equiv g^c \equiv g^{(a_{i_1} + a_{i_2} + \cdots + a_{i_h})},$$

where the a_i's are h elements of the original (unscrambled, unpermuted) knapsack $\vec{a}$ formed in step 4 of the key generation. Each a_i is the logarithm of a linear polynomial, thus, $s(t)$ has h linear factors! In step 5 the roots are found and unscrambled to reveal the original plaintext.

2. Functional Specification

2.1. Design Criteria

Figure 2.1 contains the remark that

> Every user can use the same p and h. The probability of collisions (two users having the same keys) is negligible.

Pursuit of this option imbues advantages to a system: Because these parameters are constants they can be *hard-wired* into the system along with several other important values like the factorization of p^h-1, and the size of polynomials in the fields. The system can be *tuned* to the selected values, and coding tricks employed[1], to achieve the greatest efficiency possible. Thus we expect Chor-Rivest installations to adopt a particular pair p and h as a matter of course.

Our primary aim is to generate data for the cryptanalyst, so we need the capability to emulate any given installation. That is the implementation described herein shall operate for arbitrary choice of p and h within the constraints set forth in figure 2.4. The third constraint derives from the choice of algorithm to compute discrete logarithms. The Pohlig-Hellman algorithm, selected at the outset of the project, is the most general discrete logarithm procedure recommended by the Chor [Pohl78, Chor85].

---

(1) $p < 256$ must be a prime power.
(2) $h \leq 25$ is a positive integer strictly less than p.
(3) The factorization of p^h-1 may consist of at most 32 distinct primes, the largest being less than 10^8.

Figure 2.4. Constraints on the choice of p and h.

[1]This is particularly true for fields with characteristic 2.

Secondarily, we aim to give the user control over the system. In particular, he must have access to intermediate results and the ability to override normally randomly generated values if he so desires.

The third major objective is that the implementation should be as efficient as possible. The rational for ranking this objective is that the system is an analytical tool not a production cryptosystem. Thus, where generality and efficiency are in conflict the former will prevail. However, due to the complexity of the system, maximal efficiency must be regarded as a critical goal. This is the impetus for selecting the C programming language.

Two final objectives are all encompassing: The system shall be well documented and easily modified, and it shall be immune to user error through the incorporation of consistency checks and error messages. The design criteria are shown in table 2.1.

2.2. Functions Performed

There are three main functions performed by this system, corresponding to the phases of the Chor-Rivest cryptosystem: a) System Generation, b) Encryption, and c) Decryption. The first produces matched public/private key pairs to be used as inputs to the Encryption and Decryption functions respectively. The encryption function accepts the public key and a binary plaintext as input and outputs the ciphertext. Conversely, the decryption function uses the private key to transform the ciphertext message back to

Table 2.1. Primary Design Concerns.

Rank	Criteria
1	Easy to modify and well documented
2	Robust
3	Emulate most other installations
4	Allow maximal user control
5	Efficient

the original plaintext.

2.3. Limitations and Restrictions

Keep in mind that this product is designed under the assumption that the cryptanalyst is the end user. No attempt is made to deliver a system that will be usable by a community of potential correspondents. There are no provisions for establishing or maintaining a public key directory nor are there any message passing protocols. These concerns constitute a separate problem in the realm of networks and communication.

A further restriction, is on the format of messages accepted for encryption. In selecting values for the system parameters the user also defines the allowable set of plaintext messages. A message submitted to the enciphering facility must consist of precisely p bits, h of which are ones (white space is ignored). There is no provision for the arbitrary mappings of ASCII, EBCDIC, or natural language to this format.[2]

2.4. User Inputs

The parameters p and h as well as a file containing the factorization[3] of p^h-1 are required as command line arguments to the **generate** program. Additionally, if the option to override random generation is selected (also a command line argument), the user is prompted for several values.

The **encryption** and **decryption** programs require the appropriate key file to be the first argument on the command line (unless options are selected). The remaining arguments are to be files containing binary plaintext or the multiprecision ciphertext respectively. An arbitrary number of message files may be submitted to either program. Additionally, **decrypt** permits an arbitrary number of ciphertext messages within each file

[2]Chor provides simple algorithms for doing this [Chor85]. We chose not to incorporate them in order to keep the primary algorithm clear.

[3]An auxiliary program **factor**, described in the users manual, will perform this factorization.

(separated by a newline character).

2.5. User Outputs

If no options are specified, the program **generate** runs silently, creating two files containing the keys for encryption and decryption. A verbose option is provided causing important intermediate results to be printed to **stdout**. The dump option is similar to the verbose option, except that it applies exclusively to values generated within the logarithm computations.

The program **encrypt** simply places the ciphertext produced on the **stdout**. If more than one plaintext file was submitted for encryption, the associated ciphertext is output one message per line, in order. This output may be redirected into a file to be submitted to the deciphering function, since decrypt allows an arbitrary number of messages in each ciphertext file.

The function **decrypt** also places each plaintext message on the **stdout** followed by a newline character. To enhance readability, these binary messages have a space after every forth bit. Like the generate command, a verbose option is provided, to reveal intermediate results.

A clock option is also provided for the above commands. In the generate procedure the time required for each step of the computation is output. In the encrypt and decrypt functions, the time to transform each individual message is printed, as is the time to load the key file and do initial calculations.

The system is very robust. If an error occurs, a message is placed on the **stderr** file, and the program exits with a value of 1. Successful completion returns a value of 0.

3. Changes to Original Scope

At the outset of this project an interactive system incorporating all phases of the Chor-Rivest cryptosystem along with several additional functions to aid the cryptanalyst was envisioned. Of course it was imperative to permit arbitrary choice of the parameters p and h (in the proposed range) to emulate any other Chor-Rivest cryptosystem. These objectives are conflicting however; there are simply not enough resources (space and time) available on the systems at Rochester Institute of Technology. Even if sufficient core were available, given the time requirements an interactive approach is not appropriate: One is more inclined to run the key generation as a background process under UNIX.

In order to allow the user to manipulate intermediate values, all variables must be accessible, regrettably this is not viable due to the complexity of computing logarithms. Indeed, it is a struggle to accommodate the primary requirement for arbitrary system parameters in the proposed range. A very frugal attitude toward storage evolved in this programmer as the coding progressed, thus functions to take arbitrary logarithms, display field elements or re-execute commands are not provided.

CHAPTER 3

Implementation Details

1. Architectural Design

The advantages of a modular style in designing computer software are widely recognized. To this end, the C language supports and encourages separate compilation of files [Kern78]. Through disciplined programming, files can be equivalent to modules. We assimilate this approach, so the system is split across several files, with related functions or header information collected in each file. In this chapter we describe the system architecture, showing how the modules fit together to make up the system. The UNIX *make* facility aids the programmer in this regard, allowing him to specify dependencies among numerous files in a special file called the *Makefile* [Feld79]. The *Makefile* for this system, reproduced in appendix C, supplies much the same information as this chapter, but in a more concise and precise rendering. It should be perused before modifying, or extracting any part of the system.

Four executable commands constitute the cryptosystem:

a) *generate*, the key generation facility;
b) *encrypt*, the encryption facility;
c) *decrypt*, the decryption facility;
d) *factor*, computes data necessary for key generation.

The main program for each command resides in a module of the same name, with a ".c" suffix. These modules also contain high-level routines called by the given main program.

At the other extreme are the modules which house related low-level routines:

a) *poly.c*, the polynomial manipulation package;

b) *mp.c*, the multiprecision package for handling huge integers;
c) *error.c*, routines to handle various system errors;
d) *utility.c*, miscellaneous low-level routines.

The functions in these files are primitive operations for the routines in the other files. Similarly, low level definitions and declarations utilized by all program units are encapsulated in modules:

a) *types.h*, global typedefs, macro definitions and constants;
b) *mp.h*, constants and macros used in the multiprecision package;
c) *global.c*, declarations of global variables.

These universally accessible units are examined in detail in § 3 of this chapter.

The remaining modules are for the major subphases of the key generation phase of the cryptosystem:

a) *gf.c*, to construct Galois fields;
b) *logarithm.c*, the Pohlig-Hellman discrete logarithm procedure;
c) *chinese.c*, the Chinese remaindering technique.

Isolating these fosters easy modification, replacement, or transfer to another application. The bulk of the programming effort centered on the procedures contained in these modules and scrutinized in chapter 4.

2. Naming Conventions

To enhance the readability and modifiability of the code, some guidelines for identifier names were adopted. Of course, names of variables and constants suggest their purpose as much as possible. Where published algorithms have been used, the source is referenced in a comment, and the same variable names are used if possible. The names from Chor's specification of the cryptosystem take precedence if there is a conflict [Chor85].

If a variable is a polynomial, it has the form a_b where a is its name and b is the formal variable. Again to be consistent with Chor, t is the formal variable for elements

of the field extension whereas x implies a polynomial in the base field. So for instance, the variable *g_t* holds the polynomial *g(t)*, the generator of the field extension.

A function to manipulate polynomials exclusively in the field extension begins with a capital letter, to distinguish it from the complementary function which acts over the small field. An example is the pair *Mult( )* and *mult( )* to multiply elements of the respective fields.

All externally visible multiprecision functions have entirely capitalized names, like *MULT( )* to multiply large integers.

3. Data Structures and Global Variables

There is a set of variables that so pervade the system it is desirable to make them accessible to all routines. By doing so, we avoid cluttering functions with arguments that are extraneous to their basic purpose. Even more importantly, we avoid the overhead of stacking and unstacking these variables on each function call. Due to the intensive computations performed by this system, this is an efficiency concern that cannot be overlooked. The file *global.c* contains the declarations for all of these global variables.

The parameters p and h, the most heavily referenced variables of the system, are declared to be short integers. These two and other values computed from them are the largest subset of the global variables. The value of p^h-1 is saved in the MP variable *p_h_m1*, the logarithm to the base 2 of this number is kept in the int *p_h_log*. The prime factorization of p^h-1 is represented by the structure

```
struct prime_factorization {
        int nf;                 /* the number of prime factors */
        int f[MAX_nf];          /* the prime factors           */
        int n[MAX_nf];          /* exponents of the factors    */
        int r[MAX_nf];          /* relatively prime factors    */
        MP  S[MAX_nf];          /* p_h_m1 / f [i]              */
} q;
```

The relationship between the components of this structure are

$$p^h - 1 = \prod_{i=0}^{nf-1} f[i]^{n[i]} = \prod_{i=0}^{nf-1} r[i].$$

Table 3.1, shows the values structure components for the case $p = 9$ and $h = 3$, where $p^h - 1 = 728 = 2^3 \cdot 3^1 \cdot 13^1$.

Table 3.1. Components of struct prime factorization for p = 9, h = 3.

i	f[i]	n[i]	r[i]	S[i]
0	2	3	8	364
1	7	1	7	104
2	13	1	13	56

The generator of the field $GF(p^h)$ is maintained as the global variable *g_t*. The irreducible polynomial *f_t* is also global.

The only other global variables are flags to control the amount of input and output of the programs. These variables are set by the user as options on the command line, and are referenced by many functions.

The file *types.h* is a header file of type, macro and constant definitions that is included in all of the source files. Constants are used to determine the space allocated at compile time, as well as for error checking at run time. These constants, and their current values are:

```
#define MAX_p      256
#define MAX_h      25
#define MAX_log    200
#define MAX_nf     32
#define SIZE       14
```

MAX_p and MAX_h are the maximum allowed values of the global system parameters p and h. The values 256 and 25 are the largest values proposed by Chor and Rivest [Chor84]. MAX_log is the greatest value for $\lceil \log_2(p^h - 1) \rceil$ used to allocate space for the powers of 2 of the generator of the field extension $GF(p^h)$. Notice the consistency of

these values: $\lceil \log_2(256^{25}-1) \rceil = 200$.

MAX_nf is the greatest number of factors permitted in the factorization of p^h-1. Because the Chinese remaindering technique uses a divide and conquer approach to convert from modular notation to radix notation, efficient implementation requires that MAX_nf is a power of 2: Our choice of 32 should cover all cases in our range of p and h.

The constant SIZE is used to allocate space for multiprecision variables used by the program. A multiprecision number is actually a representation of the given number in base 2^{30}. The typedef for these large numbers is

```
#typedef    int                    MP[SIZE];
```

where the zeroeth array element holds the number of 2^{30} digits in the number. Therefore, we can represent numbers to $2^{30^{13}}$ which is adequate for our purposes.

The other typedefs are

```
#typedef    unsigned char          coeff;
#typedef    coeff                  *polynomial;
#typedef    MP                     *knapsack;
```

Thus a polynomial variable is actually a pointer to coeff, short for coefficient. Each is dynamically allocated according to its degree D via the macro

```
#define poly_alloc(D)    (polynomial) calloc( (D+1), sizeof(coeff))
```

so a polynomial behaves as an array of coefficients, where the index corresponds to the power of the formal variable. In field $GF(p)$ the coefficients are from the ring $\mathbb{Z}_p$. The coefficients of $GF(p^h)$, on the other hand, are elements of $GF(p)$ expressed as powers of the generator α. For example, consider $x(t) \in GF(4^3)$:

$$x(t) = \alpha^3 t^2 + \alpha^2.$$

In the internal representation of this polynomial, we store only the powers of α or zero[1]:

$$x\_t \leftarrow (2, 0, 3).$$

Type *coeff* is synonymous with the base type *unsigned char*. This rather uncommon type is an eight bit byte on most machines, providing the decimal range $0 \cdots 255$. Inasmuch as MAX_p − 1 = 255 is the maximum value to be encountered (in either field), this type is tailor made for this implementation. If the constant *MAX_p* is increased, then this typedef needs to be changed accordingly.

A similar macro is used to dynamically allocate space for a knapsack which is a vector of multiprecision numbers having A elements:

```
#define knapsack_alloc(A)          (knapsack) calloc( (A), sizeof(MP))
```

[1]The case $\alpha^0 = 1$ never arises in our implementation, only $\alpha^{p^h}-1$ is used to denote 1. Thus the value 0 as a coefficient is handled as a special case: it always means 0 not α^0.

CHAPTER 4

Algorithm Design and Analysis

In this chapter we examine the primary algorithms needed to implement the Chor-Rivest cryptosystem. The polynomial routines are presented first, since the field construction and discrete logarithm procedures use these as primitive operations.

1. Polynomial Manipulation

1.1. Multiplication of Field Elements

The procedure *Mult(u_t, v_t, r_t)*, shown in figure 4.1, multiplies polynomials $u(t)$ and $v(t)$ over $GF(p^h)$ placing the residue modulo the irreducible polynomial $f(t)$ in $r(t)$. This is the most heavily used code in the system - profiling the execution of *generate* over moderately sized fields (order $\sim 10^{25}$) reveals that

a) *Mult( )* is called in the neighborhood of 250,000 times,
b) accounting for roughly 80% of the execution time.

First $x = u \cdot v$ is found (step 1), if the degree of x (step 2) exceeds h, the degree of the irreducible polynomial f, then x / f is found (step 3) and the remainder is placed in r (step 4). The calls to *plus( )*, *times( )*, *minus( )*, and *div( )* return the results of arithmetic on the coefficients over $GF(p)$. Each of steps 1 and 3 require h^2 of these basic operations, so the overall complexity is $O(2h^2)$. For efficiency's sake, these are macros which do a table lookup of precomputed values.

Since this *Mult( )* is compulsory for decryption, the arithmetic tables over $GF(p)$ must somehow be made available to the decryption facility. We admit the addition and multiplication tables to the private key, then compute the subtraction and division tables

```
Mult( u_t, v_t, r_t )          /*  r = u * v (mod f_t) in GF */
polynomial u_t, v_t, r_t;
{
        extern short p, h;                          /* for GF( p^h )*/
        extern polynomial f_t;                      /* irreducible  */
        short i, j, q, m, x_t[2*MAX_h+1];

        for ( i.= 0; i < 2*h; i++ ) x_t[i] = 0;     /*  init x      */
1.      for ( i = 0; i < h; i++ )                   /*  x = u * v   */
                for ( j = 0; j < h; j++)
                        x_t[i+j] = plus( x_t[i+j], times( u_t[i], v_t[j] ));
2.      for ( m = 2*(h-1); x_t[m] == 0; m-- )       /*  find degree */
                ;                                   /*    of x      */
3.      for ( i = m-h; i >= 0; --i ) {              /*  x = x % f_t */
                q = div( x_t[h+i], f_t[h] );
                for ( j = h+i-1; j >= i; --j )
                        x_t[j] = minus( x_t[j], times( q, f_t[j-i] ));
                }
4.      for ( j = 0; j < h; j++ )                   /* put remainder*/
                r_t[j] = x_t[j];                    /*   into r     */
}
```

Figure 4.1. Function to multiply field elements.

from these.

1.2. Exponentiation of Polynomials

In figure 4.2 the function *Raise_poly(x_t, e, r_t)* computes $r(t) = x^e(t)$, by combining the powers of two in the binary expansion of e. Initially the result is set to one, and the current power of x, called xp is set to zero. If e is odd, then the result is multiplied by x raised to the 2^{xp} power. Then xp is incremented, and e is halved (integer division). The steps are repeated until the value of e reaches zero, so at most $2h$ log p modular multiplications are needed. The procedure *Mult()* entails at most $2h^2$ primitive operations over the base field. Consequently $4h^3$ log p operations over $GF(p)$ will suffice to exponentiate over $GF(p^h)$.

```
Raise_poly( x_t, e, r_t )              /* exponentiate polys */
polynomial x_t, r_t;                   /*   over GF          */
MP e;                                  /*     r = x^e        */
{
        short i;
        coeff xp_t[MAX_h];             /* curr pwr of 2 of x */
        MP ex;

        ASSIGN( ex, e );               /* copy so no side eff */
        for ( i = 1; i < h; i++ )      /* initialize result   */
                r_t[i] = 0;            /*   to const poly     */
        r_t[0] = p-1;                  /*   "one"             */
        for (i = 0; i < h; i++)        /* initialize xp_t to  */
                xp_t[i] = x_t[i];      /*     x_t             */

        while (COMP( ex, ZERO )) {
                if (ex[1] & 1)         /* ex odd ?            */
                        Mult( r_t, xp_t, r_t );
                Mult( xp_t, xp_t, xp_t );
                HALF( ex );
        }
}
```

Figure 4.2. Procedure to exponentiate a polynomial over $GF(p^h)$.

1.3. Evaluation of Polynomials

Horner's Rule (also known as synthetic division) has been shown to require the least number of additions and multiplications to evaluate a polynomial [Aho74]. Observe that a polynomial

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n + \cdots$$

may be expressed as

$$f(x) = a_0 + x(a_1 + x(a_2 + x(\cdots x a_n)\cdots))).$$

This observation is the basis of Horner's Rule which is shown in coded form in figure 4.3 (for the field extension).

```
coeff Evaluate( a, x, n )    /* Horner's Rule over GF          */
polynomial a; coeff x;
short n;
{
        coeff total;

        total = a[n];
        while ( --n >= 0 )
                total = plus(times(total, x), a[n]);
        return total;
}
```

Figure 4.3. Function to evaluate a polynomial using Horner's Rule.

1.4. Generation of Random Polynomials

The function *rndm_poly(u, a, b, d)* forms a somewhat random polynomial u of degree d. When the second argument a is not zero, then the leading coefficient of u will be set to a; with $a = 1$ this forces a monic polynomial. The third argument b defines the upper limit on the value of each coefficient in u, thus this procedure works for both the base field or its extension.

The function works by setting each coefficient to

$$random() \% b,$$

where *random()* is the C library function returning a pseudorandom integer.[1] If per chance the constant term is zero (implying that zero is a root), it is changed to one instead.

To select a candidate polynomial $f(t)$ hopefully irreducible over $GF(p^h)$ use the procedure *Irreducible(f_t)*. This procedure begins with a call to *rndm_poly()* for a monic degree h polynomial over $GF(p)$, then employs some hueristics to improve its

[1]Function *random()* is not available on all UNIX systems. An alternative is the older *rand()*.

chances for irreducibility.

The first hueristic is that a sparse polynomial is more likely irreducible than a dense polynomial, since every missing term i precludes at least one pair of factors having degrees totaling i. However, the polynomial received from *rndm_poly()* will never be sparse due to the nature of the function *random()*. We thus make a random number (between 1 and $h-1$) of coefficients zero.

This sparse polynomial is then checked for linear factors using *Evaluate()* for each element of $GF(p)$. If a root is found we start a new, and repeat the hueristics until a suitable candidate is produced.

While one could add several other tests, to the extreme of completely factoring a candidate, experience has showed that this is not prudent. For $GF(197^{24})$, as an example, seventy candidates were tried before finding a legitimate irreducible polynomial prior to the addition of the "sparsing step" to the code. Afterwards, only three attempts were needed, while *Irreducible()* remains very fast.

2. Construction of Galois Fields

An algorithm to construct fields is central to the key generation. Two Galois fields are created, the base field $GF(p)$ and its h-degree extension, $GF(p^h)$. Two separate but similar procedures are needed, *compute_gf()* and *compute_GF()*, respectively. The major difference is that in the former arithmetic is over the ring of integers modulo the characteristic of the field, whereas in the latter arithmetic is over the base field expressed as powers of the generator. In fact, once tables have been made for arithmetic over $GF(p)$, this field is discarded (dynamically allocated space is returned). During the creation of these tables the entire field has to be resident in memory. This suggests the only other difference between *compute_gf()* and *compute_GF()*: there is no way to store all the elements of $GF(p^h)$ for the proposed magnitudes of p and h.

Figure 4.4 is the code for *compute_GF()*, notice that only the powers of 2 of the generator candidate g, $\{g^{2^i} \mid 0 \le 2^i < \log_2(p^h)\}$, are found in step 3. Having the powers of 2 of the generator candidate in hand, we check to see if g[0] is indeed a primitive element, and whether t is algebraic over $GF(p)$, following Theorem 8 of chapter 2. Step 4, computes $g\_phm1\_t = g^{p^h-1}$, which if equal to 1 (actually the multiplicative identity, $p-1$), satisfies half of the criteria for Theorem 8. The other half is checked in step 5:

```
compute_GF()                    /* define GF by finding f(t) &  g^2^i  */
{
        short i;                        /* loop control var             */
        extern polynomial h_t[];        /* in logarithm.c               */
        polynomial g_phm1_t;            /* temporary polynomial         */

1.      for (i = 0; i <= p_h_log; i++) g[i] = poly_alloc( h-1 );
        for (i = 0; i <= q.nf; i++)  h_t[i] = poly_alloc( h-1 );
        f_t = poly_alloc( h );
        g_phm1_t = poly_alloc( h-1 );

    select_g:
2.      Irreducible( f_t );
        rndm_poly( g[0], 0, p, h-1 );

3.      for (i = 1; i <= p_h_log; i++)          /* g to pwrs of 2 */
                Mult( g[i-1], g[i-1], g[i] );

4.      raise_g( p_h_m1, g_phm1_t );            /* insure that g^p^h = 1 */
        if ( !one( g_phm1_t, h-1, p-1 )) goto select_g;

5.      for (i = 0; i < q.nf; i++) {            /* g to p_h_m1/factors */
                raise_g( q.S[i], h_t[i] );
                if (one( g_phm1_t, h-1, p-1) ) goto select_g;
                }

6.      for (i = 1; i <= p_h_log; i++)          /* free all but g[0] */
                cfree( g[i] );
}
```

Figure 4.4. Procedure to construct a Galois field, $GF(p^h)$.

that $g^{S_i} \neq 1$ where $S_i = (p^h-1)/f_i$, for each of the prime factors f_i of p^h-1. These special polynomials are saved in the array of polynomials h_t[ ] for later use in computing logarithms. If either of these conditions is not satisfied, we simply start over, thus the infamous **goto** statement best captures the essence of the algorithm, so it is unabashedly used.

Step 1 of the code dynamically allocates space for the polynomials required, step 6 returns this space, except that the generator itself must be saved as part of the private key.

The procedure *raise_g()* is a preconditioned form of the procedure *Raise_poly()* of figure 4.2, using the powers of 2 of g computed in step 3.

3. Discrete Logarithms

To fashion knapsacks according to the Bose-Chowla construction, we need a way to determine logarithms of certain field elements. We choose the Pohlig-Hellman algorithm, applicable to arbitrary $GF(p^h)$, provided that p^h-1 does not have a large prime factor.

The objective is to find $1 \leq a \leq p^h-1$ such that $g^a = x$, given primitive element g and some polynomial x in the field. This algorithm actually produces the vector $\vec{a}$, the modular representation of a, where $a_i = a$ modulo r_i. The relatively prime moduli $r_i = f_i^{n_i}$ where f_i are prime and

$$p^h-1 = \prod_{i=0}^{nf-1} f_i^{n_i}.$$

Chinese remaindering[2] is then employed to transform the vector of residues into its radix equivalent. Each component of $\vec{a}$ is determined in the same manner, so the subscript i

[2]Chinese remaindering is a very old technique to transform numbers in modular notation to radix notation. We have used Aho, Hopcroft and Ullman's procedure directly [Aho74].

is dropped for clarity in the following.

We express a in base f as $(b_{n-1} \cdots b_0)$,

$$a = \sum_{i=0}^{n-1} b_j r^j \pmod{r}.$$

The approach is to determine the b_j starting with the least significant digit b_0, then a is shifted to the right to find b_1 and so on. To do this, observe that

$$g^a = g^{b_{n-1}(p^h-1)/f} \times g^{b_{n-2}(p^h-1)/f^2} \times \cdots \times g^{b_1(p^h-1)/f^{n-1}} \times g^{b_0(p^h-1)/f^n}.$$

To find b_0, raise x to the $(p^h-1)/f$ power:

$$y = x^{(p^h-1)/f} = g^{a(p^h-1)/f} = (g^{(p^h-1)/f})^{b_0}.$$

Because b_0 is a base f digit, its possible values correspond to f possibilities for y:

$$H^0, H^1, \cdots, H^{f-1}$$

where $H = g^{(p^h-1)/f}$; a search reveals $y = H^{b_0}$. Recall the array of polynomials h_t[] saved during *compute_GF()*, these are the H^1 for each factor f.

Having determined b_0, the successive digits, $b_1 \cdots b_{n-1}$, are resolved by forming

$$y = (x \cdot (g^{-1})^{b_0} \cdot (g^{-1})^{b_1 f} \cdots (g^{-1})^{b_j f^j})^{(p^h-1)/f^j}.$$

and again searching for $y = H^{b_j}$.

Clearly, if many logarithms are needed, it is wise to precompute, and perhaps sort, the powers of H for each factor of p^h-1. The complexity of this operation is a function of the sum of these prime factors, both in time and space: each of the $\sum_{i=0}^{nf-1} f_i$ polynomials resulting are found via a polynomial multiplication. For perspective consider logarithm computations in $GF(197^{24})$. There are $nf = 18$ factors whose sum approaches 16 million, hence about 16 million degree 23 polynomials are called for. In figure 4.5 we present a C coded form of the algorithm, in single precision so as not to shroud the logic too much. The code for large fields is identical except that many expressions must be broken down into more primitive multiprecision function calls making the algorithm

harder to follow.

Let us sketch this algorithm as it operates in the key generation phase of the Chor-Rivest system. The insight gained will set the stage for the ensuing discussion of improvements specific to this cryptosystem. Say we have selected $p = 9$ and $h = 3$, then found a generator g of the field $GF(9^3)$. Following the Bose-Chowla theorem, we need to find the nine logarithms, $\log_g(t + i)$ for $i = 0 \cdots 8$.

```
logarithm( x_t, a )      /* preconditioned pohlig-hellman algorithm */
int a[];                 /* the logarithm of x, as a vector of    */
polynomial x_t;                  /* the residues, a mod q.f[i]            */
{
        int i, j, E, b, BP;
        polynomial y_t, Y_t;

        for ( i=0; i < q.nf; i++ ) {        /* for each factor     */
                a[i] = 0;                   /* do initializations  */
                E = p_h_m1;
                for ( j = 0; j < h; j++ ) Y_t[j] = x_t[j];
**              for ( j=0; j < q.n[i]; j++ ) {
                        E /= q.f[i];
                        if (j != 0) {       /* only if q.n[i] > 1  */
                                BP = b * power( q.f[i], j-1 );
                                Raise_poly( g_inv, BP, y_t );
                                Mult( Y_t, y_t, Y_t );
                                }
                        Raise_poly( Y_t, E, y_t );
                        for ( b=0; b < q.f[i]; b++ )
                                if ( same_poly( y_t, H[i][b], h-1)) break;
                        a[i] += b * power( q.f[i], j ) % q.r[i];
                        }
                }
        return TRUE;
}
```

Figure 4.5. C implementation of the Pohlig-Hellman Algorithm.

The preconditioning requires the factorization $p^h-1 = 728 = 2^3 \cdot 7 \cdot 13$. It is apparent that three lists of polynomials are needed having lengths 2, 7, and 13, we will call these lists H_2, H_7 and H_{13} in this example. After computing and saving these 22 polynomials we are ready to compute arbitrary logarithms to the base g.

Each call to *logarithm* will return a three element vector

$$\vec{a} = (a_0 (mod\ 2), a_1 (mod\ 7), a_2 (mod\ 13)),$$

which will be converted into an integer between 0 and 728 by Chinese remaindering. Determining a_0 is more difficult than a_1 or a_2 since the loop marked '**' in figure 4.5 must be traversed three times. In each traversal we do some arithmetic to form the polynomial y, which must match one of the two polynomials in the list H_2. As the positions of the three matches are found they are combined to form a_0.

Moving on to a_1 we see that we only have to form the polynomial y once, then find its position in the list H_7. The case for a_3 is analogous to a_2.

We improve the practical behavior of the Pohlig-Hellman algorithm for this application by capitalizing on the the following observations:

a) we are interested in precisely p logarithms;
b) most of the factors of p^h-1 are raised to the power of unity.
c) the values a_i are independent and may be computed in any order.

The first refinement we make is to compute a_0 for each of the p logarithms needed, then we compute all of the a_1's, and so on. By doing so we realize a fantastic savings in space, for we can do the preconditioning in stages. For the example above, we would first construct the list H_2, then compute a_0 for the nine logarithms needed. At this point we are done with list H_2. As a result we only need space to save a number of polynomials equal to the largest factor of p^h-1, plus the rather trivial table to store each $\vec{a}$ (p of them).

The next refinement is not as obvious, and works only when the current factor f_i appears in the first power ($n_i = 1$). But since this property applies to most factors of p^h-1 it is worthwhile. More importantly, usually only the very smallest factors are raised to powers greater than one. The trick is to do the algorithm backwards. That is, we compute and save the p polynomials y for the current factor, then generate the polynomials H_{f_i}. As each element of H is computed it is checked for a match with the p polynomials, then it is discarded. This approach reduces the number of polynomials stored from $max(f_i)$ to p.

CHAPTER 5

System Performance

The system has been subjected to a battery of test cases, shown in figure 5.1, to verify correct performance. For each test case a minimum of five messages were successfully encrypted and decrypted following key generation.

Table 5.1. Test Cases

p	h	factorization of p^h-1	
		number factors	largest factor
97	**12**	12	3,169
103	**12**	13	31,357
113	**12**	12	4,219
125	**9**	8	31,051
128	9	6	649,657
128	**12**	12	14,449
151	**12**	11	8,522,341
197	**10**	9	77,081
197	**12**	13	36,013
197	**24**	18	10,316,017
211	**12**	14	8,655,349
243	**18**	16	927,001
256	**9**	12	38,337

The trials incorporate values of p, alternately a prime and a prime power, spanning the allowable range.

Having thus validated the system, we turn our attention to the viability of the Chor-Rivest cryptosystem based on our experiences here. To settle this qualitative issue one must consider two elementary questions:

1) Is the security of the cryptosystem satisfactory?
2) Is the efficiency of the cryptosystem satisfactory?

The remainder of this chapter is devoted to data and analysis to help the potential user

judge the applicability of the system to his needs.

1. Security of the Chor-Rivest Cryptosystem

Having no secret information available, one is limited to two known modes of attack on the cryptosystem: *brute force* or *low density*. Both of these will attack only a given instance of the subset sum problem, that is, only one message is revealed.

There are $\binom{p}{h}$ ways to choose h of p elements. In the brute force attack one systematically tries all of the possibilities until h elements are found whose sum matches the ciphertext. Chor and Rivest propose an algorithm with an expected run time of

$$2 \cdot \binom{p}{h} \ln \binom{p}{h} \; / \; \binom{p/2}{h/2}.$$

For p and h in the vicinity of 200 and 25 the brute force attack is impractical requiring more than 2^{58} operations [Chor84]. However smaller values of p and h may well succumb to exhaustive search attacks.

Low density attacks proposed to date may be expected to work up to knapsack densities of 1.0 [Laga83,Radz86]. Chor and Rivest report that the L^3 algorithm was unsuccessful for the knapsacks created with $GF(197^{24})$ as well as $GF(103^{12})$ [Chor84]. Of course it is hoped that this implementation will in some way help to advance the state of the art in this area.

It is impossible to determine the private key, even if the above attacks are successful. Thus, if the message is useful for only a limited time, normal decryption will most certainly be faster than cryptanalysis. To reconstruct the private key one almost certainly has to have a portion of the secret information. The cryptographers have demonstrated that if the noise factor d and any other component of the private key are known then the rest of the key can be construed [Chor84].

2. Efficiency of the Chor-Rivest Cryptosystem

Three components of efficiency are considered: time, space and information rate. For most cryptographic applications, reasonably fast encryption and decryption are tantamount concerns. Also the keys should be compact: this is particularly true in a public key cryptosystem, where the public keys must be maintained in a universally accessible file perhaps, on several machines. Finally, a high information rate is desirable: this implies low data expansion incurred by encryption. Certainly, as the number of bits transmitted is lessened, the speed and cost of operation improves.

2.1. Time Requirements

In the class of public key cryptosystems, it is understood that the key generation is necessarily complex to impart security to the system. Being a one-time computation for each user, we can afford to be lenient in our expectations for this phase. Despite this, the twenty three hours (table 5.2) required by our implementation for p =243, h = 18 would undoubtly discourage use of the system without some overhaul.

Table 5.2. Run Time Statistics

p	h	largest factor	Processor Time (DEC VAX11 780)			
			construct GF	logarithms	encrypt	decrypt
97	**12**	3,169	0:00:35.52	0:36:29.11	0:00:00.07	0:00:01.23
103	**12**	31,357	0:00:42.82	0:42:44.54	0:00:00.05	0:00:01.33
113	**12**	4,219	0:01:27.75	0:41:22.99	0:00:00.07	0:00:01.40
125	**9**	31,051	0:01:00.78	0:15:07.92	0:00:00.07	0:00:00.72
128	**12**	14,449	0:01:19.47	0:39:20.25	0:00:00.08	0:00:01.47
197	**12**	36,013	0:03:43.67	1:37:29.55	0:00:00.12	0:00:01.67
243	**18**	927,001	0:07:07.15	22:50:37.56	0:00:00.08	0:00:05.00
256	**9**	38,737	0:09:27.85	0:44:49.27	0:00:00.12	0:00:00.87

On the positive side, Chor reports that his implementation, specific to $GF(197^{24})$ requires only "a couple of hours on a minicomputer" to generate the system [Chor85].

Encrypting a message of length p and weight h is very fast, requiring the addition of h integers each less than p^h. Decryption, dominated by exponentiation over the field

$GF(p^h)$, is considerably slower. At most $2h \log p$ multiplications are needed where each multiplication consists of $2h^2$ operations using the algorithms of chapter 4. A total of $4h^3 \log p$ operations are needed for decryption.

2.2. Space Requirements

The public key used to encrypt messages with length p and weight h consists of p numbers, $1 \leq c_i \leq p^h - 1$. In this implementation, we have placed the keys in text files, for simplicity and human readability. Where a key directory must be maintained, binary files should be used. The size of a single public key is then $p \log_2 p^h = p\, h \log_2 p$ bits.

The private key, a lesser concern, is also much smaller than the public key. Its four components are two polynomials, a p element permutation and an integer less than $p^h - 1$. The polynomials having degrees h and $h-1$ require one byte for each coefficient or $2h + 1$ bytes total. Each element of the permutation will also fit in a single eight bit byte for the proposed range of p and h. The size in bits of the private key is then

$$| \text{ private key } | \leq 8((2h + 1) + p) + h \log_2 p.$$

We have again placed the private key in a text file, but had to add information specifying the underlying field (i.e. the addition and multiplication tables over $GF(p)$). It would be simple to replace this format with binary files for an actual cryptosystem, and the additional information could be dropped, being common to everybody in the system.

2.3. Information Rate

As noted in chapter 1, the density of a knapsack gives an indication of the information rate R. Thus high density knapsacks offer a twofold advantage: not only is security heightened, but efficiency also benefits. Recall the definition $R = \log_2 | M | / N$ where M is the message space and N is the number of bits in the ciphertext. In the Chor-Rivest scheme the message space is $\binom{p}{h}$ and the ciphertext contains $\log_2 p^h$ bits, so

$$R = \log \binom{p}{h} / \log_2 p^h$$

Table 5.3 summarizes the information rate for several cases.

Table 5.3. Information Rate R vs. Knapsack Density d

p	h	d	R
97	**12**	1.21	1.05
103	**12**	1.27	1.11
113	**12**	1.38	1.22
125	**9**	1.98	1.82
128	**9**	2.03	1.83
128	**12**	1.52	1.34
151	**12**	1.74	1.31
197	**10**	2.56	1.30
197	**12**	2.14	1.24
197	**24**	1.08	.56
211	**12**	2.27	1.23
243	**18**	1.70	.76
256	**9**	3.56	1.60

CHAPTER 6

Conclusions and Recommendations for Future Work

1. Problems Encountered and Solved

Early versions of this system were impractical due to slow execution, particularly in the system generation phase. Profiling the execution of several moderately sized test cases revealed that the program was spending most of its time in functions *minus()* and *div()*, which subtract and divide elements of $GF(p)$. These functions are primitive operations of the function *Mult()* to multiply elements of $GF(p^h)$. By precomputing these values and replacing the functions with table lookups, a massive time savings was realized in exchange for the trivial space required for the tables (each a $p \times p$ array of unsigned char).

Further modifications to *Mult()* yielded still better response. Originally, this function was very general, accepting as arguments the two polynomials to be multiplied, the polynomial modulus, and the degree of each. By taking advantage of the knowledge that the modulus is always a monic h-degree polynomial, and that the multiplicands have degree at most $h-1$, the procedure was streamlined to the form presented in chapter 4. For instance the system generation time for $GF(103^{12})$ was reduced from about eight hours to less than two hours with these simple modifications!

Running this version of the program with larger choice of h uncovered another shortcoming, this one in the code to construct $GF(p^h)$. Once again, the code, though correct, required an unacceptable amount of time to do the computation. Chor's prescription for this phase of the computation is to select at random a degree h monic

polynomial f, irreducible over $GF(p)$. Given f irreducible we are to find a primitive element g of $GF(p^h)$ by randomly selecting an $r \in GF(P^h)$ until one is found which satisfies $r^{(p^h-1)/s} \neq 1$ for all prime factors s of $p^h - 1$.

Our approach was to select f at random, but not necessarily irreducible (we did however check that f had no linear factors). By imposing the additional constraint that $r^{(p^h-1)} = 1$, it was felt that the programming required to explicitly test irreducibility could be avoided. However, the system ran unacceptably slow until the hueristics described in chapter 4 were added.

Time was not the only resource to pose problems. The space requirements of the Pohlig-Hellman algorithm as originally originally implemented were extraordinary: We exhausted the available core part way through the logarithm preconditioning for the interesting values of p and h. The first response to this enigma was to reduce the amount of space required by the program. A major modification ensued, in which every variable was scrutinized and if possible reduced to short or char rather than integer type. The size of multiprecision variables was also halved, and some static variables were converted to automatic variables. After three weeks of squeezing space out of the system, the *Gordian Knot* was finally cut with the realization that the precomputation could follow the Pohlig-Hellman algorithm proper for this application.

2. Discrepancies and Shortcomings of the System

This implementation works for fields that are amenable to discrete logarithm calculations using the Pohlig-Hellman algorithm. Unfortunately, due to the generality imposed on the system, key generation is quite slow. An estimate of the run time for the generate command (in hours) is obtained by summing the distinct factors of p^h-1 then dividing by 275K.

Of the recommended values for p and h given by Chor, all may be run with this program except $GF(256^{25})$ since the largest factor (3,173,389,601) of $200^{25}-1$ does not fit in a VAX machine word. It seems necessary to use the Coppersmith algorithm to evaluate logarithms in this field [Copp84]. However, using this alternative would actually impose more restriction on the choice of fields, since Coppersmith works only for fields of characteristic 2.

3. Recommendations for Future Work

The most obvious extension of the project presented here would be to use the program as it was originally intended. That is, to try to break the Chor-Rivest scheme. The most promising approach seems to be Radziszowski and Kreher's algorithm further modified to handle higher density knapsacks [Radz86].

Another obvious extension is to transform the code into an actual cryptosystem replete with a public key directory and the associated protocols for message passing and maintenance of this directory. Recall that the system parameters p and h would in this case be constant, rather than variable as in the current code. This would open the door for numerous efficiency improvements.

The greatest gains could be realized by fixing p to a power of 2. In this case, the faster Coppersmith algorithm could replace the Pohlig-Hellman algorithm for discrete logarithm calculations. Arithmetic would also be simplified in $GF(2^n)$ since addition and subtraction are identical, and bit operations could replace slower integer arithmetic.

Certain other efficiency gains result regardless of the choice of p and h, as long as they are constant values. Several values that need to be computed at each system generation in the present version could be *hard-wired* into the system. Included in this set are the factorization of p^h-1, and the results of the Chinese remandering precomputation. A significant time savings would result from eliminating the factoring step. Addi-

tionally, storage required for key generation, encryption and decryption could be allocated at compile time, rather than dynamically at run time. Not only would time consuming calls to the operating system storage allocator be eliminated, but the code would be much simpler. The error checking currently required (in case there is no core available) could be eliminated, as well as all *garbage- collecting* code. Finally, the multiprecision package could be modified to take advantage of the known ranges of the program variables. Thus, it would be a very interesting project to modify the current program to a usable public key cryptosystem.

Perhaps even more exciting, is the discrete logarithm problem itself. By extracting the code to construct arbitrary Galois fields and the associated polynomial manipulation routines, one could program several of the prominent algorithms for computing these logarithms, then experimentally evaluate their relative run times. Using the code to construct arbitrary fields and the associated polynomial manipulation routines presented herein, one could focus attention exclusively on the logarithm problem. A good place to begin would be the cataloging of techniques by Odlyzko [Odly86]. This is by no means a trivial project: Some state of the art techniques have not yet been completely analyzed [Odly86].

Bibliography

[Adle83] Adleman, L.M. "On Breaking the Generalized Knapsack Public Key Cryptosystems", *ACM Symposium on Theory of Computing*, New York: ACM. (1983), 402-412.

[Aho74] Aho, A.V., J.E. Hopcroft and J.D. Ullman *The Design and Analysis of Algorithms*, Reading, Mass.: Addison-Wesley Publishing Company, 1974.

[Beke82] Beker, H. and F. Piper *Cipher Systems*, New York: John Wiley and Sons, 1982.

[Bose62] Bose, R.C. and S. Chowla "Theorems in the Additive Theory of Numbers", *Comment. Math. Helvet.*, Vol.37 (1962), 141-147.

[Bric83] Brickell, E.F. and G.V. Simmons "A Status Report on Knapsack Based Public Key Cryptosystems", *Congressus Numerantium*, Vol.37 (1983), 3-72.

[Buel82] Buell, D. "The New Cryptography", Louisiana State University, Baton Rouge, Louisiana, Technical Report #82-022, (1982).

[Chor84] Chor, B. and R.L. Rivest "A Knapsack Type Public Key Cryptosystem Based On Arithmetic in Finite Fields", *Advances in Cryptology: Proceedings of CRYPTO 84*, Berlin: Springer-Verlag, (1985), 54-65.

[Chor85] Chor, B. "Two Issues in Public-Key Cryptography - RSA Bit Security and a New Knapsack Type System", Massachusetts Institute of Technology, Cambridge, Mass., doctoral thesis (1985).

[Copp84] Coppersmith, D. "Fast Evaluation of Logarithms in Fields of Characteristic Two", *IEEE Transactions on Information Theory*, Vol.30, No.4 (1984), 587-594.

[Diff76] Diffie, W. and M. Hellman "New Directions in Cryptography", *IEEE Transactions on Information Theory*, Vol.22, No.6 (1976), 644-654.

[Feld79] Feldman, S. I. "Make - A Program for Maintaining Computer Programs", *Software - Practice and Experience*, Vol.9 (1979), 255-265.

[Gare79] Garey, M.R. and D.S. Johnson *Computers and Intractibility*, New York: W.H. Freeman and Company, 1979.

[Gilb76] Gilbert, W.J. *Modern Algebra with Applications*, New York: John Wiley and Sons, 1976.

[Kern81] Kernighan, B.W. and D.M. Ritchie *The C Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

[Knut81] Knuth, D.E. *The Art of Computer Programming*, Vol.2, Reading, Mass.: Addison-Wesley Publishing Company, 1981.

[Laga85] Lagarias, J.C. and A.M. Odlyzko "Solving Low-Density Subset Sum Problems", *Journal of the ACM*, Vol.32, No.1 (1985), 229-246.

[Lens82] Lenstra, A.K., H.W. Lenstra, Jr. and L. Lovasz "Factoring Polynomials with Rational Coefficients", *Mathematische Annalen*, 261 (1982), 515-534.

[Lens83] Lenstra, H.W., Jr. "Integer Programming with a Fixed Number of Variables", *Mathematics of Operations Research*, Vol.8, No.4 (1983), 538-548.

[Lewi81] Lewis, H.R., and C.H. Papadimitriou *Elements of the Theory of Computation*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

[Merk78] Merkle, R. and M. Hellman "Hiding Information and Signatures in Trapdoor Knapsacks", *IEEE Transactions on Information Theory*, Vol.24 (1978), 525-530.

[Merk82] Merkle, R.C. *Secrecy, Authentication, and Public Key Systems*, Ann Arbor, Michigan: UMI Research Press, 1982.

[Odly86] Odlyzko, A.M. "Discrete Logarithms in Finite Fields and their Cryptographic Significance", (1986) preprint.

[Pohl78] Pohlig, R.C. and M. Hellman "An Improved Algorithm for Computing Logarithms Over $GF(P)$ and its Cryptographic Significance", *IEEE Transactions on Information Theory*, Vol.24 (1978), 106-110.

[Radz86] Radziszowski, S.P. and D.L. Kreher "Solving Subset Sum Problems with the L^3 Algorithm", (1986) preprint.

[Radz86a] Radziszowski, S.P. and D.L. Kreher, private communication (1986).

[Rive78] Rivest, R., A. Shamir, and L. Adleman "A Method for Obtaining Digital Signatures and Public-key cryptosystems", *Communications of the ACM*, Vol.21, No.2 (1978), 120-126.

[Sham82a] Shamir, A. "Embedding Cryptographic Trapdoors in Arbitrary Knapsack Systems", Massachusettes Institute of Technology, Cambridge, Mass., Technical Report MIT/LCS/TM-230, (1982).

[Sham82b] Shamir, A. "A Polynomial Time Algorithm for Breaking the Basic Merkle-Hellman cryptosystem", *Proceedings of the 23rd Conf. on Foundations of Computer Science*, New York: IEEE, (1982), 145-152.

[Shan49] Shannon, C.E. "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, Vol.10 (1949), 656-715.

[Simm79] Simmons, G. "Symmetric and Asymmetric Encryption", *Computing Surveys*, Vol.11, No.4 (1979), 305-323.

[Wagn84] Wagner, N.R. "Searching For Public-Key Cryptosystems", IEEE Symposium on Network Security, (1984), 91-98.

APPENDIX A

GF(64)

This field is constructed with the irreducible polynomial $f(t) = \alpha^3 t^3 + \alpha$ as the modulus.

$0 = 0t^2 + 0t + 0$
$g = \alpha^3 t + \alpha^3$
$g^2 = \alpha^3 t^2 + \alpha^3$
$g^3 = \alpha^3 t^2 + \alpha^{3t} + \alpha^2$
$g^4 = \alpha t + \alpha^3$
$g^5 = \alpha t^2 + \alpha^2 t + \alpha^3$
$g^6 = \alpha^3 t^2 + \alpha t + \alpha$
$g^7 = \alpha^2 t^2$
$g^8 = \alpha^2 t^2 + \alpha^3$
$g^9 = \alpha^2 t^2 + \alpha^3 t$
$g^{10} = \alpha t^2 + \alpha^3 t + \alpha^3$
$g^{11} = \alpha^2 t^2 + \alpha$
$g^{12} = \alpha^2 t^2 + \alpha t + \alpha^2$
$g^{13} = \alpha^3 t^2 + \alpha^3 t + \alpha$
$g^{14} = \alpha^2 t$
$g^{15} = \alpha^2 t^2 + \alpha^2 t$
$g^{16} = \alpha^2 t + \alpha^3$
$g^{17} = \alpha^2 t^2 + \alpha t + \alpha^3$
$g^{18} = \alpha^3 t^2 + \alpha^2 t$
$g^{19} = \alpha t^2 + \alpha^2 t + \alpha$
$g^{20} = \alpha^3 t^2 + \alpha^3 t + \alpha^3$
$g^{21} = \alpha^2$
$g^{22} = \alpha^2 t + \alpha^2$
$g^{23} = \alpha^2 t^2 + \alpha^2$
$g^{24} = \alpha^2 t^2 + \alpha^2 t + \alpha$
$g^{25} = \alpha^3 t + \alpha^2$
$g^{26} = \alpha^3 t^2 + \alpha t + \alpha^2$
$g^{27} = \alpha^2 t^2 + \alpha^3 t + \alpha^3$
$g^{28} = \alpha t^2$
$g^{29} = \alpha t^2 + \alpha^2$
$g^{30} = \alpha t^2 + \alpha^2 t$
$g^{31} = \alpha^3 t^2 + \alpha^2 t + \alpha^2$
$g^{32} = \alpha t^2 + \alpha^3$
$g^{33} = \alpha t^2 + \alpha^3 t + \alpha$
$g^{34} = \alpha^2 t^2 + \alpha^2 t + \alpha^3$
$g^{35} = \alpha t$
$g^{36} = \alpha t^2 + \alpha t$
$g^{37} = \alpha t + \alpha^2$
$g^{38} = \alpha t^2 + \alpha^3 t + \alpha^2$
$g^{39} = \alpha^2 t^2 + \alpha t$
$g^{40} = \alpha^3 t^2 + \alpha t + \alpha^3$
$g^{41} = \alpha^2 t^2 + \alpha^2 t + \alpha^2$
$g^{42} = \alpha$
$g^{43} = \alpha t + \alpha$
$g^{44} = \alpha t^2 + \alpha$
$g^{45} = \alpha t^2 + \alpha t + \alpha^3$
$g^{46} = \alpha^2 t + \alpha$
$g^{47} = \alpha^2 t^2 + \alpha^3 t + \alpha$
$g^{48} = \alpha t^2 + \alpha^2 t + \alpha^2$
$g^{49} = \alpha^3 t^2$
$g^{50} = \alpha^3 t^2 + \alpha$
$g^{51} = \alpha^3 t^2 + \alpha t$
$g^{52} = \alpha^2 t^2 + \alpha t + \alpha$
$g^{53} = \alpha^3 t^2 + \alpha^2$
$g^{54} = \alpha^3 t^2 + \alpha^2 t + \alpha^3$
$g^{55} = \alpha t^2 + \alpha t + \alpha^2$
$g^{56} = \alpha^3 t$
$g^{57} = \alpha^3 t^2 + \alpha^3 t$
$g^{58} = \alpha^3 t + \alpha$
$g^{59} = \alpha^3 t^2 + \alpha^2 t + \alpha$
$g^{60} = \alpha t^2 + \alpha^3 t$
$g^{61} = \alpha^2 t^2 + \alpha^3 t + \alpha^2$
$g^{62} = \alpha t^2 + \alpha t + \alpha$
$g^{63} = \alpha^3$

APPENDIX B

Example Chor-Rivest Knapsack

For convinience the numbering of the steps involved in this example corresponds to Chor's thesis [Chor85].

a. System Generation

1. Let p be a prime power, $h \leq p$ an integer such that discrete logarithms in $GF(p^h)$ can be efficiently computed. Here we choose $p = 4$, and $h = 3$.

2,3. In these steps, the base field $GF(p)$ and its h-degree extension are found. These operations are described in § 1.3.1 above, with the resulting field $GF(p^h)$ given in Appendix A.

4. Compute $a_i = \log_g(t+i)$ for $i = 0, \alpha, \alpha^2, \cdots, \alpha^{p-1}$

$$a_0 = \log_g(t+0) = 56$$

$$a_1 = \log_g(t+\alpha) = 58$$

$$a_2 = \log_g(t+\alpha^2) = 25$$

$$a_3 = \log_g(t+\alpha^3) = 1$$

5. Scramble the a_i's giving $b_i = a_{\pi(i)}$ where π is a random permutation of $0, 1, 2, \cdots, p-1$. Let $\pi = (2, 3, 1, 0)$ yielding:

$$b_0 = a_{\pi(0)} = a_2 = 25$$

$$b_1 = a_{\pi(1)} = a_3 = 1$$

$$b_2 = a_{\pi(2)} = a_1 = 58$$

$$b_3 = a_{\pi(3)} = a_0 = 56$$

6. Add noise to the b_i's giving $c_i = b_i + d$, where d is a random number $0 \leq d \leq p^h - 1$. Let $d = 52$ yielding:

$$c_0 = b_0 + 52 = 77$$

$$c_1 = b_1 + 52 = 53$$

$$c_2 = b_2 + 52 = 110$$

$$c_3 = b_3 + 52 = 108$$

7. Public key:

$$\{c = (77, 53, 110, 108);\ p = 4;\ h = 3\}$$

8. Private key:

$$\{f(t) = t^3 + \alpha;\ g = t + 1;\ \pi = (0, 3, 1, 2);\ d = 52\}$$

b. Encryption

Encrypt a message M which is a binary vector of length p containing exactly h ones as follows:

$$E(M) = \sum_{i=0}^{i=p-1} c_i \cdot M_i \ (mod\ p^h - 1)$$

Thus if we wish to send the message $M = 0111$, we find

$$E(M) = (0 \cdot 77) + (1 \cdot 53) + (1 \cdot 110) + (1 \cdot 108) = 271 = 19\ (mod\ 63)$$

c. Decryption

1. Compute $r(t) = t^h\ (mod\ f(t)) = \alpha$.

2. Compute $s = E(M) - hd\ (mod\ p^h - 1) = 19 - (3)(52)\ (mod\ 63) = 52$.

3. Find $q(t) = g^s (mod\ f(t)) = g^{52} = \alpha^2 t^2 + \alpha t + \alpha$.

4. Find $s(t) = t^h - r(t) + q(t) = t^3 + \alpha^2 t^2 + \alpha t$.

5. By (at most p) successive substitutions we can factor $d(t)$:

$$s(t) = (t + 0) \cdot (t + \alpha) \cdot (t + \alpha^3)$$

This corresponds to the pattern 1101, to which π^{-1} is applied to recover the original message $M = 0111$.

APPENDIX C

Makefile for the System

```
#
#       Makefile for Chor-Rivest Cryptosystem
#
BELL            = "G"

SRCS            = Makefile types.h global.h mp.h global.c\
                  poly.c utility.c error.c\
                  gf.c chinese.c logarithm.c\
                  generate.c encrypt.c decrypt.c

GEN             = global.o poly.o error.o utility.o gf.o mp.o\
                  chinese.o logarithm.o generate.o

ENC             = global.o error.o mp.o encrypt.o

DEC             = global.o poly.o error.o mp.o decrypt.o

FAC             = factor.o mp.o

#
#       Targets for the make command
#
see:
                  cat $(SRCS) | more

list:
                  cat $(SRCS) | pr -h CHOR_RIVEST > _temp
                  cat _temp | lpr
                  rm _temp

generate:         $(GEN)
                  cc -O -o generate $(GEN) -lm
                  @echo $(BELL)

encrypt:          $(ENC)
                  cc -O -o encrypt $(ENC) -lm
                  @echo $(BELL)

decrypt:          $(DEC)
                  cc -O -o decrypt $(DEC) -lm
                  @echo $(BELL)
```

```
factor:         $(FAC)
                cc -O -o factor $(FAC) -lm

#
#       Dependencies of object files on header files
#
chinese.o:      types.h mp.h
decrypt.o:      types.h global.h mp.h
encrypt.o:      types.h global.h mp.h
factor.o:       mp.h
generate.o:     types.h global.h mp.h
gf.o:           types.h global.h mp.h
global.o:       types.h mp.h
logarithm.o:    types.h global.h mp.h
mp.o:           mp.h
poly.o:         types.h global.h mp.h
```

APPENDIX D

Users Manual for the Chor-Rivest Cryptosystem

1. Background

This document describes the commands comprising a *public key cryptosystem* first proposed by MIT cryptographers Ben Zion Chor and Ronald Rivest [Chor84]. Throughout the remainder of this manual we refer to this cryptosystem as the *Chor-Rivest system.* Derived from the knapsack problem, this system differs from earlier knapsack public key systems in that computations to create the knapsack are done in finite algebraic structures, called *Galois fields.* An interesting result from Bose and Chowla supplies a method of constructing higher densities than previously attainable [Bose62]. Not only does an increased information rate arise, but the new system so far is immune to the low density attacks levied against its predecessors, notably those of Radziszowski-Kreher and Lagarias-Odlyzko [Radz86, Laga85].

Normally, a given Chor-Rivest community is serviced by an installation which has fixed the parameters p and h. The programs described herein can mimic most anticipated installations by permitting the user to specify these as input. In so doing, the cryptanalyst user is afforded means to produce data requisite in breaking a particular installation or the cryptosystem itself.

2. Review of the Cryptosystem

It is not the purpose of this document to describe the underlying workings of the Chor-Rivest system: This information is presented elsewhere, and is not essential in order to use the system (see the references, § 5, for more information). However, if one

intends to exploit the flexibility provided by the command modifying options, a more detailed understanding of the system is required. In this section we give only a superficial sketch of the algorithms, primarily to indicate variable names.

To generate a matched pair of keys for encryption and decryption the user first selects a prime power p and a positive integer $h < p$. These parameters ultimately define the message space associated with the keys: Messages will consist of p bits, exactly h of which must be 1's and the other $p-h$ will be 0's. Thus, the size of the message space is $\binom{p}{h}$.

The system (more precisely, the key generation program **generate**) proceeds by defining a Galois field $GF(p)$ and an h-degree extension $GF(p^h)$ of it. This is done by finding (at random) a polynomial $f(x)$ such that $GF(p) = \mathbb{Z}_p[x]/(f(x))$, and α a multiplicative generator of $GF(p)$. The field extension is defined by randomly selecting $f(t)$ a degree h polynomial, irreducible over $GF(p)$, and a g a multiplicative generator of $GF(p^h)$. The next phase of the key generation requires discrete logarithm calculations (to the base g over $GF(p^h)$. From a computational standpoint, this is a very difficult problem. As a result, certain constraints, detailed in figure 1, are imposed on the choice of p and h. From certain logarithms, a p element knapsack is formed, then further disguised by applying a permutation π and adding a noise factor d. The resulting knapsack $\vec{c}$, along with p and h, comprise the *public key* (also called the encryption key). The

---

(1) $p < 256$ must be a *prime power*.

(2) $h \leq 25$ is a positive integer strictly less than p.

(3) The *prime factorization* of $p^h - 1$ may consist of at most 32 distinct primes, the largest less than 10^8.

Figure 1. Constraints on the choice of system parameters p and h.

private (or decryption) *key* consists of $f(t)$, g, π, d and arithmetic tables over $GF(p)$.

3. Use of the Cryptosystem

Three executable modules are provided:

a) **generate**, produces a public/private key pair;
b) **encrypt**, encrypts messages utilizing the public key;
c) **decrypt**, decrypts messages with the private key;

To run these, one enters the command name followed by arguments to specify options or data for the program. The usual conventions are used to specify syntax in the following sections:

a) optional arguments are enclosed in brackets [ ],
b) arguments are processed in order (left to right),
c) ellipses ". . ." indicate an arbitrary number of the argument type.

3.1. Specifying Options

A few options are provided to alter the behavior of the basic commands. Options are preceded by a minus "-" sign, and may be placed anywhere on the command line. Note however, that in the case of *encrypt* or *decrypt* commands, the options will take effect only for the arguments following the option. So for instance, the command

```
command -x file1 -y file2
```

will process file1 with only the "x" option enabled, while file2 will be handled under the influence of both "x" and "y". When several options are desired they may be lumped together as a string with a single minus sign prefix, so

```
command -x -y arguments . . .
```

has the same effect as

```
command -xy arguments . . .
```

The available options are **c**, **d**, **r**, **v** for clock, dump, random and verbose respectively. The clock option will time various portions of the *generate* command or the time to translate each message in the other commands. The verbose option will reveal important intermediate results to the **stdout**. The dump option is similar, but only applies to results of the logarithm calculations, performed by **generate**. The random option, described below, actually disables random generation of variables, and the user is prompted for several additional inputs.

3.2. The Generate Command

The **generate** command produces keys needed encrypt and decrypt messages so **generate** must be run prior to **encrypt** and **decrypt**. The public and private keys that result are placed into files **pub_ppp_hh** and **pri_ppp_hh** respectively, where ppp = p on three digits and hh = h on two digits. The format of these keys is explained later (§ 4).

The syntax of the **generate** command is

generate [-c] [-v] [-r] [-d] -f file p h

In its most basic form, the parameters p and h must be specified along with a file containing the factorization of p^h-1. This file consists of integers separated by whitespace (i.e. blanks, tabs or newline characters), where the first entry is the number of factors and successive entries are a prime and the corresponding power. For example, if $p = 4$ and $h = 3$ then

$$p^h-1 = 63 = 3^2 \cdot 7^1.$$

This information is ordered in an arbitrary file as "2 3 2 7 1" or "2 7 1 3 2". An auxiliary program **factor** prompts the user for p and h and outputs the factorization of p^h-1. If the output is redirected into a file, it can be easily edited to conform to the above requirements. Constraints on the choice of p and h are detailed in figure 1.

Normally, the system randomly selects possibilities for the special polynomials $f(x)$, α, $f(t)$ and g until suitable candidates are found; the noise factor d is also a random number (§ 2). The user can override this random approach by specifying the **-r** option (any where on the command line) to the **generate** program. In this case he is prompted for these values as they are required. Figure 2 illustrates the features of this option. There are several points of interest in this session. First, notice that the system has prompted for the same information in steps (1) and (3), and in steps (2) and (4): This indicates that the first set of coefficients supplied for $f(x)$ and α were unacceptable. In steps (1) and (2) we tried the polynomials $f(x) = x^2 + x + 1$ and $\alpha = 1$. Clearly

---

(0) **generate 4 3 -f factors_of_63 -r**
(1) Enter coefficients of f(x), modulus of GF(p) (degree 2)
  term 2: **1**
  term 1: **1**
  term 0: **1**
(2) Enter coefficients of generator of GF(p) (degree 1)
  term 1: **0**
  term 0: **1**
(3) Enter coefficients of f(x), modulus of GF(p) (degree 2)
  term 2: **1**
  term 1: **1**
  term 0: **1**
(3) Enter coefficients of generator of GF(p) (degree 1)
  term 1: **1**
  term 0: **1**
(4) Enter coefficients of f(t), modulus of GF(p^h) (degree 3)
  term 3: **3**
  term 2: **0**
  term 1: **0**
  term 0: **1**
(5) Enter coefficients of g, generator of GF(p^h) (degree 2)
  term 2: **3**
  term 1: **0**
  term 0: **1**
(6) Enter noise factor d. Enter size and decimal number **1 40**

Figure 2. Execution with **-r** option (user inputs in bold).

a constant polynomial will not prove to be a multiplicative generator of the field $GF(p)$, therein lies the reason for the repeated requests (3) and (4). The reason that we are prompted for both $f(x)$ and α is that the system tries these as a pair: Upon failure, the system has no way of knowing which polynomial was in error. By reiterating the choice of $f(x)$ and changing to $\alpha = x + 1$ the system is satisfied and moves on to the next prompt.

The next phase is to define an extension of $GF(p)$, namely $GF(p^h)$ by finding an irreducible h-degree polynomial $f(t)$ and a generator g in steps (5) and (6). Here we are specifying coefficients as powers of the generator α of $GF(p)$, or 0. So our entries of **(3,0,0,1)** for the coefficients of $f(t)$ really indicate the coefficients $(\alpha^3, 0, 0, \alpha^1)$. The same is true of g, the generator, so we have successfully specified the field $GF(4^3)$ by the polynomials

$$f(t) = a^3t^3 + \alpha = t^3 + \alpha,$$

$$g = \alpha^3t^2 + \alpha = t^2 + \alpha.$$

Recall that in $GF(p)$, $\alpha^{p-1} \equiv 1$ for p (a prime power). Also note that both $f(x)$ and $f(t)$ must be monic (leading coefficient of 1): Because in $GF(p^h)$ we are entering powers of α, the leading coefficient of $f(t)$ must be set to $p-1$ to yield a monic polynomial.

In step (6) the system prompts for d the noise factor to be added to each knapsack element. This value must be an positive integer, but can be quite large, so it must be entered piecemeal, in base 10^8. In this example the notation **1 40** means that there is one 10^8 digit, and this digit is 40 yielding $d = 40$. If on the other hand we had wanted to set d to 111,112,222,222,233,333,333 then the following entry would be in order:

3 11111 22222222 33333333.

Basically, just break the number into eight digit pieces, beginning with the ones digit, and prefix this with the number of pieces.

3.3. The Encrypt Command

The command **encrypt** uses the public key to transform an arbitrary number of plaintext messages, one per file, into ciphertext. The output is written to the **stdout**, with a newline character following each ciphertext message. If this output is instead redirected into **a** separate file, then it is ready to use as an argument to the program **decrypt**.

There is a very rigid format for each plaintext message: Only one message per file is permitted. Furthermore, the message must consist of exactly h 1's and $p-h$ 0's, however, whitespace may be inserted as desired to enhance readability.

The syntax is

encrypt [-c] public_key [-c] file1 [-c] file2 . . .

where file*n* are plaintext files having the format just described. Clocking takes effect for each file to the right of the **-c** on the command line. Once started, there is no way to disable the clock, although it will time each of the message transformations individually. The output from this option is also to the **stdout**. If the output is redirected to a file, the timing information must be removed (using an editor) prior to decryption.

3.4. The Decrypt Command

The command **decrypt** will decrypt an arbitrary number of messages produced by **encrypt**, using the private key.

The syntax is

decrypt [-c] [-v] private_key [-c] [-v] file1 [-c] [-v] file2 . . .

where each file*n* contains an arbitrary number of ciphertext messages, one per line. The same rules apply for the options **-v** and **-c** as for the **-c** option to **encrypt**. That is, the options take effect on subsequent files, and cannot be disabled once enabled. Here

again, the output is to the **stdout**, one plaintext translation per line.

4. Communication Between Modules

The generate command produces two text files that can be inspected or renamed. One file is the public key file originally named **pub_ppp_hh**, which must be the first (non-option) argument to **encrypt**. The other is the private key file originally named **pri_ppp_hh**, which must be the first (non-option) argument to **decrypt**.

4.1. The Public Key

The public key is really a knapsack vector $\vec{c}$: There are p elements, for which any h-fold sum (modulo p^h-1) is unique.

The file containing the public key contains the parameters p h and the knapsack vector $\vec{c}$. This file has $p+1$ lines, p and h occupy the first line separated by a space. An element c_i of $\vec{c}$ is found on each successive line.

The c_i are very large numbers, represented internal to the system as arrays of 2^{30} digits, with a header indicating how many of these digits are used to hold the number. In the public key file, the header and the 2^{30} digits are separated by a space,

$$c_i[0] \quad c_i[c_i[0]] \quad c_i[c_i[0]-1] \quad \cdots \quad c_i[2] \quad c_i[1],$$

where $c_i[1]$ is the least significant digit.

4.2. The Private Key

The private key, composed of $f(t)$, g, π, d and arithmetic tables over $GF(p)$, is supplied as the first argument to **decrypt**. The polynomials $f(t)$ and g are represented by a series of integers in the range $0 \cdots p-1$. These denote the power of α of each coefficient, or zero for a missing term. In the key file the first line has $h+1$ entries one for each coefficient of the h-degree polynomial $f(t)$. Similarly, the second line has h entries for g. The third line of the public key file contains the permutation π such that $i \rightarrow \pi_i$

for $0 \leq i \leq p-1$. The noise factor d occupies the fourth line using the same notation as the c_i of the public key (§ 4.1). The $p \times p$ addition and multiplication tables occupy the next $2p$ lines of the file with p values on each line.

5. References

The references provided here offer first hand information concerning public key systems in general, the Chor-Rivest system in particular and the theory of Galois fields which are central to the application.

[Chor84] Chor, B. and R.L. Rivest "A Knapsack Type Public Key Cryptosystem Based On Arithmetic in Finite Fields", *Advances in Cryptology: Proceedings of CRYPTO 84*, Berlin: Springer-Verlag, (1985), 54-65.

[Chor85] Chor, B. "Two Issues in Public-Key Cryptography - RSA Bit Security and a New Knapsack Type System," doctoral thesis, MIT (1985).

[Diff76] Diffie, W. and M. Hellman "New Directions in Cryptography", *IEEE Transactions on Information Theory*, Vol.22, No.6 (1976), 644-654.

[Merk82] Merkle, R.C. *Secrecy, Authentication, and Public Key Systems*, Ann Arbor, Michigan: UMI Research Press, 1982.

[Pohl78] Pohlig, R.C. and M. Hellman "An Improved Algorithm for Computing Logarithms Over $GF(P)$ and its Cryptographic Significance", *IEEE Transactions on Information Theory*, Vol.24 (1978), 106-110.

[Sala86] Salamone, R.T., Jr. *An Implementation of the Chor-Rivest Knapsack Based Cryptosystem*, Master of Science Thesis: Rochester Institute of Technology, 1986.

[Shan49] Shannon, C.E. "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, Vol.10 (1949), 656-715.