Project Number: WJM-5100

COMBINATORIAL STRUCTURES IN CRYPTOGRAPHY

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Seth Hardy

Date: April 26, 2002

Approved:

Professor William J. Martin, Advisor

1. **combinatorics**
2. **cryptography**
3. **authentication**

Abstract

Error correcting codes, such as Reed-Solomon codes, can be used to create authentication codes based on orthogonal arrays. These codes are provably secure up to a certain number of uses; however, as the number of desired uses goes up, so does the keylength. This project researches the security of a code whose messages (which function as private keys) have specific form that allows them to be represented in a more compact fashion. Specifically, messages with low Hamming weight are considered.

Contents

1 Introduction **1**
1.1 Project Description . . . 1
1.2 Authentication . . . 2
1.3 Overview . . . 3

2 Background **5**
2.1 Error Correcting Codes . . . 5
2.1.1 Introduction . . . 5
2.1.2 Definitions . . . 7
2.1.3 Linear Codes . . . 7
2.1.4 Cyclic Codes . . . 9
2.1.5 Reed-Solomon Codes . . . 10
2.2 Authentication Codes From EC Codes . . . 11
2.2.1 Orthogonal Arrays . . . 11
2.2.2 Authentication Codes . . . 12
2.2.3 Orthogonal Arrays from EC Codes . . . 12
2.2.4 Orthogonal Arrays as Authentication Codes . . . 15

3 Research **17**
3.1 Introduction . . . 17
3.2 Overview of the Scheme . . . 18
3.3 Analysis of a Simple Case . . . 19
3.4 ϵ-Biased Arrays . . . 21
3.5 Counting Methods . . . 23
3.5.1 Simple Lower Bound . . . 24
3.5.2 Overcounting Approximations . . . 25
3.5.3 Counting Method Difficulties . . . 25
3.6 Average Case Analysis . . . 25

3.6.1 Derivation of the Plunge Estimate 26
3.6.2 Sample Parameter Values by the Plunge Estimate . . . 28
3.7 Computational Security 29

4 Conclusions **31**
4.1 Summary 31
4.2 Future Work 32

A1 Maple Worksheets **33**
A1.1 mqp1.mws 33
A1.2 mqp2.mws 34
A1.3 mqp3.mws 36
A1.4 mqp4.mws 37

A2 Source Code **40**
A2.1 genkey.c 40
A2.2 rs-auth.c 41

List of Tables

2.1 Obtaining an authentication rule $e_k : \mathcal{S} \to \mathcal{A}$ from the rows of an array . . . 13

2.2 Example of an orthogonal array: $OA_1(4, 4, 2)$ . . . 15

3.1 Authentication Scheme Parameters . . . 18

3.2 Probability distribution of $OA_1(4, 4, 2)$ . . . 21

3.3 $OA_1(4, 4, 2)$ with 4 errors . . . 22

3.4 Probability distribution of $OA_1(4, 4, 2)$ with 4 errors . . . 22

3.5 Average number of uses t for parameters k, ℓ, and $q = 2^{128}$ . . . 28

3.6 Average number of uses t for parameters q, ℓ, and $k = 3000$ . . . 29

List of Figures

1.1 Authentication Model 2

2.1 Example of a Noisy Channel: the Binary Symmetric Channel 6
2.2 Model For Error Correcting Code Use 6

Chapter 1

Introduction

1.1 Project Description

With the recent increase in mobile communications and commerce, the need for security on mobile computing platforms has become evident. Traditional cryptographic systems, such as the RSA, Diffie-Hellman, or elliptic curve algorithms, require a great deal of computing power and often are not feasible to implement on a mobile device such as a smartcard or Personal Digital Assistant (PDA). A system that requires less computing power and has a smaller footprint is very desirable for this sort of application.

This project looks at authentication codes derived from error-correcting codes, specifically Reed-Solomon codes. This type of authentication code is chosen for the low computing power that it requires, as arithmetic is done over finite fields that are easy to implement in software and in hardware. The arithmetic takes considerably less computing power than that of the RSA, Diffie-Hellman, or elliptic curve cryptosystems.

The size of keys in a Reed-Solomon based authentication system, however, is dependent on the number of desired uses of the system before a new key is chosen. A goal of the project is to investigate ways to reduce the key size of the system, without reducing the security of the system to the point where it is unusable. This should be done by finding some mathematical properties that would allow for the key size to be reduced while still maintaining some mathematically verifiable level of security. Restricting the possible values of keys to a smaller space is an option, but may weaken the system. So, with any improvement to the key size of the system, one must provide some

lower bound on security. The system is said to be *unconditionally secure* if a random guess of the key from a large number of candidate keys is the best way an impersonator has of determining the key. An alternative to unconditional security is computational security. The system is *computationally secure* if an adversary would need more computing power than is currently available to succeed in his task of determining the key.

1.2 Authentication

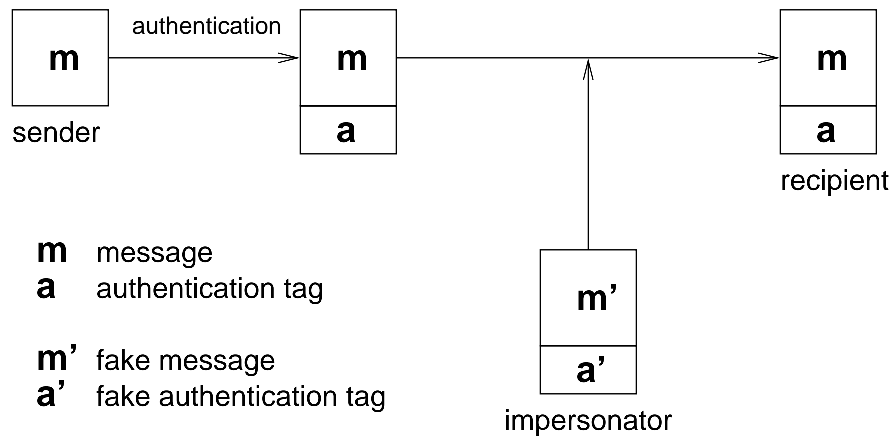

Figure 1.1: Authentication Model

Authentication is not a form of encryption or secrecy. It is a way of verifying the sender of a message, and ensuring that the message received is correct. For example, Alice may want to authenticate her financial transactions that she makes on her PDA. For each message m, Alice would use her private key k to come up with an *authentication tag* a for the message. Instead of just sending her message, Alice would then send the pair (m, a) to her bank. The bank would receive this pair, and first make sure that they could verify the authentication tag a on the message m using Alice's key. If the authentication tags match up, then the message must have been authenticated with Alice's key, and since the keys are private the message is assumed to have come from Alice, and has not been tampered with en route.

An impersonator watching the transactions would be able to observe all of the message and authentication tag pairs that the user is transmitting to

her bank. By Kerckhoff's assumption, this impersonator would also have full access to all details of the authentication system except for the user's key. The goal of an impersonator is to create a message-authentication tag pair (m', a') such that a' verifies as a valid authentication tag for m' using the unknown key.

We would like to force the impersonator to make a random choice among some large number of keys. The level of uncertainty as to which key is actually Alice's should not be conditional on the impersonator's computing resources. This is the idea of unconditional security.

1.3 Overview

During the course of the project, the work went in a number of different directions. Roughly the first half of the project was spent learning the background material necessary. A good portion of this time was focused on coding theory, including the details of linear codes, cyclic codes, BCH codes, and eventually Reed-Solomon codes.

From there, the background research moved to the topic of orthogonal arrays, including the properties that allow them to function well as authentication codes, and the limitations that prevent them from being used in practical situations. Using error correcting codes to create orthogonal arrays, and then orthogonal arrays to create authentication codes, tied all of the subjects together. Here, the choice to look at an authentication code created from a Reed-Solomon subcode consisting of message polynomials with low Hamming weight was made.

The second half of the project was spent working on methods for determining the security of this authentication code. The first step was to look at simple cases for particular parameters, and see if any general observations could be made. ϵ-biased arrays were also considered, but due the amount of mathematical background needed for the analysis, it was decided that it was not the best route to take toward solving the problem. After this, work began on the counting methods to approximate the number of possible keys after t authentications. The counting methods took up a good amount of time, and eventually it was decided that the methods being used were not the most effective way (given the time span of the project) to solve the problem, either.

After the counting methods, the project shifted focus to an average case analysis. This proved to be more successful than the other methods, and an

approximation for the number of secure authentications (which we call the Plunge Estimate) was determined, along with a surprising observation on how the alphabet size changes this number of authentications.

The project ended with some observations on the similarity of the problem we consider to another existing problem that is known to be NP-complete. Whether or not the problem that we look at in this project is NP-complete is unknown, but it is possible that the known hard problem may be extended in a way that would show our problem to be hard as well.

Chapter 2

Background

2.1 Error Correcting Codes

2.1.1 Introduction

Error correcting codes are used to encode information so that errors in the encoded information (such as those introduced by sending information over a noisy channel) can be detected and corrected. This would allow the recipient to reconstruct the original message, even if errors did occur during transmission.

A noisy channel is one that may introduce errors to information transmitted over it. For example, a channel may be able to transmit a bit at a time, and has a probability p of flipping the bit during transmission. Generally, $0 \leq p \leq \frac{1}{2}$. If $p > \frac{1}{2}$, then the recipient would be able to flip the received bit and have an error probability $p' \leq \frac{1}{2}$. This example channel, known as the binary symmetric channel, is shown in Figure 2.1.

A sender would convert a message into a codeword, and then send the codeword instead of the original message. This codeword would need to have the property of being easily distinguished as a codeword, so that the recipient can check its validity. The recipient would take the received information, and determine whether any errors had occurred by performing this check to see if the received information is a codeword. If the received data is not a codeword, it would be converted back into the valid codeword (assuming that there were not too many errors to correct), and then the original message data could be recovered from the codeword.

Let the message $u = u_1 u_2 \cdots u_k, u_i \in \{0, 1\}$ be a vector of k elements

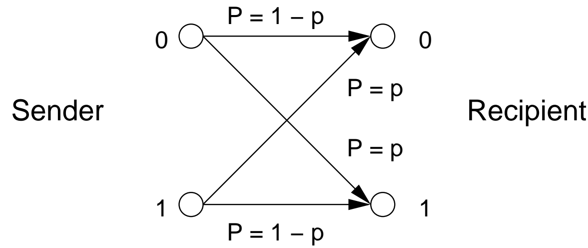

Figure 2.1: Example of a Noisy Channel: the Binary Symmetric Channel

from $\{0, 1\}$. These information symbols are then encoded to the vector $x = x_1x_2\cdots x_n, x_i \in \{0, 1\}$. This is the codeword x which will be sent over the channel. Now, let $e = e_1 \cdots e_n$ be an error vector, representing the errors introduced during transmission. In the above example, where p is the probability of an error in transmission, then $e_i = 0$ with probability $1 - p$, which means that $y_i = x_i$. We have $e_i = 1$ with probability p, meaning that $y_i \neq x_i$.

The recipient receives the vector $y = x + e$, which is then checked to see if it is a valid codeword. If the error vector $e = 0$, then $y = x$ and y is a valid codeword. Otherwise, it may or may not be. The decoder must then find the "nearest" codeword x' to y, and decode this codeword to the vector u'.

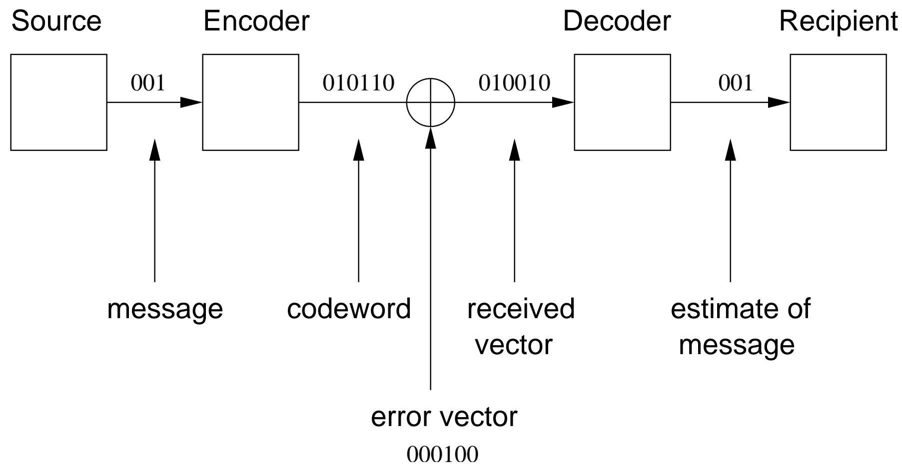

Figure 2.2: Model For Error Correcting Code Use

A code that encodes k message symbols to codewords of length n is said to have *dimension* k and *length* n. Because the code expands k message symbols to a codeword of length n, it is also convenient to define the *rate* or efficiency of a code, $R = \frac{k}{n}$. This is the fraction of channel bits that contain information. The rest of the bits are overhead added by the error correcting code.

It is advantageous to generalize beyond $\{0, 1\}$ for the information symbols that the message and codeword consist of. Instead of restricting ourselves to $\{0, 1\}$, we use the Galois Field $GF(q)$, where $q = p^m$ for some prime p.

2.1.2 Definitions

To further understand the properties of error correcting codes, we will need to define some terms. The first counts the number of nonzero entries in a vector:

Definition. The *Hamming weight* of a vector $x = x_1 \cdots x_n$ is the number of nonzero x_i. The Hamming weight of a vector is denoted $\text{wt}(x)$.

Having notation for the number of nonequal entries in two vectors is also useful, because it lets us consider the minimum distance of a code.

Definition. The *Hamming distance* between two vectors $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ is the number of i such that $x_i \neq y_i$. The Hamming distance between two vectors is denoted $\text{dist}(x, y)$.

The definition of Hamming distance comes directly from the definition of Hamming weight, as $\text{dist}(x, y) = \text{wt}(x - y, 0)$. This leads to the definition of the minimum distance of a code:

Definition. The *minimum distance* of a code is the minimum Hamming distance between any two distinct codewords.

2.1.3 Linear Codes

Van Lint gives the following definition in [9] for linear codes:

Definition. A linear code $\mathcal{C}$ is a linear subspace of $GF(q)^n$, where $q = p^m$,

p prime. If $\mathcal{C}$ has dimension k then $\mathcal{C}$ is called an $[n, k]$ code.

The minimum distance of a code $\mathcal{C}$ is equal to the minimum nonzero weight of a codeword. Because adding any two codewords will give another valid codeword ($\mathcal{C}$ is a linear subspace), the codeword 0 must be in the code, and we have: $\text{dist}(x, y) = \text{dist}(x - y, 0) = \text{wt}(x - y)$. We note that the Hamming distance is clearly invariant under translations.

A code with minimum distance d can correct up to $\frac{1}{2}(d-1)$ errors. Conceptually, this is a very simple process: if the distance between each codeword is at least d, then with up to $\frac{1}{2}(d-1)$ errors, it is possible to search all of the codewords and find the "nearest" one (smallest distance). This nearest codeword will be unique, by the construction of the code. However, this is infeasible in practice.

A linear code with length n, dimension k, and minimum distance d is denoted as an $[n, k, d]$ code. If d is unknown, or unimportant in our discussion of the code, we omit it.

Linear codes can also be defined in terms of a parity check matrix and a generator matrix:

Let H be a $m \times n$ matrix where $H_{ij} \in GF(q)$ for $0 < i \leq m, 0 < j \leq m$. Then, the linear code with parity check matrix H consists of all vectors x such that $Hx^T = 0$. This shows why linear codes are in fact linear: if x, y are codewords of a code with a parity check matrix H, then $H(x+y)^T = Hx^T + Hy^T = 0 + 0 = 0$, and $H(\beta x)^T = \beta(Hx^T) = \beta 0 = 0$.

As an example of a linear code defined this way, let $q = 2$, and have the parity check matrix

$$H = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Because $Hx^T = 0$, we have

$$\begin{aligned} x_2 + x_3 + x_4 + x_5 &= 0 \\ x_1 + x_3 + x_4 + x_6 &= 0 \\ x_1 + x_2 + x_4 + x_7 &= 0 \end{aligned}$$

for any codeword x.

This gives $q^k = 16$ possible codewords:

0000000	1111111	0110011	1001100
0011001	0101010	0010110	1011010
0100101	1100110	1000011	1010101
1101001	0111100	0001111	1110000

The parity check matrix H often has the form

$$H = [A|I_{n-k}]$$

where A is some matrix over $GF(q)$ and I_s is the $s \times s$ identity matrix. This is not a requirement, but H must have $n-k$ linearly independent rows.

The generator matrix of the code is denoted G. It is related to H in that $GH^T = 0$ and $H^T G = 0$. By this relation, we have one possible choice for G:

$$G = [I_k| - A^T]$$

Again, denoting the vector of information symbols to be encoded by $u = u_1 u_2 \cdots u_k$, we find that the valid codewords of the code are

$$(u_1 * \text{row } 1) + (u_2 * \text{row } 2) + \cdots + (u_k * \text{row } k)$$

. This means that the valid codewords of the code are linear combinations of the rows of G, which is why G is called the generator matrix.

2.1.4 Cyclic Codes

A code $\mathcal{C}$ is *cyclic* if it is linear, and if any cyclic shift of a codeword is also a codeword, i.e. if $c = (c_0, c_1, ..., c_{n-1})$ is a codeword, so is $c' = (c_1, c_2, ..., c_{n-1}, c_0)$, $c'' = (c_2, c_3, ..., c_0, c_1)$, etc.

We can look at these codewords as vectors (as above), or instead as polynomials. The mapping from vector to polynomial is very simple:

$$(c_0, c_1, ..., c_n) \to c_0 + c_1 x + \cdots + c_n x^n$$

If F is a field, and $F[x] = \{a_0 + a_1 x + +a_2 x^2 + \cdots : a_i \in F\}$ is the set of polynomials with coefficients from F, then $\mathcal{C}$ is an ideal of the ring $R_n = F[x]/(x^n - 1)$. That is, for any codeword $c(x) \in \mathcal{C}$ and polynomial $f(x) \in R_n$, $cf \in \mathcal{C}$, and $xc(x) \in \mathcal{C}$. Multiplication by x in R_n is equivalent to a cyclic shift: because the arithmetic is being done modulo $x^n - 1$, we have

$x^n = 1$, and $xc(x) = c_0x + c_1x^2 + \cdots + c_{n-1}x^n = c_0x + c_1x^2 + \cdots + c_{n-1} = c_{n-1} + c_0x + c_1x^2 + \cdots + c_{n-2}x^{n-1}$.

Because we are working over the ring R_n, every ideal is a principal ideal, which means that any ideal I consists of all multiples of a fixed polynomial $g(x)$. MacWilliams and Sloane give the following proof in [5] that the code $\mathcal{C}$ is generated by a single polynomial $g(x)$. The polynomial $g(x)$ is minimal degree in $\mathcal{C}$, and is unique: if we suppose $g(x)$ and $g'(x)$ are distinct minimal degree polynomials, then $g(x) - g'(x)$ is also in the code (because it is linear), and has a lower degree than $g(x)$ or $g'(x)$, which is a contradiction. Therefore, $g(x) = g'(x)$, and so, $g(x)$ is unique. A codeword $c(x) \in \mathcal{C}$ can be expressed as $c(x) = q(x)g(x) + r(x)$. Rewriting this, we have $r(x) = c(x) - q(x)g(x)$, which means that $r(x) \in \mathcal{C}$ because we are subtracting one codeword from another. Since $\deg(r)$ is less than the minimal degree, we have $r(x) = 0$, and so, $c(x)$ is a multiple of $g(x)$. This polynomial $g(x)$ is the *generator polynomial* of the ideal, and the principal ideal generated by $g(x)$ (the code $\mathcal{C}$) is denoted by $\langle g(x) \rangle$.

Let $u = u_0u_1 \cdots u_{k-1}, u_i \in GF(q)$ be the k information symbols of a message. To encode the message, convert it into a polynomial $f(x) = u_0 + u_1x + u_2x^2 + \cdots + u_{k-1}x^{k-1}$, and then multiply it by the generator polynomial:

$$c(x) = f(x)g(x)$$

2.1.5 Reed-Solomon Codes

Reed-Solomon codes are nothing new; almost everyone has seen some implementation of them, even if he or she is completely unaware of their existence. Compact Discs are one popular use of the codes, providing error-correction of the digital music and preventing skips even if the disc is scratched or otherwise unreadable. For more information on the uses of Reed-Solomon codes, including the details of how they are used on Compact Discs, see [6].

There are two general methods of constructing Reed-Solomon codes. The first is the original approach of Reed and Solomon, and the second views Reed-Solomon codes as any other cyclic codes.

Original Approach

Let $u = u_0u_1 \cdots u_{k-1}, u_i \in GF(q)$ be the k information symbols of a message. To construct a Reed-Solomon code, we take this message and we convert it

into a polynomial $f(x) = u_0 + u_1 x + u_2 x^2 + \cdots + u_{k-1} x^{k-1}$. The Reed-Solomon codeword for this message is the polynomial $f(x)$ evaluated at every element of $GF(q)$.

This means that the Reed-Solomon code of length $n = q$ and dimension k is

$$\mathcal{R} = \{c_f = [f(0), f(\alpha), f(\alpha^2), \ldots, f(\alpha^{q-1})] : \text{ deg } (f) < k\}$$

where α is a primitive element of $GF(q)$, so $0, \alpha, \alpha^2, \ldots, \alpha^{q-1}$ are the q elements of $GF(q)$.

Cyclic Code Approach

Reed-Solomon codes are cyclic codes, which means that they can be generated by a generator polynomial as described in the section on cyclic codes. Encoding a message u into a codeword c is then done by multiplying the polynomial representation of the message by the generator polynomial.

The generator polynomial must have $2v$ consecutive powers of α as roots, where v is the desired number of errors to correct:

$$\prod_{j=0}^{2v-1} (x - \alpha^{b+j})$$

The minimum distance of the code $d = 2v + 1$.

The Reed-Solomon codes generated in this manner have length $n = q - 1$. The codes may be extended to longer lengths (such as length $n = q$ through the original generation approach), but are no longer cyclic.

Detecting and correcting errors uses a more complicated method that is not used in the scope of this project. For more information on Reed-Solomon codes, and error correcting codes in general, can be found in [5] and [10].

2.2 Authentication Codes From EC Codes

2.2.1 Orthogonal Arrays

An *orthogonal array* is a combinatorial structure that has long been used in statistics for the purpose of experiment design. When used for the design of an experiment, the rows of the orthogonal array represent the experiments to be performed, and the columns represent the variables which are being

analyzed. The symbols in the array correspond to varying levels at which the variables are to be applied in the experiment.

Orthogonal arrays are beneficial because they provide a way to test all possible combinations of the different variables on the experiments with the least number of experiments needed. This is very useful when testing to see whether any of the variables interact.

The properties of orthogonal arrays that make them good for use in statistics can also be used here, for construction of authentication codes.

First, we must give a definition for orthogonal arrays:

Definition. An orthogonal array $OA_\lambda(t, k, v)$ is a $\lambda v^t \times k$ array of v symbols, such that in any two columns of the array every one of the possible v^t t-tuples of symbols occurs in exactly λ rows.

Using orthogonal arrays, we are able to construct authentication codes. But first, we need a definition of what an authentication code actually is.

2.2.2 Authentication Codes

Authentication codes are described in detail in [7]. Stinson defines an authentication code in four parts: a finite set of "source states" $\mathcal{S}$, a finite set of "authentication tags" $\mathcal{A}$, a finite set of keys $\mathcal{K}$, and for each $K \in \mathcal{K}$ an "authentication rule" $e_k : \mathcal{S} \to \mathcal{A}$. The source states correspond to the possible messages, and the set of authentication tags corresponds to the alphabet that the authentication code is using.

As an example, let $\mathcal{S} = GF(2^2)$, $\mathcal{A} = GF(2)$, $\mathcal{K} = GF(2^2) \times GF(2^2)$, and let e_k be defined as in Table 2.1.

To determine what the authentication tag for a given key and source state is, find the entry in the table where the row corresponds to the key being used, and the column corresponds to the source state (message) being authenticated. For example, a user with key $(\alpha, 1)$ would authenticate message β with authentication tag 1, and message 1 with authentication tag 0.

2.2.3 Orthogonal Arrays from EC Codes

When looking at the way to construct an orthogonal array from an error correcting code, we need the definition of a dual code:

	k	0	1	α	β
	$(0, 0)$	0	0	0	0
	$(0, 1)$	0	0	0	1
	$(0, \alpha)$	0	0	1	0
	$(0, \beta)$	0	0	1	1
	$(1, 0)$	0	1	0	0
	$(1, 1)$	0	1	0	1
	$(1, \alpha)$	0	1	1	0
	$(1, \beta)$	0	1	1	1
	$(\alpha, 0)$	1	0	0	0
$\rightarrow$	$(\alpha, 1)$	1	0	0	1
	(α, α)	1	0	1	0
	(α, β)	1	0	1	1
	$(\beta, 0)$	1	1	0	0
	$(\beta, 1)$	1	1	0	1
	(β, α)	1	1	1	0
	(β, β)	1	1	1	1

Table 2.1: Obtaining an authentication rule $e_k : \mathcal{S} \rightarrow \mathcal{A}$ from the rows of an array

Definition. If $\mathcal{C}$ is an $[n,k]$ linear code over F, its *dual* or *orthogonal code* $\mathcal{C}^\perp$ is the set of vectors which are orthogonal to all codewords of $\mathcal{C}$:

$$\mathcal{C}^\perp = \{u | \forall v \in \mathcal{C}(u \cdot v = 0)\}$$

If H is the parity check matrix of a code, and G the generator matrix, then the parity check matrix of $\mathcal{C}^\perp$ is $H^\perp = G$, and the generator matrix of $\mathcal{C}^\perp$ is $G^\perp = H$.

We consider here the Reed-Solomon code $\mathcal{R}$ over $GF(q)$, H is the parity check matrix of $\mathcal{R}$, and $G^\perp = H$ is the generator matrix of $\mathcal{R}^\perp$. $\mathcal{R}^\perp$ has length n and dimension $n-k$. The rank r of H is $n-k$, so H has $n-k$ linearly independent columns.

Bose and Bush determined the following construction, which relates error correcting codes to orthogonal arrays:

Theorem. Suppose that M is an (R, K, q)-array of elements from the finite field $GF(q)$, such that any T columns of M are linearly independent (as vectors in $GF(q)^*$). Then there exists an $OA_{q^{R-T}}(T, K, q)$.

In the case of $M = H$, the number of rows is equal to the rank: $R = r = T = n - k$, and the number of columns of the matrix is equal to the length of the code: $K = n$. This means that, by this construction, the linear combinations of the rows of H (which are the codewords of $\mathcal{R}^\perp$) form an $OA_{q^{(n-k)-r}}(r, n, q) = OA_1(n-k, n, q)$.

If $T = R$, such as in the above case, we can instead choose M as

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & \alpha & \alpha^2 & \cdots & \alpha^{n-2} \\ 0 & 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(n-2)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \alpha^{k-1} & \alpha^{2(k-1)} & \cdots & \alpha^{(n-2)(k-1)} \end{pmatrix}$$

This matrix M is actually the generator matrix G for $\mathcal{R}$:

$$\begin{pmatrix} u_0 & u_1 & u_2 & \cdots & u_{k-1} \end{pmatrix} \cdot M = \begin{pmatrix} f(0) & f(1) & f(\alpha) & \cdots & f(\alpha^{n-2}) \end{pmatrix}$$

This shows that taking all of the codewords of $\mathcal{R}$ yields an $OA_1(k, n, q)$.

For using Reed-Solomon codes to generate orthogonal arrays, the original approach of Reed and Solomon for creating their codes works best. If we look

back to Figure 2.1, we see that the codewords of the code form the rows of the array, and the codeword of each row corresponds to one of the q^k different message polynomials.

If each user has a message polynomial as his or her key, then by the original definition of the code by Reed and Solomon, to generate a particular entry of the codeword the user must evaluate the message polynomial at a single element of $GF(q)$. This means that only one polynomial evaluation is needed to generate an authentication tag for a message, a very efficient method.

2.2.4 Orthogonal Arrays as Authentication Codes

The authentication rule e_k as defined in Table 2.1 in the example of a simple authentication code is actually an orthogonal array, as shown in Table 2.2.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Table 2.2: Example of an orthogonal array: $OA_1(4, 4, 2)$

Stinson gives an important theorem in [7], which relates orthogonal arrays to authentication codes. Using our notation, this theorem is:

Theorem. Suppose there is an orthogonal array $OA_\lambda(t, k, v)$. Then there is

an authentication code $(\mathcal{S}, \mathcal{A}, \mathcal{K}, \mathcal{E})$ where $|\mathcal{S}| = k$, $|\mathcal{A}| = v$, $|\mathcal{K}| = \lambda t^v$, and $Pd_0 = Pd_1 = 1/v$.

The value Pd_0 here is the probability of impersonation by an attacker: the probability that an attacker could introduce a valid combination (s, a), where $s \in \mathcal{S}$ and $a \in \mathcal{A}$, into the channel. Pd_1 is the probability of a valid substitution by an attacker: the probability that an attacker could replace an observed (s, a) with another message $(s', a'), s' \in \mathcal{S}, a' \in \mathcal{A}$ and have the new message be accepted as authentic.

The fact that $Pd_0 = Pd_1 = 1/v$ means that the system has perfect (unconditional) secrecy: that is, the probability that the key is f given observation of an authentication tag g is equal to the probability that the key is f.

What this theorem means is that orthogonal arrays can be used as authentication codes. The rows correspond to the keys, and the columns correspond to the source states. The authentication tags are the elements of the alphabet that appear in the orthogonal array, and the encoding function for a given key and source state corresponds to finding the authentication tag that is in the row corresponding to the key and the column corresponding to the source state.

Referring back to the previous example, where a user with key $(\alpha, 1)$ authenticates message β with authentication tag 1, and message 1 with authentication tag 0, we see that an impersonator has no advantage when trying to authenticate another message as the user. Even with full knowledge of the array, the impersonator only has a $\frac{1}{2} = \frac{1}{|\mathcal{A}|}$ chance of guessing the correct authentication tag due to the properties of the OA. This holds true as long as the number of uses of the authentication code is less than or equal to the strength of the OA.

In a practical application, the source states are hashes of messages, and the alphabet used is $GF(2^m)$ where m equals the bit length of the hash. This means that the hash values can be represented as elements of $GF(2^m)$. The keys are message polynomials of the error correcting code, and the encryption functions correspond to evaluating the message polynomial with x equal to the message hash.

Chapter 3

Research

3.1 Introduction

Orthogonal arrays give perfect secrecy when the number of authentications t is less than or equal to the strength of the array. For a large number of desired authentications, the strength of the OA must be large, which means that the OA itself will be very large. A number of different bounds exist on how large the OA must be, including the Rao bound, the Bush bound, the Bierbrauer-Friedman bound, and linear programming bounds. Here we look at the Rao bound:

Theorem (Rao, 1947). If an $OA_\lambda(t, k, v)$ exists, then

$$\begin{aligned} \lambda v^t &\geq 1 + \sum_{i=0}^{t/2} \binom{k}{i} (v-1)^i \text{ (t even)} \\ \lambda v^t &\geq 1 + \sum_{i=0}^{u} \binom{k}{i} (v-1)^i + \binom{k-1}{u} (v-1)^{u+1} \text{ (t odd)} \end{aligned}$$

with $u = \frac{t-1}{2}$.

When v is small, λ needs to be sufficiently large that the above conditions are met. However, when v is large, then v^t is also large and the bound is not as strong for the value of λ required. In the case we are looking at, orthogonal arrays created from Reed-Solomon codes, v is large and $\lambda = 1$.

For a particular user, we are only concerned with one row in the OA: the codeword created from the single message (by evaluation of the message polynomial) which corresponds to the user's secret key. This message can be very large itself, having t coefficients over the alphabet $GF(q)$. If $q = 2^m$, then the number of bits needed for the key is $t * m$. For codes that will be used many times, the value of $t * m$ can get considerably large.

To reduce the size of the secret key, we look for message polynomials of a special form that can be represented in less space, but still has unconditional security.

3.2 Overview of the Scheme

To construct the authentication code, we look at the subcode of a Reed-Solomon code consisting of message polynomials of low Hamming weight (number of nonzero coefficients): polynomials of maximum degree k with at most ℓ nonzero coefficients.

$\mathcal{R}$ is the complete Reed-Solomon code:

$$\mathcal{R} = \{c_f = [f(0), f(\alpha), f(\alpha^2), \ldots, f(\alpha^{q-1})] : \text{ deg } (f) < k\}$$

$\mathcal{C}$ is the chosen subcode:

$$\mathcal{C} = \{c_f \in \mathcal{R} : wt(f) \leq \ell\}$$

Table 3.1 describes the parameters for this authentication scheme.

Parameter	Description
k	Maximum degree of message polynomial
ℓ	Maximum Hamming weight of message polynomial
q	Alphabet size
t	Desired number of authentications

Table 3.1: Authentication Scheme Parameters

3.3 Analysis of a Simple Case

As a starting point for looking at the problem, we can look at a very simple case, where $q = 2^m$, $m \geq 1$, $k = 2$, and $\ell = 2$.

Theorem 3.3.1 *Let $q = 2^m$, $m \geq 1$, and let $\mathcal{C}$ denote the subcode of the Reed-Solomon code consisting of the codewords of messages corresponding to polynomials of degree at most two, having at most two nonzero coefficients:*

$$\mathcal{C} = \{c_f \in \mathcal{R} | f = rx^2 + sx + t, rst = 0\}$$

Given any two distinct coordinates i and j, and any α and β, there is a unique codeword c with

$$c_i = f(i) = \alpha \qquad \text{and} \qquad c_j = f(j) = \beta$$

iff:

1. $i = 0$, $\alpha = \beta \neq 0$
2. $i, j \neq 0$, $\alpha = \beta = 0$

Proof For all i, j, α, β there is at least one such message in $\mathcal{C}$, as the codewords of $\mathcal{C}$ generated by message polynomials of degree of 0 or 1 form an orthogonal array of strength 2. By examining the possible cases for the given i and j, we can show which message polynomials $f(x)$ are unique:

1. Assume $i = 0$, $j \neq 0$.

   If $\alpha = \beta = 0$, let $f(x) = x(x - j)$. This f gives a second codeword meeting the criterion that $f(i) = \alpha = 0$ and $f(j) = \beta = 0$.

   Otherwise, when $\alpha \neq \beta$, we seek $f(x) = rx^2 + t$, with $f(0) = \alpha$ and $f(j) = \beta$. Since $i = 0$, $t = \alpha$.

   So, $f(j) = rj^2 + \alpha = \beta$, and $r = \frac{\beta - \alpha}{j^2}$. So, $f(x) = rx^2 + \alpha$ satisfies $f(0) = \alpha$ and $f(j) = \beta$.

   In the case where $\alpha = \beta \neq 0$, then $f(x) = rx^2 + sx + t$ satisfies $f(j) = rj^2 + sj + \alpha$. Letting $f(j) = \alpha$ and $s = 0$, we find that since $\alpha = f(j) = rj^2 + sj + \alpha$, we have $0 = j(rj + s)$, which gives $r = -\frac{s}{j}$. Since $s = 0$, $r = 0$ as well.

   However, this is polynomial of degree 0, so it was already counted above. Therefore, it is unique.

2. Assume $i \neq 0$, $j \neq 0$, $i \neq j$.

   If $\alpha \neq \beta$, we express $f(x) = rx^2 + t$, and we have $ri^2 + t = \alpha$ and $rj^2 + t = \beta$. Adding the two equations together, we get $r(i^2 + j^2) = \tau$, where $\tau \neq 0$ because $\alpha \neq \beta$.

   So, $r = \frac{\tau}{i^2+j^2}$. r always exists as $i^2 + j^2 \neq 0$, because $i \neq j$. Now we can take $r = \frac{\tau}{i^2+j^2}$, such that $f(x) = rx^2 + sx$ satisfies $f(i) = \alpha$ and $f(j) = \beta$.

   If $\alpha = \beta$, $f(x) = rx^2 + t$ or $f(x) = rx^2 + sx$.

   In the case of $f(x) = rx^2 + t$, then we have $ri^2 + t = rj^2 + t = \alpha$.

   So, $r(i^2 + j^2) = 0$, which means that either $r = 0$ or $i^2 + j^2 = 0$, as $GF(q)$ has no zero divisors.

   But $i^2 + j^2 = 0$ means that $i^2 = j^2$, which means $i = j$. Therefore, $r = 0$. This means that this polynomial is of degree 1, so it was already counted above, making it unique.

   In the case of $f(x) = rx^2 + sx$, we have $ri^2 + \alpha = si$ and $rj^2 + \beta = sj$.

   So, $\frac{ri^2+\alpha}{i} = \frac{rj^2+\beta}{j} = s$.

   Thus $j(ri^2 + \alpha) = i(rj^2 + \beta)$, which can be simplified in terms of r as $r = \frac{\alpha j+\beta i}{ij(i+j)}$.

   In the case of $\alpha = \beta = 0$, $r = 0$, the polynomial is of degree 1, so it was already counted above. Therefore, it is unique.

   Otherwise, we can take $r = \frac{\alpha j+\beta i}{ij(i+j)}$ such that $f(x) = rx^2 + sx$ satisfies $f(i) = \alpha$ and $f(j) = \beta$. r is non-zero, because $\alpha j + \beta i$ is non-zero: $i, j \neq 0$ and $\alpha j \neq \beta i$ because $i \neq j$ and $\alpha \neq \beta$ or $\alpha = \beta \neq 0$. r always exists, as $i, j \neq 0$, $ij(i + j) \neq 0$ as well.

As an example, over $GF(2^2) = \{0, 1, \alpha, \beta\}$, if a user authenticates message 1 as $f(1) = \beta$ and message α as $f(\alpha) = 0$, then the possible polynomials f are

$$f(x) = \alpha x^2 + 1 \qquad \text{or} \qquad f(x) = x^2 + \alpha x$$

and there is no way to tell (other than testing each) which is actually the private key. For larger parameters, the number of possible f will be larger.

Another interesting case is when $\ell = 1$. Then, we have

$$f(x) = cx^i$$

for some c and i. An impersonator who observes messages and authentication tags $(h_1, ch_1^i), (h_2, ch_2^i), \ldots, (h_t, ch_t^i)$ and wants to find the private key used for the authentications must solve a problem very similar to the discrete logarithm problem (for more details, see [6]). The only difference between this problem and the discrete logarithm problem is that each authentication tag is multiplied by some constant c.

3.4 ϵ-Biased Arrays

By definition of an orthogonal array, the appearance each t-tuple in any set of t columns is equiprobable. For example, looking at the $OA_1(4,4,2)$ from Table 2.2, we get the following probabilities of choosing two values a, b from columns i, j.

$ij \setminus ab$	00	01	10	11
12	1/4	1/4	1/4	1/4
13	1/4	1/4	1/4	1/4
14	1/4	1/4	1/4	1/4
23	1/4	1/4	1/4	1/4
24	1/4	1/4	1/4	1/4
34	1/4	1/4	1/4	1/4

Table 3.2: Probability distribution of $OA_1(4,4,2)$

For arrays other than orthogonal arrays, the probability distribution is not so even. If the probability distribution is close to uniform, then the array is close to being an orthogonal array.

If we change the $OA_1(4,4,2)$ from Table 2.2 in a few places, we get the array described in Table 3.3. This array has the probability distribution shown in Table 3.4.

In all cases for this example array, the probability p of choosing values a, b from columns i, j is in between $\frac{3}{16} \leq p \leq \frac{6}{16}$.

In general, the probability p will satisfy

$$\frac{1}{q^t} - \epsilon \leq p \leq \frac{1}{q^t} + \epsilon$$

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

Table 3.3: $OA_1(4,4,2)$ with 4 errors

$ij \setminus ab$	00	01	10	11
12	1/4	3/16	5/16	1/4
13	1/4	3/16	5/16	1/4
14	3/16	1/4	1/4	5/16
23	5/16	1/4	1/4	3/16
24	3/16	6/16	1/4	3/16
34	3/16	6/16	1/4	3/16

Table 3.4: Probability distribution of $OA_1(4,4,2)$ with 4 errors

For large t, this is approximately

$$0 \leq p \leq \epsilon$$

We want ϵ to be very small: 2^{-m}, for some large m.

This leads into the idea of bias. Bierbrauer and Schellwat give the following definition for the bias of a vector in [1]:

Definition. Let p be a prime, $v = (v_1, v_2, \ldots, v_n) \in F_p^n$. For every $x \in F_p = Z/pZ$ let $\nu_v(x)$ be the number of times x appears as an entry of v and let $\delta_v(x)$ be defined by $\nu_v(x) = \frac{n}{p} + \delta_v(x)$. Let ζ be a primitive complex p-th root of unity. The **bias** of v is defined as

$$\text{bias}(v) = \frac{1}{n}|\sum_{x \in F_p} \delta_v(x)\zeta^x| = \frac{1}{n}|\sum_{x \in F_p} \nu_v(x)\zeta^x|$$

This leads to the definition of an ϵ-biased array:

Definition. Let $0 \leq \epsilon \leq 1$. An $n \times k$ array with entries from a set of q elements is *ϵ-biased* if every nontrivial linear combination of its columns has bias $\leq \epsilon$.

This is another direction to consider when looking at this problem. Instead of taking this route, we instead choose to look at counting methods and an average case analysis.

3.5 Counting Methods

Suppose Alice uses the scheme from Section 3.2. An impersonator would want to observe her message-authentication pairs, and use the pairs gathered over time to try to find Alice's private key. With the private key, the impersonator could authenticate as many messages as he wanted as Alice.

If the impersonator observes t message-authentication pairs $(h_1, a_1), (h_2, a_2), \ldots, (h_t, a_t)$, where $a_i = f(h_i)$, then he can still not figure out Alice's secret key f. He is able to determine all of the polynomials f' that would give the same authentications, however. These polynomials f' satisfy

$f' \equiv f \pmod{g}$, where

$$g = \prod_{i=1}^{t}(x - h_i)$$

To measure the security of the authentication scheme, we want to determine how many of these f' exist for each f. Because the impersonator has no way of determining which of the f' is Alice's key f, the scheme has unconditional security. If there are w possible f', then the probability that the impersonator will be able to successfully guess Alice's key is $1/w$. For large w, this means that the impersonator's chance of successfully being able to authenticate messages as coming from Alice is very small.

Because $f' \equiv f \pmod{g}$, we have $f - f' = gz$ for some $z \in F_q[x]$. For a fixed g, the solutions to this equation are

$$N = \{(f, f', z) : f - f' = gz\}$$

For a particular polynomial f, the possible polynomials f' are

$$N_f = \{(f', z) : f - f' = gz\}$$

To approximate $|N_f|$ (which is our w from above), we can count either f' or z. A simple approximation of $|N_f|$ is $|N|/|\mathcal{C}|$, which can be used if we approximate $|N|$.

3.5.1 Simple Lower Bound

Over $GF(2^m)$, $f - f' = gz$ is equivalent to $f + f' = gz$. So, we can partition the coefficients of gz as the coefficients of f and f', letting all of the other coefficients of f and f' equal 0.

Because $wt(f) \leq \ell$ and $wt(f') \leq \ell$, we have $wt(gz) \leq 2\ell$. We define $r = \deg(z)$, and remember that $\deg(g) = t$. This means that $\deg(z) \leq 2\ell - t$.

An approximation on the number of possible f' can be made by counting the number of partitions of gz into f and f' for each z:

$$N_1 = \sum_{r=1}^{2\ell-t}(k - t - r)q^{r-1}(q-1)^2\binom{t+r}{\ell} + \binom{t}{l}(k-t)(q-1)$$

This estimate does not approximate the number of possible f' that well, however. This is made obvious by taking the value $\frac{N_1}{|\mathcal{C}|}$: the average number

of possible f' per f is very close to 0. The reason for this is that there are many other possibilities for f and f' other than partitions of the coefficients of gz.

3.5.2 Overcounting Approximations

In this case, we allow $\deg(gz) < k$, and $wt(gz) \leq 2\ell$. Allowing for all possible choices of z, and choosing up to ℓ coefficients for f', we have:

$$N_2 = \sum_{r=0}^{k-t} \sum_{i=t+r-\ell}^{\ell} \binom{t+r}{i} (q-1)^{i+r+1}$$

This method overcounts: it allows f to be of degree greater than ℓ while counting f' satisfying $wt(f') \leq \ell$.

3.5.3 Counting Method Difficulties

If gz were dense, then a good bound on the number of f' for a given f would be possible. However, since gz is not necessarily dense, there is no way of knowing $wt(gz)$. The weight of gz needs to be known for this sort of counting method, because for each zero coefficient $(gz)_i$, we can let $f_i = f'_i = 0$ and increase the degree of f and f' by one, such that it is greater than ℓ.

3.6 Average Case Analysis

Since we can not find a good method to count the number of f' for each f such that $f' \equiv f \pmod{g}$, we can instead look at the average number of f' which satisfy this equation. Because we are looking at the average, we are not able to determine how many values of f, if any, are weak keys (i.e. have a small number of possible f').

Let $\mathcal{C} = \{c_f \in R : wt(f) \leq \ell\}$, where $wt(f)$ denotes the number of non-zero coefficients of f (the weight of f), be the subcode of R consisting of message polynomials of maximum weight ℓ.

Goal: Determine how many uses t the code $\mathcal{C}$ can be used for, such that the average number of possible authentication tags for a message is greater than 2^n for some n.

3.6.1 Derivation of the Plunge Estimate

Let $(h_1, a_1), (h_2, a_2), \ldots, (h_t, a_t)$ be given, where $a_i = f(h_i)$ is the authentication tag for the hash value h_i. An impersonator would want to use this information to determine f.

If a codeword f' gives the same set of outputs as f, then $f' \equiv f \pmod{g}$, where $g = \prod_{i=0}^{t}(x - h_i)$. For it to be infeasible for an impersonator to guess f, the number of possible f' for the specific f must be large (i.e. greater than 2^m).

The number of remainders modulo g is $|\text{Pol}(t-1, q)|$, i.e. those polynomials over $GF(q)$ with maximum degree $t-1$.

The average number of message polynomials in $\mathcal{C}$ with a given remainder is the number of message polynomials in $\mathcal{C}$ divided by the number of possible remainders modulo g:

$$\frac{|\mathcal{C}|}{|\text{Pol}(t-1, q)|} = \frac{\sum_{i=0}^{\ell} \binom{k}{i} (q-1)^i}{q^t}$$

$\text{Pol}(t-1, q) = \{f \in F_q[x] : \deg f < t\}$ is the set of polynomials with coefficients over $GF(q)$, with maximum degree $t-1$.

However, since the sequence $\{\binom{k}{i}(q-1)^i\}_i$ grows roughly as fast as q^i, we only need to consider the largest term of the sequence for the sum.

So,

$$\frac{|\mathcal{C}|}{|\text{Pol}(t-1, q)|} \approx \frac{\binom{k}{\ell}(q-1)^\ell}{q^t}$$

This approximation will only make this analysis more conservative, as the quantity on the left is greater than that on the right, i.e. if $\frac{\binom{k}{\ell}(q-1)^\ell}{q^t} > 2^m$, then $\frac{|\mathcal{C}|}{|\text{Pol}(t-1,q)|} > 2^m$ is as well.

We look for the largest value of t such that $\frac{|\mathcal{C}|}{|\text{Pol}(t-1,q)|} > 2^m$. Since this function of t shrinks at such a high rate, this is essentially the same as finding the largest value of t such that $\frac{|\mathcal{C}|}{|\text{Pol}(t-1,q)|} > 1$. This does introduce a slight error, but because increasing t by 1 means multiplying the number of possible polynomials by q, the approximation is off by at most 1 for $m \leq \log_2 q$. As the number of uses will be $\lfloor t \rfloor$, this approximation is reasonably safe.

Setting $\frac{|\mathcal{C}|}{|\text{Pol}(t-1,q)|} = 1$, we have $|\mathcal{C}| = |\text{Pol}(t-1, q)|$ and so,

$$\binom{k}{\ell}(q-1)^\ell \approx q^t$$

By definition of the binomial coefficient, this is equivalent to

$$q^t \approx \frac{k!}{(k-\ell)!\ell!}(q-1)^\ell$$

Using Stirling's approximation, this becomes

$$q^t \approx \frac{\sqrt{(2k+\frac{1}{3})\pi}k^k}{\sqrt{(2k-2l+\frac{1}{3})(2l-\frac{1}{3})\pi^2}(k-\ell)^{k-\ell}\ell^\ell}(q-1)^\ell$$

And so,

$$\begin{aligned} t &\approx \frac{1}{2}\log_q(2k+\frac{1}{3})+\frac{1}{2}\log_q\pi+k\log_q k+\ell\log_q(q-1)-\frac{1}{2}\log_q(2k-2l+\frac{1}{3}) \\ &- \frac{1}{2}\log_q(2l-\frac{1}{3})-\log_q\pi-(k-\ell)\log_q(k-\ell)-\ell\log_q\ell \end{aligned}$$

With sufficiently large q, $\log_q\pi \approx 0$, $\log_q 2 \approx 0$, and $\log_q(q-1) \approx 1$. If we define

$$\begin{aligned} \theta_1 &= \frac{2k+\frac{1}{3}}{k} \\ \theta_2 &= \frac{2k-2\ell+\frac{1}{3}}{k-l} \\ \theta_3 &= \frac{2\ell-\frac{1}{3}}{\ell} \end{aligned}$$

the equation becomes

$$\begin{aligned} t &\approx \frac{1}{2}\log_q k+\frac{1}{2}\log_q\theta_1+\frac{1}{2}\log_q\pi+k\log_q k+\ell+\ell\log_q(q-1)-\frac{1}{2}\log_q(k-\ell) \\ &- \frac{1}{2}\log_q\theta_2+\frac{1}{2}\log_q\ell+\frac{1}{2}\log_q\theta_3-\log_q\pi-(k-\ell)\log_q(k-\ell)-\ell\log_q\ell \end{aligned}$$

with $2 \leq \theta_1, \theta_2, \theta_3 \leq 3$.

If we drop the small terms,

$$t \approx \ell+(\frac{1}{2}+k)\log_q k-(\ell+\frac{1}{2})\log_q\ell-(k-\ell+\frac{1}{2})\log_q(k-\ell)$$

The error ϵ of this approximation, with $q = 2^{20}$ is between $-0.0309 \leq \epsilon \leq 0.0130$. As q increases, the bounds of the error become smaller, as $\log_q \theta_i$ and $\log_q \pi$ decrease. At $q = 2^{128}$, $-0.0048 \leq \epsilon \leq 0.0020$.

If we write $\ell = \alpha k$, then $k - \ell = (1-\alpha)k$. We have a good approximation on the average number of authentications t for which this system is unconditionally secure. We call this approximation the Plunge Estimate, named for the fact that $\frac{|C|}{|\mathrm{Pol}(t-1,q)|}$ decreases so rapidly.

Plunge Estimate:

$$t \approx \ell - \log_q \sqrt{k} + (\ell + \frac{1}{2}) \log_q(\frac{1}{\alpha}) + (k - \ell + \frac{1}{2}) \log_q(\frac{1}{1-\alpha})$$

The Plunge Estimate gives a bound for the highest t such that, on average, there are still very many low weight polynomials f', $c_{f'} \in \mathcal{C}$ with the same remainder as $f \pmod g$.

3.6.2 Sample Parameter Values by the Plunge Estimate

To observe how k changes the number of authentications t for different maximum weights ℓ, we look at some sample values:

ℓ	k = 1000	k = 3000	k = 5000
10	10.618	10.742	10.800
25	26.300	26.612	26.756
50	52.216	52.844	53.134
75	77.979	78.929	79.365
100	103.639	104.916	105.499

Table 3.5: Average number of uses t for parameters k, ℓ, and $q = 2^{128}$

The number of uses t for a given k and ℓ is the floor of the above value, as t must be an integer. Because the above values are for when $\frac{|\mathcal{C}|}{|\mathrm{Pol}(t-1,q)|} = 1$, taking a more conservative estimate and subtracting 1 from the listed value for t will ensure that $\frac{|\mathcal{C}|}{|\mathrm{Pol}(t-1,q)|} \geq q$, which may be more reasonable despite the fast growth of q^i.

Because the average number of message polynomials with a given remainder is $\frac{|\mathcal{C}|}{|\text{Pol}_{(t-1,q)}|}$, decreasing the value of q decreases the number of possible remainders modulo g, and so causes the average number of message polynomials with any given remainder to increase. Decreasing the value of q would then allow for more uses of a specific key, while not harming the security of the system for as long as it was not decreased past a certain point. For example:

ℓ	q = 2^{64}	q = 2^{128}	q = 2^{256}
10	11.484	10.742	10.371
25	28.223	26.612	25.806
50	55.689	52.844	51.422
75	82.858	78.929	76.964
100	109.832	104.916	102.458

Table 3.6: Average number of uses t for parameters q, ℓ, and $k = 3000$

3.7 Computational Security

We want this authentication scheme to have unconditional security. Even if the scheme is not unconditionally secure, we can still comment on the computational power that would required to break it. Here we look at the level of computational security of the authentication scheme.

If we set $r_i(x) \equiv x^i \pmod{g(x)}$, then

$$f(x) \pmod{g(x)} = \sum_{i=0}^{q-1} a_i r_i(x)$$

If we write this in matrix form, we have:

$$\begin{pmatrix} \\ r_0(x) & \Big| & r_1(x) & \Big| & \cdots & \Big| & r_k(x) \\ \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{q-1} \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_t \end{pmatrix}$$

Our problem is to find the coefficients u_i of f after seeing t authentications, knowing that f has at most ℓ non-zero coefficients and that $t > \ell$. This

problem is similar to the following problem:

Minimum Weight Solution to Linear Equations:

Given a $t \times q$ integer matrix A, an integer vector b of length t, and an integer $\ell \leq q$, is there a rational vector x of length q such that $Ax = b$ and x has at most ℓ non-zero entries?

The Minimum Weight Solution to Linear Equations problem is NP-complete, by transformation from a known NP-complete problem, Exact Cover by 3-Sets. For more information, see [2].

If the Minimum Weight Solution to Linear Equations problem can be extended to work over any field, not just the rationals, then it may be possible to show that our problem NP-complete. If this is the case, then the authentication scheme would be considered computationally secure. However, this proof is beyond the scope of the project.

Chapter 4

Conclusions

4.1 Summary

In this project we look at using an orthogonal array created from a Reed-Solomon code as an authentication code. Authentication codes which use orthogonal arrays as the authentication rules have perfect (unconditional) security up to a number of uses equal to the strength of the orthogonal array. Any array of this sort can be used as an authentication code: the rows correspond to the private keys, the columns correspond to the messages, the symbols in the array correspond to the authentication tags, and the authentication rule for a key and message is the symbol in the row and column for the key and message.

Because the private keys of this system correspond to the index of a row in the orthogonal array, the size of the key is dependent on the size of the array. The size of the orthogonal array is dependent on the strength of the array: the number of rows is λv^t, where t is the strength of the array, v is the number of symbols in the array, and λ is the number of times each t-tuples of symbols is repeated. Although in the cases we look at, $\lambda = 1$, the number of rows is still very large.

The goal of the project was to reduce the key size by choosing a subcode of the original Reed-Solomon code to use as the authentication rules for the authentication code. The array that this subcode creates would not be orthogonal, so the security of the authentication scheme would be reduced. For this to be useful, a proof of a bound on the security of the code was required.

The project considered the subcode consisting of the Reed-Solomon codewords for message polynomials of low Hamming weight. A number of different methods of evaluating the security of this system were considered. The average case analysis looked at how many possible polynomials, on average, were possibly the secret key after t observed messages and authentications. Using this method, parameters could be chosen such that a secret key of Hamming weight ℓ, and could be used for t authentications, with $t > \ell$.

One surprising observation while looking at these parameters is that by reducing the alphabet size q, the number of possible secure authentications actually increases. This is because the approximation of the number of possible message polynomials f' for a user's private key is divided by the total number of message polynomials in the subcode. The total number of keys is q^t, which with an increase of t, grows faster than the number of possible f'. By setting q to a smaller value, q^t does not grow as fast, so larger values for t are possible.

4.2 Future Work

The next step in this project would be to continue with the counting methods, to determine a better bound for the number of possible polynomials f' such that $f' \equiv f \pmod{g}$, i.e. the possible message polynomials that could be the private key given a set of t messages and authentications. The average case analysis does not guarantee the large number of f' for each each possible message polynomial f, and does not show whether there are any "weak keys" in the set of message polynomials.

Finding the best subcode of the Reed-Solomon code to use as the authentication rules is another possible topic for future work. This project only considered the subcode using message polynomials of low Hamming weight. There may be other subcodes that can be represented compactly, which create better matrices to use for the authentication rules.

Appendix A1

Maple Worksheets

A1.1 mqp1.mws

```
l := 749;
t := 750;
k := 750;
q := 2^128;


with(combinat):

##  J := proc(r)
##  local  alp, gam, eps;
##
##       RETURN(
##         add(
##           add(
##             add(
##               binomial(k-t-r, eps) *
##               multinomial(t+r+1,   alp, gam, t+r+1-alp-gam) *
##               (q-1)^( t+r+1-alp-gam + eps ) ,
##             gam = (t+r+1-l+eps)..(t+r+1-alp) ),
##           alp = (t+r+1-l+eps)..(t+r+1) ),
##         eps = 0..(k-t-r)  )
```

```
##       );
##  end;
##
J := proc(r)  RETURN( binomial(t+r, l) ); end;


X :=  (k-t+1)*(k-t)*(l)*(l);
# Estimated number of iterations inside 4 loops
evalf(X/1000000);

Hbar := evalf( add(  q^r*(q-1)*J(r), r=0..k-t)/binomial(k,l)/(q-1)^l );
```

A1.2 mqp2.mws

```
> with(combinat):
> k := 15;
> l := 2;
> q := 2^4;

                              k := 15


                               l := 2


                              q := 16

> binomial(k,l)*(q-1)^l;

                               23625

> t := 3;

                               t := 3
```

```
> printlevel := 0:
> x := []:
> y := []:
> for p from 1 to t do
>   foo := rand() mod q:
>   while member(foo,x) do foo := rand() mod q: end do:
>   x := [op(x),foo]:
>   foo := rand() mod q:
>   while member(foo,y) do foo := rand() mod q: end do:
>   y := [op(y),foo]:
> end do;
> printlevel := 2:
> x;
> y;

                              [9, 1, 0]


                             [6, 11, 10]

> f := product( a - y[b], b=1..t):
> expand(f) mod q;

                          3      2
                         a  + 5 a  + 12 a + 12

> expand(f * (a-8)) mod q;

                         4       3      2
                        a  + 13 a  + 4 a  + 12 a

> # S stores entries in the form [[power1,coeff1],[power2,coeff2]]
> #T := []:
> #for t from 2 to 5 do
> S := []:
> for xi from 1 to k-1 do
>   xi;
>   for yi from 0 to xi-1 do
```

```
>     for i from 1 to q-1 do
>       for j from 1 to q-1 do
>         ii := 0:
>         for ti from 1 to t do
>           if (i*x[ti]^xi + j*x[ti]^yi mod q <> y[ti]) then ii := ii + 1:
>           end if;
>         end do;
>         if (ii = 0) then S := [op(S),[[xi,i],[yi,j]]]: end if;
>       end do;
>     end do;
>   end do;
> end do;
> #T := [op(T),S]:
> #print(S);
> print(t,nops(S));
> #end do;
```

```
                                3, 0
```

```
> nops(S);
```

```
                                105
```

```
> S[1];
```

```
                          [[1, 1], [0, 1]]
```

A1.3 mqp3.mws

```
> with(combinat):
Warning, the protected name Chi has been redefined and unprotected

> p := 101;
```

```
                              p := 101

> U := choose(30,5):
> nops(U);

                               142506

> x := 3;

                               x := 3

> printlevel := 0:
> ck := array(0..p-1):
> ckcmax := 2000:
> ckc := array(0..ckcmax):
> for i from 0 to p-1 do ck[i] := 0: end do:
> for i from 0 to ckcmax do ckc[i] := 0: end do:
> for i in U do
>   xx := 0:
>   for c from 1 to nops(i) do
>     xx := xx + x^(op(c,i)) mod p:
>   end do;
>   ck[xx] := ck[xx] + 1;
> end do;
> for i from 0 to p-1 do ckc[ck[i]] := ckc[ck[i]]+1: end do:
> print(ckc);
>
```

A1.4 mqp4.mws

```
> with(combinat):
Warning, the protected name Chi has been redefined and unprotected

> l := 30;
> t := 40;
```

```
> k := 100;
> q := 2^128;

                                l := 30

                                t := 40

                                k := 100

              q := 340282366920938463463374607431768211456

> C := 0:
> for i from 0 to l do
>   C := C + binomial(k+1,i)*(q-1)^i:
> end do:
> # this method is what we worked out at the meeting; it doesn't do so well.
> N1 := binomial(t,l)*(k-t)*(q-1):
> for r from 1 to 2*l-t do
>   N1 := N1 + (k-t-r)* q^(r-1)*(q-1)^2*binomial(t+r,l);
> end do:
> evalf(N1/C);
>

                                          -353
                              .1850794144 10

> # this method is overcounting due to possible zeros for (gz)_i.
> N := 0:
> for r from 0 to 2*l-t do
>   for i from t+r-l to l do
>     N := N + ( binomial(t+r,i)*(q-1)^i ) * ( (q-1)^(r+1) ):
>   end do:
> end do:
> evalf(N/C);
```

```
                                                801
                                 .4173317968 10

> # this method overcounts. similar to above.
> N2 := 0:
> for r from 0 to k-t do
>   for j from t+r-2*l to t+r-1 do
>     for i from 0 to max(l,t+r-j) do
>       N2 := N2 + binomial(t+r,j) * binomial(t+r-j,i) * (q-1)^(i+1):
>     end do;
>   end do;
> end do;
> evalf(N2/C);

                                                1198
                                 .1671846889 10

> N3 := q-1:
> for r from 1 to 2*l-t do
>   for i from t+r-l to l do
>     N3 := N3 + (q-1)*q^r*binomial(t+r,i):
>   end do:
> end do:
> evalf(N3/C);

                                                 30
                                  .9631469980 10

> evalf(sum(q^(2*l-t-ii)*binomial(t+ii,l),ii=0..2*l-t)/C);
```

Appendix A2

Source Code

This is a sample implementation of the authentication code using full-length keys.

genkey.c creates a random key: a polynomial of degree 4999, having coefficients over $GF(2^{128})$. This key is exactly $5000 * 128 = 640000$ bits.

rs-auth.c takes input from stdin or a filename from the command line. The program will hash the input using the MD5 algorithm (Ronald Rivest's implementation), and then calculate the authentication tag for the source state corresponding to the message hash.

A2.1 genkey.c

```
#include <stdio.h>
#include <stdlib.h>

#define K 5000
#define Q 128

int main(int argc, char **argv) {

  FILE *key;
  int i;
  unsigned char k;

  srand(time(NULL));
```

```
  key = fopen("rs-auth.key","w");

  for(i=0;i<K*Q/8;i++) {
    k = rand() % 256;
    fwrite(&k,1,1,key);
  }

  fclose(key);


  return 0;

}
```

A2.2 rs-auth.c

```
// authentication scheme from reed-solomon codes
// using the md5 message digest function
// all md5 code is from rivest's implementation, modified to work here

#include <stdio.h>
#include <time.h>
#include <string.h>
#include "global.h"
#include "md5.h"

#define MD 5
#define MD5_CTX MD5_CTX
#define MDInit MD5Init
#define MDUpdate MD5Update
#define MDFinal MD5Final

typedef unsigned char uc;

static void MDFile(char *, uc[16]);
static void MDFilter(uc[16]);
static void hexprint(uc[16]);
```

```
void gf_add(uc[16], uc[16], uc[16]);
void gf_mul(uc[16], uc[16], uc[16], uc[16]);


int main (int argc, char **argv) {

  int i;
  FILE *key;
  uc dig[16];
  uc coeff[16];
  uc temp[16];
  uc expt[16];
  uc at[16];
  uc irr[16];

  if( (key = fopen("rs-auth.key","r")) == NULL ) {
    fprintf(stderr,"error: couldn't find key.\n");
    return(1);
  }

  if (argc > 1)
     MDFile (argv[1], dig);
  else {
    printf("enter your message, followed by ^D:\n\n");
    MDFilter (dig);
  }

  printf("\n        message ID: ");
  hexprint(dig);
  printf("\n");

  for(i=0;i<16;i++) {
    expt[i] = dig[i];
    irr[i] = 0;
  }

  irr[0] = 67;

  fread(at,1,16,key);
```

```
  for(i=1;i<5000;i++) {

    fread(coeff,1,16,key);
    gf_mul(expt,coeff,temp,irr);
    gf_add(at,temp,at);

    gf_mul(expt,dig,expt,irr);

  }

  printf("authentication tag: ");
  hexprint(at);
  printf("\n");

  fclose(key);

  return (0);
}


/* Digests a file and prints the result.
 */
static void MDFile(char *filename, uc digest[16]) {

  FILE *file;
  MD5_CTX context;
  int len;
  unsigned char buffer[1024];

  if ((file = fopen(filename, "rb")) == NULL) {
    printf ("%s can't be opened\n", filename);
    exit(1);
  }
  else {
    MDInit (&context);
    while (len = fread (buffer, 1, 1024, file))
      MDUpdate (&context, buffer, len);
    MDFinal (digest, &context);

    fclose (file);
```

```
  }

}

/* Digests the standard input and prints the result.
 */
static void MDFilter (uc digest[16]) {

  MD5_CTX context;
  int len;
  unsigned char buffer[16];

  MDInit (&context);
  while (len = fread (buffer, 1, 16, stdin))
  MDUpdate (&context, buffer, len);
  MDFinal (digest, &context);

}

/* Prints a message digest in hexadecimal.
 */
static void hexprint(uc digest[16]) {

  unsigned int i;

  for (i = 0; i < 16; i++)
    printf ("%02x", digest[i]);

}

void gf_add(uc x[16], uc y[16], uc z[16]) {

  int i;

  for(i=0;i<16;i++) z[i] = x[i]^y[i];

  return;

}
```

```
void gf_mul(uc x[16], uc y[16], uc z[16], uc irr[16]) {

  int i,j,shift,junk,carry;
  uc temp[32];

  for(i=0;i<32;i++) temp[i] = 0;

  for(i=0;i<16;i++)
    for(j=0;j<16;j++) {
      junk = temp[i+j] + x[i]*y[i];
      carry = 0;
      while( junk >= 256 ) {
        carry++;
        junk -= 256;
      }
      temp[i+j] += junk;
      temp[i+j+1] += carry;
    }

  for(i=31;i>15;i--) {
    if(temp[i] != 0) {
      shift = i-16;
      temp[i] = 0;
      for(j=shift;j<16+shift;j++)
        temp[j] = temp[j] + irr[j-shift];
    }
  }

  for(i=0;i<16;i++) z[i] = temp[i];

  return;

}
```

Bibliography

[1] J. Bierbrauer and H. Schellwat. Weakly biased arrays, almost independent arrays and error-correcting codes. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **56** (2001) 33–46.

[2] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[3] A. S. Hedayat, N. J. A. Sloane, and John Stufken. *Orthogonal Arrays: Theory and Applications.* Springer-Verlag, 1999.

[4] R. Hill. *A First Course in Coding Theory.* Oxford University Press, 1986.

[5] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes.* North-Holland Publishing Company, 1977.

[6] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[7] D. R. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

[8] D. R. Stinson. The combinatorics of authentication and secrecy codes. *J. Cryptology* **2** (1990), no. 1, 23–49.

[9] J. H. van Lint. *Introduction to Coding Theory.* Springer-Verlag, 1982.

[10] S. Wicker and V. Bhargava. *Reed-Solomon Codes and Their Applications.* IEEE Press, 1994.