2015

A Plausibly Deniable Encryption Scheme for Personal Data Storage

Andrew Brockmann
Harvey Mudd College

A Plausibly Deniable Encryption Scheme for Personal Data Storage

Andrew Brockmann

Talithia D. Williams, Advisor

Arthur T. Benjamin, Reader

HARVEY MUDD COLLEGE

Department of Mathematics

May, 2015

Abstract

Even if an encryption algorithm is mathematically strong, humans inevitably make for a weak link in most security protocols. A sufficiently threatening adversary will typically be able to force people to reveal their encrypted data.

Methods of deniable encryption seek to mend this vulnerability by allowing for decryption to alternate data which is plausible but not sensitive. Existing schemes which allow for deniable encryption are best suited for use by parties who wish to communicate with one another. They are not, however, ideal for personal data storage.

This paper develops a plausibly-deniable encryption system for use with personal data storage, such as hard drive encryption. This is accomplished by narrowing the encryption algorithm's message space, allowing different plausible plaintexts to correspond to one another under different encryption keys.

Contents

List of Figures

List of Tables

Acknowledgments

I am grateful to both Talithia Williams, my thesis advisor, and Art Benjamin, my second reader.

I also extend special thanks to Vincent Fiorentini and Megan Shao, without whom my final code might never have worked.

Chapter 1

Introduction

Encryption is a mathematical means of achieving information security. A person's unprotected information, called the *plaintext*, is passed as a parameter to an encryption function. The function requires another input, called an encryption *key*, which can be viewed as a string or a number. Given these inputs, the encryption algorithm produces an output, referred to as the *ciphertext*. The corresponding decryption algorithm, when given a ciphertext and the appropriate key (which may be distinct from the encryption key), produces the original plaintext.

The ciphertext usually has length comparable to (or the same as) that of the plaintext. However, while the plaintext can typically be easily read and understood, the ciphertext will in general appear random or nonsensical. The protection offered by encryption is a result of the difficulty involved in deducing anything about the plaintext when only the ciphertext is readily available. A good encryption algorithm can be performed quickly and produces a ciphertext that yields no useful information on its own. An adversarial third party can usually recover the plaintext by attempting decryption with every possible decryption key—however, the number of possible keys for any good algorithm is large enough that brute force attacks of this sort are completely impractical.

On its own, strong encryption can thwart *passive* eavesdroppers who may attempt to access individuals' sensitive information without ever interacting with the individuals in question. If information is stored in encrypted form, to be decrypted only when its owner has need of it, then anyone else who attempts to access the information will be presented with an unhelpful ciphertext. However, *active* eavesdroppers still pose a threat. An active eavesdropper may threaten people to reveal their plaintext information under threat of force, blackmail, or some other type of coercion. If the threat is strong enough, an individual may have little choice but to reveal their information; decrypting a ciphertext with an incorrect key will, with overwhelming probability, yield junk data, letting the threatening eavesdropper know that the decryption key used was incorrect.

The existence of threatening adversaries motivates the creation of encryption schemes which allow for plausible deniability. An encryption scheme is called *plausibly deniable* if it enables a user to maintain the secrecy of their plaintext even when faced with a threatening adversary. This is usually accomplished by allowing users to decrypt their ciphertexts to multiple sensible plaintexts, at least one of which can be safely revealed if necessary. Thus, when confronted by a threatening adversary, a user of a plausibly deniable scheme can choose to reveal a non-sensitive plaintext, preserving the security of their sensitive information.

The existing literature on plausibly deniable encryption includes several working schemes. However, all but one of them are not well-suited to large, personal data storage, and the remaining scheme may fail in the face of especially persistent adversaries.

The one existing scheme that works well for personal data storage uses *private-key* encryption algorithms. Private-key algorithms are those which use the same key for both encryption and decryption. The encryption and decryption algorithms are mirrors of one another. The scheme in question can be used with several freely available hard drive encryption programs, such as TrueCrypt. In many of the scenarios in which plausible deniability is desirable, this scheme works excellently; it is furthermore easy to use. However, the deniability offered by this scheme is often plausible at best. That is, it often does not afford *probably* deniability, which may be necessary to ward off certain malicious adversaries.

The remaining existing schemes are all *public-key* schemes, which use separate keys for encryption and decryption. The keys generated in public-key schemes are mathematically related and come in pairs. Each key can decrypt what the other encrypts, but neither key can decrypt what it itself has encrypted. Public-key algorithms are so-named because one of the two keys in any pair can be safely made public, allowing anyone to send an encrypted message to the owner of the keys. As long as the other key, called the *private key*, is kept secret, only the owner of the keys will be able to read the messages directed to him or her.

The existing plausibly deniable public-key schemes may, in theory, work for hard drive encryption. However, they are not ideal for this task, in part because the existing schemes were developed for use by mutually trusting parties who wish to communicate with one another. Typical messages between two parties will be negligibly short compared to even the smallest of hard drives. The content of a typical hard drive is large enough that the use of the existing public-key schemes for hard drive encryption may be very awkward.

Another problem that would arise from the application of the existing public-key schemes is the necessary key length. The keys used in public-key algorithms must be much larger than those used in private-key algorithms in order to achieve the same level of security. In short, this is because the two keys used in public-key schemes are mathematically related, and this relation allows attackers to deduce the private key from the public key by means faster than a brute force attack. The easiest way of defending against these attacks is to simply increase the key length, preventing anybody from executing such an attack in a reasonable amount of time.

The inconvenience involved in using long keys in public-key schemes is compensated for by the convenience gained in key exchanges. Two parties who wish to securely communicate using a private-key scheme must somehow agree on a key without revealing their key to eavesdroppers. If the two parties use a public-key scheme, however, they can simply exchange their public keys without worrying about eavesdroppers who learn their public keys. But while this is an attractive feature of public-key algorithms, it does not benefit individuals who wish to keep the contents of their hard drives secret. Thus, using the existing plausibly deniable public-key schemes for hard drive encryption would require the use of unnecessarily long keys.

This paper develops a plausibly deniable private-key encryption scheme by creating a means by which one sensible plaintext can be encrypted to another. That is, when one plausible plaintext is encrypted, the resulting ciphertext is itself another plausible plaintext. This way, a user can encrypt a non-sensitive dummy plaintext using an existing private-key encryption algorithm. If forced to reveal their encrypted information, the user can decrypt their ciphertext to their dummy plaintext. On the other hand, if the user wishes to access his or her real plaintext, he or she need only decrypt their ciphertext to obtain their dummy plaintext and then decrypt their dummy plaintext to their real plaintext.

Chapter 2

Background Information

2.1 The XOR Cipher

The *basic* or *XOR* cipher is an encryption algorithm predating the field of modern cryptography. The cipher makes use of a plaintext and a key of the same length as the plaintext to produce a ciphertext that is also of the same length as the plaintext. This is accomplished via bitwise application of the XOR, or exclusive OR, function. Given two bits b_1 and b_2, the exclusive OR value $b_1 \oplus b_2$ is equal to 1 if exactly one of b_1 and b_2 is equal to 1; otherwise, $b_1 \oplus b_2 = 0$. Equivalently, the XOR function can be defined as in Table 2.1.

Given a plaintext P with bit sequence $P = p_1 \dots p_n$ and an equal-length key K with bit sequence $K = k_1 \dots k_n$, a ciphertext $C = c_1 \dots c_n$ is produced by letting $c_i = p_i \oplus k_i$ for each $i = 1, \dots, n$. The XOR cipher is at the heart of many commercially used encryption algorithms. It also has the desirable feature of allowing any equal-length plaintext and ciphertext to correspond to one another under some key.

In fact, when a key is generated for the sole purpose of encrypting a

b_1	b_2	$b_1 \oplus b_2$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1 Tabular Definition of the XOR Function.

message via XOR encryption and is then never reused, the result is the so-called "one-time pad" encryption scheme. The one-time pad is famous for being one of the only provably secure encryption schemes. Any message of fixed length can be encrypted to any other message of fixed length via the XOR cipher, so it is impossible to deduce anything about a plaintext given only its ciphertext under a one-time pad encryption. One-time pads are also exceptionally simple from a mathematical and programming standpoint. The cipher can be applied in $O(n)$, with n the bit length of the plaintext, and is nearly trivial to implement in code.

Notably, one-time pad encryption does provide an easy means of creating a scheme with many of our desired properties. For any two plaintexts P_1 and P_2 of the same length, we could randomly produce a key K_1 and then obtain a ciphertext C by applying the XOR cipher to P_1 and K_1. By applying the XOR cipher to P_2 and C, we produce another key K_2 which can be used to decrypt the ciphertext C to the second plaintext P_2. Since K_1 was randomly produced, C will be random, and hence K_2 will also be random. This scheme easily allows us to decrypt a ciphertext C to two (or more) different plaintexts of our choosing.

However, while this scheme is easy and works excellently, it is completely impractical in the context of hard drive encryption. This is because the encryption keys used must be of the same length as the plaintext. The most popular private-key encryption schemes require keys between 16 and 32 bytes in length. To encrypt a 500 gigabyte hard drive (an example we will use throughout this paper), we would need a key with slightly more than 500 billion bytes. Increasing the key length by a factor of multiple billions is simply not practical, especially considering that 32 byte keys are already difficult for most people to remember.

Initially, it might appear as though we can avoid this difficulty by randomly generating a 32 byte key and then producing our 500 billion byte key by repeating the same 32 bytes over and over. Our encryption key is not actually random, but it is obtained by repetition of a random key, so it may seem as though this key is good enough.

However, creating an XOR cipher key in this way is a grave mistake. One-time pads are so-called because the encryption key must never be used to encrypt another plaintext. If the same key K is used to encrypt two plaintexts P_1 and P_2, resulting in ciphertexts C_1 and C_2, then an eavesdropper can learn the bitwise XOR $P_1 \oplus P_2$ of the two plaintexts by computing the bitwise XOR of the ciphertexts. Once the bitwise XOR of the two plaintexts is known, it is usually possible to easily deduce both plaintexts by performing certain types of frequency analysis on $P_1 \oplus P_2$. Thus, resusing a one-time pad key results in the compromise of the security of *all* encrypted plaintexts. The provable security of one-time pads breaks down if a key is reused, because the existence of multiple ciphertexts under the same key can give an attacker useful information about the plaintexts.

This might not immediately seem to pose a problem for encryption using a long key formed by repeating the same random bits. However, encrypting a long plaintext with such a key is equivalent to encrypting many shorter component plaintexts with the same key. Thus, an attacker can use the attack described above to recover the long plaintext with high probability.

The XOR cipher does not by itself allow for a practical plausibly deniable scheme. However, this cipher will be the backbone of the scheme which we do eventually develop.

2.2 TrueCrypt Hidden Volumes

TrueCrypt is a freely available piece of software that allows for hard drive encryption. The fundamental units under TrueCrypt encryption are called *volumes*. A user can encrypt their entire hard drive with TrueCrypt, in which case their whole hard drive is the volume. Alternately, a user can create a volume by setting aside a fixed amount of space on their hard drive. The volume is encrypted by TrueCrypt. If the plaintext of a volume contains empty space, a user can decrypt the volume, add another plaintext file to the volume, and the re-encrypt the volume. Users can also nest volumes by placing encrypted volumes in the plaintext of larger encrypted volumes.

TrueCrypt is well known for its *hidden volume* functionality, described by Andrew Y. (2012). Hidden volumes provide a way for users to hide sensitive information along plausible dummy plaintexts such that the dummy plaintexts can be revealed without revealing the real plaintext. The only requirement is that the dummy plaintexts and real plaintexts have total length equal to the capacity of their containing volume.

For the sake of simplicity, suppose we have a single real plaintext P_r and a single dummy plaintext P_d. A user should choose one of the private-key encryption algorithms available for use with TrueCrypt as well as two private keys K_r and K_d. The real plaintext is encrypted using the key K_r, yielding a ciphertext C_r, while a ciphertext C_d is obtained by encrypting the dummy plaintext with the other key K_d. Each plaintext will be of the same length as its corresponding plaintext. (All encryption algorithms used by TrueCrypt have this property.) A single ciphertext C is then formed by concatenating the two ciphertexts C_r and C_d. When this ciphertext is decrypted using the dummy key K_d, the portion of the ciphertext formed from C_d is correctly decrypted to the dummy plaintext P_d, whereas the remainder C_r of the ciphertext is incorrectly decrypted and appears as a random mess of bits. Similarly, decryption using the real key K_r allows a user to recover their real plaintext. Therefore, a hidden volume user is able to decrypt their data to a plausible plaintext while also retaining the ability to recover their real plaintext if desired.

Notably, when a volume containing a hidden volume is decrypted, it appears as though all empty space has been filled with random data (which is actually an incorrectly decrypted ciphertext). This should not in general raise the suspicions of threatening adversaries, since it is not uncommon for people to fill the empty space on their hard drives with pseudo-random data. In fact, TrueCrypt does this by default.

Somewhat more specifically, a TrueCrypt volume contains both a standard volume header as well as space for a hidden volume header. If no hidden volume is present, the hidden volume header space is filled with pseudo-random data. When TrueCrypt is given a decryption key, it first attempts to decrypt the standard header. If this decryption is successful, TrueCrypt decrypts the remainder of the volume using the same key. If, on the other hand, decryption of the standard header fails, TrueCrypt then attempts to decrypt the hidden volume header using the same key, again decrypting the remainder of the volume if decryption is successful. Refer to Figure 2.1.

It is worth mentioning that the addition of files to a volume containing a hidden volume may destroy part of the hidden plaintext. The hidden plaintext will appear as random data if the volume is decrypted using the dummy key, and since TrueCrypt writes over all empty space with pseudo-random data, the bits containing the hidden plaintext will be treated as "empty". Adding files to the volume decrypted with a dummy key may therefore overwrite some of the hidden plaintext bits, effectively corrupting the real ciphertext. A meddlesome adversary may therefore destroy hidden plaintexts by tampering with the volume. It is not advisable to prevent changes to the data in the volume, however, because this may clue the adversary in to the existence of a hidden volume.

TrueCrypt hidden volumes are extremely practical and will work just fine in many (perhaps most) cases. They do, however, struggle to provide *probable* deniability; plausible deniability is often the best that they can offer.

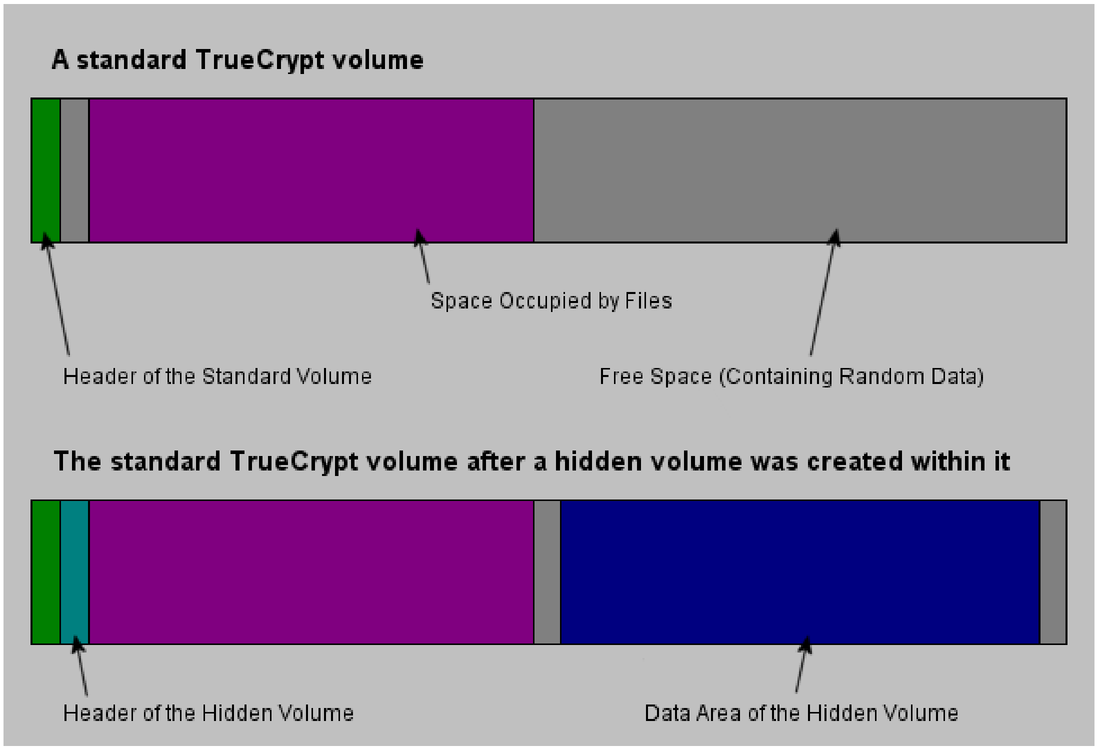
A standard TrueCrypt volume
Space Occupied by Files
Header of the Standard Volume
Free Space (Containing Random Data)
The standard TrueCrypt volume after a hidden volume was created within it
Header of the Hidden Volume
Data Area of the Hidden Volume

Figure 2.1 A demonstration of how TrueCrypt hidden volumes work. Source: Y. (2012).

Hidden volumes are a highly advertised feature of TrueCrypt. Therefore, an adversary who is familiar with TrueCrypt may feel suspicious upon seeing empty space inside a decrypted volume, regardless of whether there is actually any further hidden content. If a volume has a capacity of n bits, a particularly threatening adversary may continue to threaten a user until n bits of plaintext have been revealed from the volume. Such adversaries are troublesome both for people who do and who don't have hidden volumes.

A TrueCrypt user might try to hide their real plaintext in a small hidden volume within a much larger volume. This way, an adversary who sees the decrypted volume will notice that very little of the space in the volume is "empty", hence making it somewhat more plausible that there is no further hidden content. This method might work, depending on the adversary in question. However, it is risky and also undesirable because it requires a user to dedicate only a relatively small portion of their hard drive to their sensitive information.

TrueCrypt hidden volumes and their potential problems served as the original inspiration for this thesis.

2.3 Existing Public-Key Schemes

As explained in the introduction, the existing deniable encryption schemes other than TrueCrypt hidden volumes are not ideal for deniable hard drive encryption. This is because the encryption keys need to be very long, and also because the existing schemes are best suited for communication rather than storage purposes. Nonetheless, these public-key schemes are of theoretical (if not, for our purposes, practical) interest.

Deniable encryption was introduced by Canetti et al. (1996) in their paper "Deniable Encryption". The authors define a scheme to be *sender-deniable* if the sender of an encrypted message, when approached by a malicious party, can decrypt their message to a plaintext different than the one that was actually communicated. The notion of receiver-deniability is defined similarly. A scheme is sometimes called *bi-deniable* if it is both sender- and receiver-deniable. Furthermore, a scheme is called δ-deniable if the existence of the original plaintext can be detected with probability δ or less.

The authors demonstrate how to construct δ-deniable schemes, for arbitrarily small δ. However, the length of each message in any such scheme scales roughly as $1/\delta$. Thus, achieving a reasonably low detection probability requires very long communications—longer than would be typical, certainly.

Canetti et al. also provide a deniable private-key scheme, but it is simply the XOR cipher. This is not very practical for communication purposes because it requires two communicating parties to share a large number of bits not known to anybody else. The XOR cipher is not suitable for mass data storage for reasons already explained.

Since the seminal paper by Canetti et al., other researchers have built on their work. For example, Dürmuth and Freeman (2011) produced the first public-key algorithm that is both sender-deniable and δ-deniable for arbitrary δ. In the same year, O'Neill et al. (2011) introduced the first bi-deniable public-key algorithm. Also, Klonowski et al. (2008) expanded differently on the work of Canetti et al. by constructing a scheme that allows for arbitrarily small detection probability δ while also providing a user with the ability to deny that he or she is using a deniable encryption scheme. In short, the work on public-key deniable encryption since Canetti et al. has provided bi-deniability, stronger overall deniability, and the ability to achieve deniability in shorter messages.

The existing deniable public-key schemes are clever and should not be disregarded entirely, even for the purposes of personal data storage. However, if a practical private-key scheme can be constructed, then it would likely be preferable.

2.4 Steganography

Steganography is the study of methods of achieving information security by hiding the very existence of the information. By contrast, observe that cryptography does not hide the existence of information; rather, it renders information unreadable by anyone who does not know the decryption key. Strong encryption thwarts eavesdroppers by preventing them from making sense of information. Well-implemented steganographic methods thwart eavesdroppers by preventing them from even finding the information in the first place.

An example of a purely cryptographic method is encrypted email. A sophisticated eavesdropper will not be able to make sense of encrypted email, but will in general be able to see that the encrypted email exists.

An example of a purely steganographic method, on the other hand, is the concealment of information within an image file. Suppose we have an

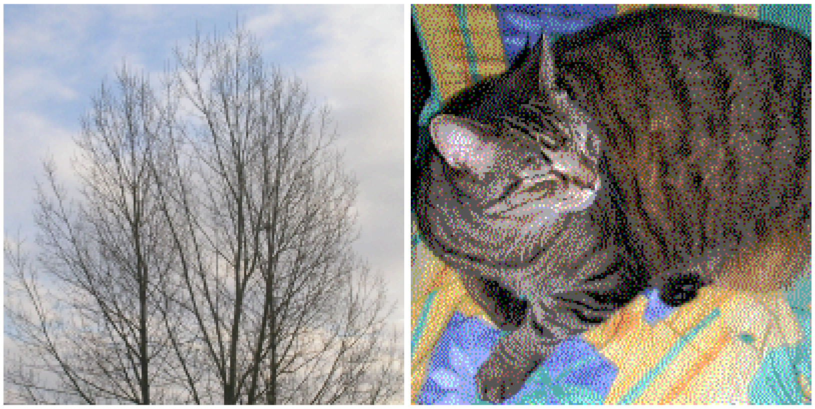

Figure 2.2 The image of a cat is steganographically hidden within the image of a tree. Source: Greene (2009).

n-bit image file I_1 that we wish to hide as well as another image I_2 containing at least n bytes (not bits). We can conceal the former within the latter by overwriting the least significant bit of the ith byte of I_2 with the ith bit of I_1, for each $i = 1, \ldots, n$. This embeds I_1 within I_2, so that anyone aware of the existence of I_1 can easily extract it from the new image I_2'. However, the differences between I_2 and I_2' will usually not be apparent to the naked eye, so people unaware of the existence of I_1 will tend to remain unaware of I_1 even after viewing the new image I_2'. Refer to Figure 2.2 for an example of this image concealment algorithm in action.

Encryption on its own is strong but does not allow for deniability. Steganography allows for deniability but is weak on its own. Observe that even if an eavesdropper is not previously aware of the existence of a concealed image, he or she may still be able to easily extract the hidden image. This is not unrealistic—there are pieces of software made for the sole purpose of finding steganographically hidden information within image files. However, cryptography and steganography can be combined to great effect. For instance, if information is encrypted and then embedded within an image file, the image may look essentially the same as before while still providing an eavesdropper with no easy means of extracting the hidden information.

Deniable encryption schemes are both cryptographic and steganographic. Our scheme will encrypt a dummy plaintext while also, in some sense, hiding a real plaintext within the dummy plaintext. The methods employed in this paper will never be explicitly steganographic. However, it is worthy of note that the development of a deniable encryption scheme represents a departure from conventional, “pure” cryptography.

Chapter 3

Framework For The Scheme

3.1 Our General Approach

There are several different ways to go about constructing a plausibly deniable encryption scheme. Perhaps the most natural approach is to algebraically force two chosen plaintexts (one secret and one dummy) to the same ciphertext using different encryption algorithms or keys. This way, a user can choose which plaintext to reveal by specifying the appropriate combination of algorithm and key.

A second, more steganographic approach is to "encrypt" a chosen real plaintext to a chosen dummy plaintext, which is then encrypted using a standard encryption algorithm to obtain a ciphertext. The dummy plaintext can be recovered by undoing the standard encryption. The real plaintext can be recovered by recovering the dummy plaintext and then "decrypting" it to the real plaintext. This approach is more steganographic than the first because the real plaintext is, in some sense, hidden within the dummy plaintext. These two approaches are explained pictorially by Figure 3.1.

This paper will focus exclusively on the second approach. The first approach turns out to be intractable in the absence of serendipity—that is to say, it does not work for most possible plaintext pairs. The second approach, while still difficult, is possible.

Our approach, more specifically, will be to create a means by which an arbitrary pair of sensible plaintexts can be associated with one another via a key of reasonable length. The crux of this paper will be the creation of a mathematical framework which allows for this.

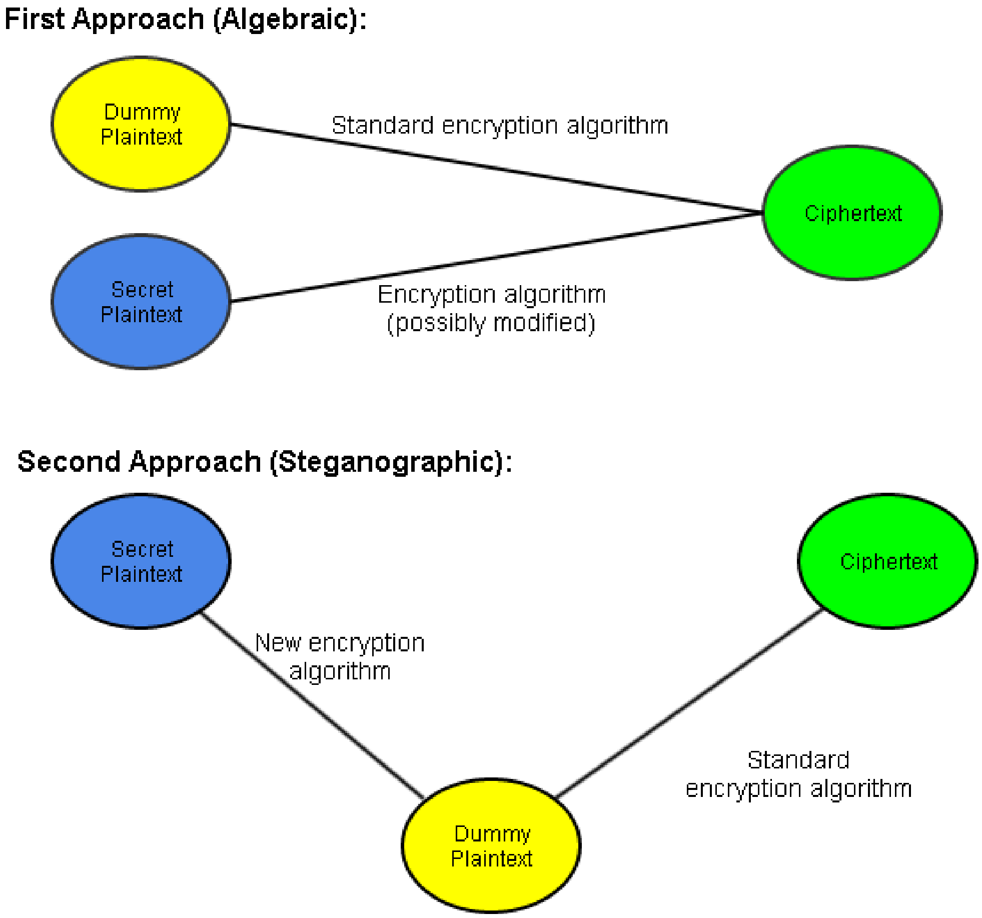
First Approach (Algebraic):
Dummy Plaintext
Standard encryption algorithm
Ciphertext
Secret Plaintext
Encryption algorithm (possibly modified)
Second Approach (Steganographic):
Secret Plaintext
Ciphertext
New encryption algorithm
Standard encryption algorithm
Dummy Plaintext

Figure 3.1 Two different ways to approach the construction of a plausibly deniable encryption scheme. (Figure produced by Gliffy.)

3.2 Message Spaces

The phrase *Message Space* refers to the set of messages (typically viewed as binary strings) to which a given cryptographic algorithm can be applied. The phrases *Ciphertext Space* and *Key Space* are defined similarly. Most algorithms in widespread use have as their message space the set $\{0,1\}^n$ of all binary messages of length n, for some n specific to the algorithm in question. This makes sense—these algorithms work equally well on all possible input messages, so it makes little sense to needlessly prohibit certain messages from being used as input.

In the context of deniable cryptosystems, however, there are some advantages to be had by limiting the size of our message space. Suppose, for instance, that our message space M consists of messages of length n and satisfies $|M| \leq 2^k$, for some $k < n$. Then it is possible to create an injective map $f : M \to \{1, \ldots, 2^k\}$. Hence, we can represent any message $m \in M$ using fewer than n bits, since $f(m)$ is unique to m and must be representable using at most k bits. Given two messages $m_1, m_2 \in M$, we can then "encrypt" m_1 to obtain m_2 as follows. First compute $f(m_1)$. Perform the XOR cipher on $f(m_1)$ using $f(m_1) \oplus f(m_2)$ as the encryption key; the resulting ciphertext is $f(m_2)$, because

$$f(m_1) \oplus \big(f(m_1) \oplus f(m_2)\big) = f(m_2) \tag{3.1}$$

Lastly, apply the inverse function f^{-1} to this ciphertext. This yields m_2, because

$$f^{-1}\big(f(m_2)\big) = m_2 \tag{3.2}$$

Observe that the map f need not be invertible, since it is injective but not necessarily surjective. However, it is invertible for any value corresponding to a message in M, which is sufficient for our purposes. (More formally, we could instead let M' be the image of M under f. The mapping $f : M \to M'$ is then invertible.)

Although the only encryption algorithm we have used is the XOR cipher, we have successfully mapped one message of length n to another using a key with fewer than n bits. In essence, we have accomplished this by compressing the two messages of length n and then using an XOR cipher to obtain the compressed representation of one message from the compressed representation of the other.

Given such a mapping f and a suitably small message space M, we can achieve plausible deniability using two keys of reasonable length as follows. Select a real plaintext $m \in M$ and a dummy plaintext $m' \in M$. Let $K_1 = f(m) \oplus f(m')$ be our first key. Select a standard encryption algorithm E as well as a second key K_2 that can be used with the algorithm E. The ciphertext C in this protocol is given by

$$C = E(m \oplus K_1, K_2) = E(m', K_2) \tag{3.3}$$

In other words, encrypt the dummy plaintext with the standard algorithm E and key K_2 to obtain the plaintext. The real plaintext can be recovered as follows:

$$m = f^{-1}\left(K_1 \oplus f\left(E^{-1}(C, K_2)\right)\right) \tag{3.4}$$

In words, we undo the encryption E to obtain the dummy plaintext m', compute $f(m')$, perform the XOR cipher on $f(m')$ with key K_1 to obtain $f(m)$, and then perform the inverse map f^{-1} to recover m. Recovery of the dummy plaintext m' is simpler:

$$m' = E^{-1}(C, K_2) \tag{3.5}$$

Importantly, the real message m is never obtained in plaintext during the process to recover the dummy message m'. Therefore, an adversary monitoring all of the intermediate results of encryption and decryption will never see m in plaintext form. A scheme such as this one allows for plausible deniability, provided that the key K_1 is sufficiently short.

3.3 Practical Considerations

While the protocol described in the previous section works excellently in theory, it is difficult to use in practice. If we were to use the message space $M = \{0, 1\}^n$ of all binary messages of length n, then the first key would need to have length $|K_1| = n$, while the second key would probably have size on the order of $|K_2| \sim 128$. Thus, if the message to be encrypted is the content of a 500 GB hard drive, then the total necessary key length for the protocol is

$$|K_1| + |K_2| \approx 2^{42}, \tag{3.6}$$

which is unacceptably long. This is why we seek to restrict the size of our message space. However, even if we reduce the message space size by a factor of square root to $|M| = 2^{n/2}$, then the total necessary key length is still

$$|K_1| + |K_2| \approx \frac{1}{2} \cdot 2^{42} = 2^{41}, \tag{3.7}$$

which is still unacceptably long. Evidently, we need our message space to be miniscule compared to the universal message space $\{0,1\}^n$.

Even if the message space is reduced to a manageable size, another potential problem is the algorithmic complexity incurred by the map $f : M \to \{1,\ldots,|M|\}$. Recovery of the dummy plaintext requires no use of the map f. Recovery of the real plaintext, on the other hand, requires at least one application each of f and f^{-1}. Thus, the time required to recover the real plaintext in this protocol is lower-bounded by the time required to compute both f and f^{-1}. If these operations cannot be performed quickly, then the protocol overall will be slow; in a worst-case scenario, it may be too slow for practical use. Therefore, it is mandatory in practice that we have efficient algorithms for the operations f and f^{-1}.

Another similar requirement is that it should be possible to perform the encryption and decryption algorithms using a reasonable amount of computer memory. A fairly modest amount of memory for computers in 2015 is 4 gigabytes, although it would be better to limit memory usage to 1 gigabyte at a time since a substantial amount of computer memory may already be devoted to other tasks.

The only practical consideration addressed in this paper is the necessary total key length. While efficient algorithms for f and f^{-1} would be vital for any implementation of our scheme, the purpose of this paper is to develop a theoretical framework by which plausible deniability can be achieved. Implementation of the scheme developed here is beyond the scope of the paper. However, Appendix A develops efficient algorithms for f and f^{-1} for a simplified version of our scheme as a proof of concept.

There are numerous other concerns that would arise if the scheme developed here were ever implemented in production code, but these are all considerations for software engineers and are thus beyond the scope of this paper.

Chapter 4

Methods of Message Space Reduction

It would be ideal to create a standard message space M for use with our scheme so that users do not have to specify their message spaces as additional parameters. Other than its size, which should be negligible in comparison to $\{0,1\}^n$, the only requirement for M thus far is that it should contain any user's real plaintext and dummy plaintext. Roughly speaking, then, we may proceed by letting M consist only of the sensible messages of length n.

We can perhaps do better than just this. For instance, a collection $m_1, \ldots, m_r$ of messages might all represent the same sensible data, differing only in their representations of blank space or in the order of its distinct sections. For each such family of messages, we can choose exactly one of the m_i to include in M, effectively decreasing the size of M without losing any real content.

It is not immediately clear, however, how large the resulting message space M will be, or what, generally, it will look like. In large part, this is because we have not specified what it means for a message to be "sensible"—nor does it seem likely that this can be specified in a satisfactory way. We might say that the number of distinct sensible messages of fixed length is *uncountably finite*, meaning that it is finite but has not been expressed combinatorially in closed-form. (Note that this definition differs sharply from the topological notions of countable and uncountable infinities.) Even if the number of distinct sensible plaintexts is uncountably finite, however, we may still be able to construct a message space M of reasonable size which contains all or most distinct sensible plaintexts.

One way of proceeding is to list some expected characteristics common to all plausible plaintexts and then count the number of binary messages with those characteristics.

4.1 The Even-Split Property

One characteristic we might expect plausible plaintexts to have is that they have roughly equal numbers of 1s and 0s in their bit sequences. This is because even if a message has significant overall structure, it is still likely to look random on first glance at a bit or byte level. We say that a binary string has the *even-split property*, or ESP, if exactly half of its bits are 0s.

Several potential concerns arise from this property definition. Firstly, a binary string representing the plausible content of a computer hard drive may fail to have ESP if the hard drive has blank space. If the blank space is represented, for instance, as a large string of 0s, then the hard drive content may be plausible and yet have many more 0s than 1s. This problem can probably be entirely avoided in practice by requiring users to write over blank hard drive space with pseudo-random data, which will tend to exhibit ESP. If the user's actual content and blank space both have ESP, then their hard drive overall will also have ESP. Even if a user's actual content does not exhibit ESP, then their overall message may still have ESP if the blank space is altered so as to counteract the discrepancy in the numbers of 0s and 1s. (Here, we are essentially identifying a family of messages corresponding to the same content, differing only in their representations of blank space, and including in M only the ones that have ESP. This both reduces the size of M and cuts down on the amount of redundancy in M.)

Perhaps a larger concern is that it will be rare for a user's plaintexts to have *exactly* equal numbers of 0s and 1s, even if the numbers of the two are roughly equal. In the chapters that follow, our binary string counting will include only those strings with exactly equal numbers of 0s and 1s. Practically speaking, I expect that this will not cause the scheme developed here to fail. If a user has even a small amount of blank space on their hard drive, then it will probably be possible to modify that blank space so that the overall message has ESP.

Failing this, however, it is possible to extend the schemes developed in this paper to strings that do not quite have ESP. Specifically, if we allow our strings to have any number of 0s from $\frac{n}{2} - \frac{k}{2}$ to $\frac{n}{2} + \frac{k}{2}$, for any $k > 0$, then our schemes will still work just as well provided that we lengthen our key slightly. The necessary lengthening is upper-bounded by $\log k$, which is a fairly small penalty.

4.2 Absence of Long Runs

Another characteristic we might expect of plausible plaintexts is that they do not contain excessive repetition of the same patterns. For example, we would expect a binary string representing actual content to not have any 1000 consecutive bits equal to 0, since it would be rare for such a substring of 0s to convey any real information. We define a *c-run* of a character α to be any occurrence of c or more consecutive α characters within a string. We might then expect that for sufficiently large c, any plausible plaintext will not contain any c-runs (of 0s or of 1s).

As was the case with ESP, several potential concerns arise here too. Firstly, if blank space is represented by a long string of 0s or 1s, then a message may be a plausible plaintext and yet contain c-runs, even for large values of c. As was the case with ESP, this problem can probably be entirely avoided in practice by modifying the blank space. In fact, in almost all cases, it should be possible to modify the blank space to force a string to have both ESP *and* no long runs, provided that the rest of the string satisfies or nearly satisfies these properties.

Writing over the blank space with pseudo-random data may or may not work, depending on the chosen value of c. However, if the actual content of a message has ESP and no long runs, then the entire message can be made to satisfy these properties too by writing over the blank space with the data 010101 . . . , where the pattern 01 repeats. On the other hand, if the content portion of the message has no long runs but has more 0s than it has 1s, then this difference can be made up by writing over the blank space with a pattern such as 011011011 . . . . In short, it is probably easy in practice to avoid any ESP or long run problems arising from blank hard drive space.

Another practical concern that arises is that it is not immediately clear which value of c should be chosen. On the one hand, we could let $c = n$, since peoples' hard drive contents will almost definitely not be entirely 0s or entirely 1s. However, this value of c is too large, since the only strings excluded in this case by the required absence of c-runs are 0^n and 1^n, which would have been excluded anyway since they don't have ESP. Thus, we see that this value of c is much too large and does not actually reduce the size of our message space. On the other hand, we could choose a small value such as $c = 2$. But the only binary strings of length n with no 2-runs are $(01)^{n/2}$ and $(10)^{n/2}$, both of which are exceedingly redundant and convey very little information. Thus, we see that $c = 2$ is too small of a value, since it excludes essentially every meaningful string from our message space.

It will be necessary to find a balance. That is, we should choose a value of c that is small enough to exclude messages with excessive redundancy and yet large enough that our resulting message space contains all or most plausible plaintexts. The chosen value of c is so important that chapter 6 is devoted to the selection of c.

Chapter 5

Assessing the Strength of ESP

Our next step is to assess the message space reductions achieved individually by ESP and the required absence of long runs. In other words, we will impose each requirement on its own on the set $\{0, 1\}^n$ of all binary messages of length n and then determine or estimate the size of the resulting message space M.

To this end, we first count the binary strings of length n that have ESP. Combinatorially speaking, this is very easy to do. The number of binary strings of length n in which half of the bits are 0s is simply equal to the number of ways to choose $n/2$ bits to be 0s (with the remaining bits all 1s). Thus, ESP on its own gives us a message space M satisfying

$$|M| = \binom{n}{n/2} \tag{5.1}$$

But while this count is absolutely correct, it does not make it obvious how $|M|$ compares to the number of binary strings of length n, which is 2^n. To more directly compare $|M|$ to 2^n, we make use of Stirling's approximation, which states the following:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{5.2}$$

Specifically, this approximation converges to equality as $n \to \infty$. The approximation is useful to us because it converges very quickly. The error in the approximation scales roughly as $\frac{1}{12n}$. When n is chosen to be 2^{42}, equal to the number of bits in a 500 GB hard drive, we see that the relative error in this approximation is truly miniscule. To make use of this approximation, we first represent $|M|$ as follows:

$$|M| = \binom{n}{n/2} = \frac{n!}{\left[\left(\frac{n}{2}\right)!\right]^2} \tag{5.3}$$

We then use Stirling's approximation to replace all of the factorials in (5.3), yielding

$$\binom{n}{n/2} \approx \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\left[\sqrt{\pi n}\left(\frac{n}{2e}\right)^{n/2}\right]^2}, \tag{5.4}$$

which simplifies to

$$|M| \approx \frac{2^n}{\sqrt{\frac{\pi n}{2}}}. \tag{5.5}$$

We thus see that ESP on its own reduces our message space by a factor of roughly $\sqrt{\frac{\pi n}{2}}$. In absolute terms, this reduction is significant. For $n = 2^{42}$, this translates to a reduction by a factor of roughly 2 million. In other words, by requiring binary strings to have ESP, we remove all but one 2-millionth of the possible binary strings from our message space, while keeping at least one binary string corresponding to each plausible plaintext.

However, ESP on its own is nowhere near strong enough to make our scheme practical. Note that in the map $f : M \to \{1, \ldots, |M|\}$, the values of $f(m)$ for $m \in M$ may be as large as $|M|$. Thus, the number of bits in our first key may be as large as

$$|K_1| = \log |M|. \tag{5.6}$$

Equations (5.5) and (5.6) together yield (after some simplification) the following:

$$|K_1| \approx n - \frac{1}{2} \log \frac{\pi n}{2}. \tag{5.7}$$

For $n = 2^{42}$, we thus obtain

$$|K_1| \approx 2^{42} - 21. \tag{5.8}$$

Although the reduction in our message space is drastic, the reduction in our key size is small. We may conclude that ESP is strong in absolute terms but weak compared to what is necessary.

Chapter 6

Selecting Our Maximum Run Length

Whereas ESP describes a single method of reducing message space size, the “no c-runs” requirement actually describes a large set of ways to shrink the message space, depending on the chosen value of c. Therefore, before assessing the strength of the effect achieved by prohibiting long runs, we first decide on a suitable value of c. Specifically, we will estimate the message space size $|M|$ in terms of n and c when c-runs are prohibited. Given this estimation, we will then decide on an acceptable size for $|M|$ and solve for c in terms of n.

We will settle for a mere estimation in this chapter because counting binary strings with no c-runs turns out to be a difficult combinatorial problem. We solve the problem exactly in chapter 7, but for the purposes of this chapter, an estimation will suffice to provide us with an idea of the value we should choose for c.

Specifically, we will estimate the number of binary strings with no c-runs by estimating the number of binary strings which *do* contain at least one c-run and then subtracting this estimate from the number 2^n of all binary strings of length n. If s is a binary string of length n, then a first c-run in s can occur at any index from 1 to $n-c+1$, inclusive. Thus, there are $n-c+1$ ways to choose the index at which the first c-run in s begins.

For any starting index of the first c-run in s, there are $n-c$ bits of s not appearing in this first c-run. These bits can take on many of their 2^{n-c} total possible values—specifically, these $n-c$ bits can take on any values which produce no further c-runs prior to the designated first c-run. We will assume here that the substring of s generated by these $n-c$ other bits can take on most of their 2^{n-c} possible values regardless of where the first c-run in s begins. This allows us to estimate the number of binary strings of length n with at least one c-run as $(n-c+1)2^{n-c}$, which in turn allows us to estimate the number of binary strings of length n with no c-runs as

$$2^n - (n-c+1)2^{n-c}. \tag{6.1}$$

If we let our message space M contain every binary string of length n with no c-runs, then (6.1) provides us with an estimate for the resulting size of M. That is,

$$|M| \approx 2^n - (n-c+1)2^{n-c}. \tag{6.2}$$

Optimistically speaking, we will be able to shrink our message space to a size of 2^{256} or less, so that the first key in our scheme will have a standard, reasonable length of a few hundred bits. To this end, we require

$$2^{256} \approx 2^n - (n-c+1)2^{n-c}. \tag{6.3}$$

Note that for n on the order of $n \sim 2^{42}$, the desired message space size 2^{256} is absolutely negligible compared to 2^n. Thus, (6.3) holds true whenever

$$2^n \approx (n-c+1)2^{n-c}. \tag{6.4}$$

This, in turn, requires

$$n \approx \log(n-c+1) + (n-c), \tag{6.5}$$

Which is easily rearranged to yield

$$c \approx \log(n-c+1). \tag{6.6}$$

Equation (6.6) is transcendental in c, but observe that in practice, we expect c to be very small compared to n. Even if we allow enormous runs of length $c = 1000$ in our messages, our chosen value of c is still tiny compared to $n = 2^{42}$. We therefore make the approximation $n-c+1 \approx n$ in order to obtain

$$c \approx \log n. \tag{6.7}$$

We have made numerous approximations along the way to obtain this result, and at least one of those approximations is quite questionable. In particular, after designating the index at which the first c-run within a string should begin, it is not clear that "most" possible choices for the remaining $n-c$

bits will be acceptable. By simply using 2^{n-c} here, we have overestimated the number of binary strings with c-runs, leading us to underestimate the number of binary strings with no c-runs. It is not immediately clear how much of an underestimate this produces.

Nonetheless, equation (6.7) makes for a valuable indicator of roughly which values of c we should choose. Observe that smaller values of c result in smaller message spaces; if we are worried about the accuracy of our estimation, we may try to compensate some by using an even smaller value of c, such as $c = \frac{1}{2}\log n$. In fact, this is the value we shall use henceforth.

It is worth pointing out that for $n = 2^{42}$, equation (6.7) gives us $c = 42$, whereas the heuristic $c = \frac{1}{2}\log n$ yields $c = 21$. I suspect that both of these values are usable in practice; after a message has been altered so as to remove unnecessary long runs, it seems unlikely to me that there will necessarily be 21-runs remaining. In fact, I suspect we could use values of c that are even a little smaller, such as $c = 16$ (which corresponds to there being no more than three consecutive identical bytes in our messages). Regardless, we shall settle on the value $c = \frac{1}{2}\log n$ for now.

Chapter 7

Assessing the Strength of the "No Long Runs" Requirement

Having decided upon a run length c to prohibit in our message space, we can now evaluate the message space reduction achieved by requiring our messages to have no c-runs, much like what we did with ESP in chapter 5. As in chapter 5, we begin by counting the binary strings of length n containing no c-runs. Unlike chapter 5, however, this combinatorics problem is highly nontrivial. The first two sections of this chapter are devoted to counting the binary strings with no c-runs. The third and final section of this chapter is where we actually evaluate the message space reduction achieved by prohibiting long runs.

7.1 Counting Binary Strings With No c-runs of 1s

As a first step, we count the binary strings of length n containing no c-runs of 1s, but which may contain c-runs of 0s. We begin with this simplified problem in part because the solution methods will prove useful, but also because this problem has already been solved in existing mathematical research. Specifically, Nyblom (2012) showed that the number of binary strings of length n containing no c-runs of 1s is given by a shifted version of the c-generalized Fibonacci sequence, F_n^c. The c-generalized Fibonacci numbers are defined by the initial conditions $F_n^c = 0$ for $n \leq 0$, $F_1^c = 1$, and

$$F_n^c = F_{n-1}^c + \cdots + F_{n-c}^c \tag{7.1}$$

for $n \geq 2$. (Observe that the 2-generalized Fibonacci sequence is just the Fibonacci sequence.)

Let S_n^c denote the number of binary strings of length n containing no c-runs of 1s. In particular, Nyblom showed that $S_n^c = F_{n+1}^c$ for all n. Nyblom proved that these sequences are the same by showing that they have the same initial conditions and satisfy the same recurrence relation.

Observe, firstly, that there exist no binary strings of length less than 0. Thus, we have

$$S_n^c = F_{n+1}^c = 0 \tag{7.2}$$

whenever $n < 0$. For $n = 0$, we have

$$S_0^c = F_1^c = 1, \tag{7.3}$$

since there is exactly one string of length 0: the empty string. (The empty string contains no c-runs of 1s for any $c > 0$.)

Equations (7.2) and (7.3) together prove that S_n^c and F_{n+1}^c have the same initial values. All that remains is to show that the two sequences satisfy the same recurrence relation. Nyblom did this via case work on the index of the first 0 in any binary string that lacks c-runs of 1s. Note that any such binary string must have its first 0 no later than index c—otherwise, the first c characters of the string would be a c-run of 1s. If the first 0 appears at index i, then the first i characters of the string are identical to the string $1^{i-1}0$. The remaining $n - i$ characters of the string can then form any of the S_{n-i}^c strings of length $n - i$ which lack c-runs of 1s. Summing over $i = 1, \ldots, c$, we therefore obtain

$$S_n^c = S_{n-1}^c + \cdots + S_{n-c}^c. \tag{7.4}$$

This recurrence is the same as the one satisfied by the c-generalized Fibonacci sequence. Thus, the sequences S_n^c and F_{n+1}^c satisfy the same initial values and recurrence relation. This concludes Nyblom's proof that $S_n^c = F_{n+1}^c$ for all $n \in \mathbb{Z}$ and $c > 0$.

7.2 Counting Binary Strings With No c-runs At All

Let U_n^c denote the number of binary strings of length n containing no c-runs (of 1s or of 0s). This sequence is more difficult to evaluate in closed-form than S_n^c, but it, too, can be expressed in terms of the generalized Fibonacci numbers.

To demonstrate why U_n^c is more difficult to evaluate than S_n^c, we begin by attempting to count U_n^c using the same methods used by Nyblom to count S_n^c. That is, we attempt to count U_n^c via case work on the index of the first 0 within a string which lacks c-runs.

Similar to S_n^c, the first 0 in any binary string which lacks c-runs can occur no later than index c, because otherwise the first c characters would be a c-run of 1s. Suppose the first 0 in a string without c-runs occurs at index i. The first i characters of the string are then $1^{i-1}0$. The remaining $n-i$ characters can be any binary string of length $n-i$ containing no c-runs, *except* for those beginning with a $(c-1)$-run of 0s—if the remainder of the string were to begin with a $(c-1)$-run of 0s, then our overall string would have a c-run of 0s beginning at index i.

It is not immediately clear how many binary string have no c-runs and begin with a $(c-1)$-run of 0s. We define another sequence, V_n^c, to denote the number of binary strings of length n which contain no c-runs and which begin with a $(c-1)$-run of 0s. Observe that the number of binary strings which contain no c-runs and which begin with a $(c-1)$-run of 1s is also V_n^c. The introduction of another sequence initially serves only to complicate things, but V_n^c serves as a useful proxy which enables us to derive a recurrence relation for U_n^c independent of V_n^c.

As a result of the definition of V_n^c and our reasoning from two paragraphs ago, we see that the number of binary strings of length n which have no c-runs and which have their first 0s appearing at index i is $U_{n-i}^c - V_{n-i}^c$. Summing over $i = 1, \ldots, c$, therefore, we obtain the following recurrence:

$$U_n^c = (U_{n-1}^c - V_{n-1}^c) + \cdots + (U_{n-c}^c - V_{n-c}^c). \tag{7.5}$$

In order to eliminate the terms of the sequences V_n^c from this recurrence, we next derive a recurrence relation for V_n^c. If s is a string of length $n \geq c$ which contains no c-runs and which begins with a $(c-1)$-run of 0s, then the first c characters of s are $0^{c-1}1$. The remainder of s after index c can be any binary string of length $n-c$ which contains no c-runs and which does not begin with a $(c-1)$-run of 1s. Thus, we obtain

$$V_n^c = U_{n-c}^c - V_{n-c}^c. \tag{7.6}$$

Applying (7.6) to each of the terms V_{n-i}^c in (7.5), we obtain

$$U_n^c = \left[U_{n-1}^c - (U_{n-1-c}^c - V_{n-1-c}^c)\right] + \cdots + \left[U_{n-c}^c - (U_{n-2c}^c - V_{n-2c}^c)\right], \tag{7.7}$$

which we can rewrite as

$$\begin{aligned} U_n^c =& (U_{n-1}^c + \cdots + U_{n-c}^c) \\ & - \left[(U_{n-c-1}^c - V_{n-c-1}^c) + \ldots (U_{n-c-c}^c - V_{n-c-c}^c)\right] \end{aligned} \tag{7.8}$$

From (7.5), we know that the portion of (7.8) appearing in brackets is the recurrence for U^c_{n-c}. Therefore, we find that

$$U^c_n = (U^c_{n-1} + \cdots + U^c_{n-c}) - U^c_{n-c}, \tag{7.9}$$

which simplifies to

$$U^c_n = U^c_{n-1} + \cdots + U^c_{n-c+1}. \tag{7.10}$$

Observe that (7.10) is a recurrence for U^c_n independent of the sequence V^c_n. Thus, our proxy sequence has been eliminated, leaving us with our recurrence for U^c_n. This recurrence is the same as the one satisfied by the $(c-1)$-generalized Fibonacci numbers.

We will prove, in particular, that $U^c_n = 2F^{c-1}_{n+1}$ for $n \geq 1$. From (7.10), we know that the two satisfy the same recurrence, so it will now suffice to show that they have the same initial values for $n = 1, \ldots, c-1$.

First, observe that $U^c_n = 2^n$ for all nonnegative $n < c$, since any binary string of length less than c has no c-runs.

It is easy to show inductively that $F^{c-1}_{n+1} = 2^{n-1}$ for $i = 1, \ldots, c-1$. It is true that $F^{c-1}_2 = 1$, since $F^{c-1}_1 = 1$ and $F^{c-1}_i = 0$ for $i \leq 0$. In the inductive step with $1 < n \leq c-1$, we use the recurrence for F^c_n to obtain

$$F^{c-1}_n = F^{c-1}_1 + \left(F^{c-1}_2 + \cdots + F^{c-1}_{n-1}\right). \tag{7.11}$$

Applying our inductive hypothesis to the terms of (7.11), we deduce that

$$F^{c-1}_n = 1 + \left(2^0 + \cdots + 2^{n-3}\right). \tag{7.12}$$

The sum of powers of 2 in parentheses is a geometric series, and as such we deduce that its value is $2^{n-2} - 1$. Therefore, from (7.12), we find that

$$F^{c-1}_n = 1 + \left(2^{n-2} - 1\right). \tag{7.13}$$

Hence, we see that $F^{c-1}_n = 2^{n-2}$, as desired.

We have shown that the sequences U^c_n and $2F^{c-1}_{n+1}$ have the same initial values for $n = 1, \ldots, c-1$ and satisfy the same $(c-1)$-order recurrence relation. It follows that $U^c_n = 2F^{c-1}_{n+1}$ for any $n \geq 1$. (Notably, this equality fails for $n = 0$, for which we have $U^c_0 = 1$ and $2F^{c-1}_1 = 2$. However, $n = 0$ is the only index for which the two sequences differ.)

7.3 Direct Comparison of U_n^c to 2^n

Now that we have shown that $U_n^c = 2F_{n+1}^{c-1}$ for all $n \geq 1$, we can analyze the message space reduction achieved by prohibiting c-runs in our messages. We can do this using several results from Dresden and Du (2014). Dresden and Du demonstrated that the values of the c-generalized Fibonacci sequences can be computed directly as

$$F_n^c = \text{Round}\left[\frac{\alpha - 1}{2 + (c+1)(\alpha - 2)}\alpha^{n-1}\right], \tag{7.14}$$

where α is the positive real root of the polynomial equation

$$x^c - x^{c-1} - \cdots - 1 = 0. \tag{7.15}$$

Dresden and Du also demonstrated that if $c \geq 4$, as it will be in our scheme, then

$$2 - \frac{1}{3c} < \alpha < 2. \tag{7.16}$$

Using (7.14) and (7.16), we obtain the following upper bound and close approximation for F_n^c:

$$F_n^c \approx \frac{1}{\frac{5}{3} - \frac{1}{3c}}\alpha^{n-1}. \tag{7.17}$$

Since $U_n^c = 2F_{n+1}^{c-1}$, we deduce from (7.17) that

$$U_n^c \approx \frac{2}{\frac{5}{3} - \frac{1}{3(c-1)}}\alpha_{c-1}^n. \tag{7.18}$$

We use our standard example $n = 2^{42}$ as well as the value $c = 21$ that results from our heuristic derived in chapter 6. Using these values in (7.18) yields

$$U_n^c \approx \frac{40}{33}\alpha_{20}^n. \tag{7.19}$$

From (7.16), we know that

$$\alpha_{20} > 2 - \frac{1}{60} \approx 2^{0.988}. \tag{7.20}$$

From (7.19) and (7.20), we deduce that

$$U_n^c \approx \frac{40}{33} \cdot 2^{0.988n}. \tag{7.21}$$

For $n = 2^{42}$ and $c = 21$, we thus see that U_n^c is smaller than 2^n roughly by a factor of

$$\frac{2^n}{U_n^c} \approx \frac{33}{40} \cdot 2^{5.3 \cdot 10^{10}}. \tag{7.22}$$

In absolute terms, this reduction is enormous—this message space reduction dwarves that obtained by using ESP on its own. However, similar to ESP, this message space reduction is weak compared to what we need. From (7.21), we see that the number of bits we would need in a key would still be about $0.988 \cdot 2^{42}$, which is large.

Chapter 8

Combining ESP and the "No Long Runs" Requirement

We have assessed both ESP and the prohibition of c-runs and deemed them to be individually insufficient. Our next step is to measure the message space reduction achieved by enforcing both requirements.

Let $U^c_{n,k}$ denote the number of binary strings of length n containing exactly k 0s and which have no c-runs. Observe that the number of binary strings of length n with ESP and no c-runs is given by $U^c_{n,n/2}$. While trying to count $U^c_{n,k}$ may seem much more complicated than just counting $U^c_{n,n/2}$, it really isn't—there is no obvious way to derive a recurrence for $U^c_{n,n/2}$ only in terms of values of the form $U^c_{r,r/2}$.

As in chapter 7, we define a proxy sequence to aid us in deriving a recurrence for $U^c_{n,k}$. Let $V^{c,0}_{n,k}$ denote the number of binary strings of length n which have no c-runs, contain exactly k 0s, and which begin with a $(c-1)$-run of 0s. Define $V^{c,1}_{n,k}$ similarly.

Rather than attempting to evaluate $U^c_{n,k}$ by case work on the index of the first 0, however, we perform case work on whether the first bit is a 0 or a 1. Let s be a binary string of length n with exactly k 0s and no c-runs. If the first bit of s is a 0, then the remainder of s can be any binary string of length $n-1$ which has $k-1$ 0s, no c-runs, and which does not begin with a $(c-1)$-run of 0s. Ths number of such strings is $U^c_{n-1,k-1} - V^{c,0}_{n-1,k-1}$.

If the first bit of s is a 1, on the other hand, then the remainder of s can be any binary string of length $n-1$ with exactly k 0s, no c-runs, and which does not begin with a $(c-1)$-run of 1s. The number of such strings is $U^c_{n-1,k} - V^{c,1}_{n-1,k}$.

Summing the numbers of possible strings s beginning with a 0 or with a 1, we obtain the following recurrence:

$$U^c_{n,k} = (U^c_{n-1,k-1} - V^{c,0}_{n-1,k-1}) + (U^c_{n-1,k} - V^{c,1}_{n-1,k}). \tag{8.1}$$

As in chapter 7, we now derive recurrences for $V^{c,0}_{n,k}$ and $V^{c,1}_{n,k}$. We begin with $V^{c,0}_{n,k}$. Let s be a string of length n which has k 0s, no c-runs, and begins with a $(c-1)$-run of 0s. Then the first c bits of s form the string $0^{c-1}1$. The remainder of s must be a string of length $n-c$ which has $k-(c-1)$ 0s, no c-runs, and does not begin with a $(c-1)$-run of 1s. Thus, we deduce that

$$V^{c,0}_{n,k} = U^c_{n-c,k+1-c} - V^{c,1}_{n-c,k+1-c}. \tag{8.2}$$

Analogous reasoning gives us the following recurrence for $V^{c,1}_{n,k}$:

$$V^{c,1}_{n,k} = U^c_{n-c,k-1} - V^{c,0}_{n-c,k-1}. \tag{8.3}$$

We next apply one of (8.2) and (8.3) to each term in (8.1) not of the form $U^c_{p,q}$, and then do so again to the resulting recurrence. This gives us

$$\begin{aligned} U^c_{n,k} =&(U^c_{n-1,k-1} - U^c_{n-c-1,k-c} + U^c_{n-2c-1,k-c-1} - V^{c,0}_{n-2c-1,k-c-1}) \\ &+ (U^c_{n-1,k} - U^c_{n-c-1,k-1} + U^c_{n-2c-1,k-c} - V^{c,1}_{n-2c-1,k-c}). \end{aligned} \tag{8.4}$$

We rearrange the terms of (8.4) to obtain

$$\begin{aligned} U^c_{n,k} =&U^c_{n-1,k-1} + U^c_{n-1,k} - U^c_{n-c-1,k-c} - U^c_{n-c-1,k-1} \\ &+ \left[(U^c_{n-2c-1,k-c-1} - V^{c,0}_{n-2c-1,k-c-1}) + (U^c_{n-2c-1,k-c} - V^{c,1}_{n-2c-1,k-c})\right]. \end{aligned} \tag{8.5}$$

From (8.1), we know that the portion of (8.5) in brackets is the recurrence for $U^c_{n-2c,k-c}$. Thus, we find that

$$U^c_{n,k} = U^c_{n-1,k-1} + U^c_{n-1,k} - U^c_{n-c-1,k-c} - U^c_{n-c-1,k-1} + U^c_{n-2c,k-c}. \tag{8.6}$$

Thus, we have reproduced the 5-term recurrence for $U^c_{n,k}$ found by Bloom (1996). Bloom derived this recurrence using a combinatorial argument, which is usually preferable to laborious algebraic derivations, such as ours. However, a generalization of our algebraic approach will allow us in chapter 9 to derive recurrences which are far too complicated to feasibly derive using combinatorial arguments.

We can complete a mathematical definition of the sequence $U^c_{n,k}$ by specifying its initial values. Strings of length less than c cannot contain any c-runs. Therefore, a string of length $n < c$ has k 0s and no c-runs if and only if it has exactly k 0s. Thus, we deduce that

$$U^c_{n,k} = \binom{n}{k} \tag{8.7}$$

for any $n < c$. Observe that this holds true even for $n < 0$.

Our initial values and recurrence relation complete our mathematical definition of $U^c_{n,k}$. Observe, however, that $U^c_{n,k}$ has not been expressed in terms of well-known and previously studied sequences, as S^c_n and U^c_n have been. The sequence $U^c_{n,k}$ remains uncountably finite. However, our recurrence and initial values allow us to speedily compute many values of $U^c_{n,k}$. Computing any one such term by brute force would require $O(2^n)$ operations, since it would be necessary to check each of the 2^n strings of length n to see whether it lacks c-runs and has the correct amount of 0s. On the other hand, computing all values of $U^c_{p,q}$ for $p, q \leq n$ requires $O(n^2)$ time when done using dynamic programming. To illustrate the profound difference in these algorithmic complexities, note that my computer stopped being able to compute terms by brute force at $n = 26$; it ran for about an hour trying to compute $U^6_{26,13}$ and ultimately failed because it ran out of memory. Using our recurrence relation and dynamic programming, however, my computer calculated $U^6_{p,q}$ for all $p, q \leq 4096$ in a matter of minutes.

The value $n = 2^{42}$ is not feasible even for an $O(n^2)$ algorithm, however, which is why I used $n = 4096$ instead. The value of c that I used was $c = \frac{1}{2} \log 4096 = 6$. Table 8.1 lists some of the values $U^6_{r,r/2}$ obtained using my code.

The message space reduction that results from enforcing both ESP and the absence of long runs is, once again, huge in absolute terms. For $n = 4000$, for example, the message space reduction is very roughly a factor of $10^{32} \approx 2^{106}$. Thus, we get a key length reduction from $|K_1| = 4000$ to about $|K_1| = 3894$.

However, these results are still not nearly strong enough. Even as n increases without bound, the key length reduction obtained by requiring ESP and the absence of 6-runs appears to stay roughly constant at 2.65%.

n	$U^6_{n,n/2}$	2^n
0	1	1
400	$1.43 \cdot 10^{116}$	$2.58 \cdot 10^{120}$
800	$2.72 \cdot 10^{233}$	$6.67 \cdot 10^{240}$
1200	$5.97 \cdot 10^{350}$	$1.72 \cdot 10^{361}$
1600	$1.39 \cdot 10^{468}$	$4.45 \cdot 10^{481}$
2000	$3.33 \cdot 10^{585}$	$1.15 \cdot 10^{602}$
2400	$8.17 \cdot 10^{702}$	$2.96 \cdot 10^{722}$
2800	$2.03 \cdot 10^{820}$	$7.66 \cdot 10^{842}$
3200	$5.10 \cdot 10^{937}$	$1.98 \cdot 10^{963}$
3600	$1.29 \cdot 10^{1055}$	$5.10 \cdot 10^{1083}$
4000	$3.29 \cdot 10^{1172}$	$1.32 \cdot 10^{1204}$

Table 8.1 Numerous values of $U^6_{n,n/2}$ obtained using dynamic programming.

Chapter 9

Generalizing ESP and “No Long Runs” to Bit Blocks

Our work thus far hinges on the assumption that a typical person’s plaintext data can be subtly modified (without changing the real content) so as to both have ESP and no long runs of the same character. This is because, although a typical sensible plaintext will be highly nonrandom, a sensible plaintext may nonetheless appear random to the naked eye when viewed as a sequence of bits. The value of any one bit can be safely treated as random, and repetition of the same pattern does not usually convey useful information, so it seems reasonable to require our message space to consist only of plaintexts with ESP and no long runs.

But while these requirements prohibit occurrences of strings such as 1^{1000}, they do not prohibit the string $(01)^{500}$—that is, 500 consecutive repetitions of the bit pattern 01. The string $(01)^{500}$ has ESP and no long runs, but, similar to 1^{1000}, it probably does not convey any meaningful information.

This observation suggests a possible generalization of the “no long runs” and ESP requirements. Instead of viewing a message as a sequence of n bits, we may instead view it as a sequence of $\frac{n}{2}$ blocks, where each block consists of two bits. Each bit has two possible values: 0 and 1. Each 2-bit block, on the other hand, has four possible values: 00, 01, 10, and 11. Rather than prohibiting c-runs of the same bit value, we can achieve a strictly greater effect by prohibiting $\frac{c}{2}$-runs of the same block value. Note that c-runs of 0s and 1s are still prohibited, since a c-run of 0s or 1s is equivalent to a $\frac{c}{2}$-run of 00 or 11 blocks. Our new requirement, however, also prohibits the repetitive strings $(01)^{c/2}$ and $(10)^{c/2}$.

The previous paragraph describes a way of generalizing the “no long runs” requirement. The even-split property, too, can be generalized using 2-bit blocks. If we make the reasonable assumption that any one 2-bit block in a plaintext message can be treated as random, then we can require the plaintexts in our message space to have equal numbers of the four possible block values 00, 01, 10, and 11. Any such plaintext will have ESP, but will also need to satisfy some additional requirements on its configuration of bits. Thus, our 2-generalized ESP is a strictly stronger requirement than ESP.

In fact, we can generalize further by viewing a plaintext message with n bits as a sequence of $\frac{n}{b}$ blocks, where each block consists of b bits. We refer to a block of b bits as a *b-block*. For fixed b, there are 2^b possible b-block values. Observe that $b = 8$ corresponds to us viewing a plaintext message as a sequence of bytes, since a byte consists of 8 bits.

In order to properly generalize ESP, we need the number of possible b-block values to be at most as large as the number of blocks in our message of length n. Otherwise, we know by the pigeonhole principle that some possible b-block value will never occur in any length n message, whereas others will, so that any such plaintext cannot possibly have generalized ESP. Thus, we require the following:

$$2^b \leq \frac{n}{b}. \tag{9.1}$$

This inequality is transcendental in b. It is not hard to solve, however, since b must be an integer. For $n = 2^{42}$, we find quickly by brute force that $b = 36$ is maximum. In practice, we will probably want to look at smaller blocks. However, it is good to know that block sizes can be fairly large while still being comfortably less than the strict upper bound given by (9.1).

We say that a plaintext with n bits has b-ESP if, when viewed as a length $\frac{n}{b}$ sequence of b-blocks, each possible b-block value occurs an equal number of times. The number of messages with n bits that have b-ESP is given by the multinomial coefficient

$$\binom{\frac{n}{b}}{\frac{n}{b2^b}, \ldots, \frac{n}{b2^b}} = \frac{\left(\frac{n}{b}\right)!}{\left[\left(\frac{n}{b2^b}\right)!\right]^{2^b}}. \tag{9.2}$$

We could use Stirling’s approximation to get a better sense for how this figure compares to 2^n, but doing so is messy and not particularly useful to us.

We refer to our generalized "no long runs" requirement as the "no long b-block runs" requirement. The number of messages of fixed length satisfying this generalized requirement is difficult to count and remains uncountably finite, even though it is possible to deduce recurrences and initial values for the corresponding sequence. If we let $U_n^{c,b}$ denote the number of 2^b-ary strings of length n containing no c-runs, then we can deduce that

$$U_n^{c,b} = 2^b U_{n-1}^{c,b} - (2^b - 1)U_{n-c}^{c,b}, \tag{9.3}$$

and that the initial values of the sequence are

$$U_n^{c,b} = 2^{bn} \tag{9.4}$$

for $n < c$. These initial values are easy to derive. We do not include the derivation for recurrence (9.3), but it can be derived by performing case work on the value of the first character and introducing a proxy sequence $V_n^{c,b}$, very similar to what we did in chapter 7.

Rather than expending further effort to count the numbers of strings satisfying our individual generalized properties, we instead focus our attention on the numbers of strings satisfying both generalized properties. Let $U_{k_1,\dots,k_{2^b}}^{c,b}$ be the number of 2^b-ary string which has no c-runs and which contains k_i of b-block value i, for each $i \in \{1, \dots, 2^b\}$. We also define proxy sequences $V_{k_1,\dots,k_{2^b}}^{c,b,i}$ for $i = 1, \dots, 2^b$. For each such i, let $V_{k_1,\dots,k_{2^b}}^{c,b,i}$ denote the number of 2^b-ary strings with no c-runs, which have exactly k_j of b-block value j for each j, and which begin with a $(c-1)$-run of the b-block value i.

By performing case work on the value of the first block in a string, we derive the recurrence

$$\begin{aligned} U_{k_1,\dots,k_{2^b}}^{c,b} = &\left(U_{k_1-1,\dots,k_{2^b}}^{c,b} - V_{k_1-1,\dots,k_{2^b}}^{c,b,1}\right) \\ &+ \dots \\ &+ \left(U_{k_1,\dots,k_{2^b}-1}^{c,b} - V_{k_1,\dots,k_{2^b}-1}^{c,b,2^b}\right). \end{aligned} \tag{9.5}$$

Using reasoning similar to that used in chapter 8, we also deduce that

$$\begin{aligned} V_{k_1,\dots,k_{2^b}}^{c,b,1} = &\left(U_{k_1-(c-1),k_2-1,\dots,k_{2^b}}^{c,b} - V_{k_1-(c-1),k_2-1,\dots,k_{2^b}}^{c,b,2}\right) \\ &+ \dots \\ &+ \left(U_{k_1-(c-1),k_2,\dots,k_{2^b}-1}^{c,b} - V_{k_1-(c-1),k_2,\dots,k_{2^b}-1}^{c,b,2^b}\right). \end{aligned} \tag{9.6}$$

The recurrences for the other $V^{c,b,i}_{k_1,\ldots,k_{2^b}}$ are analogous.

Although these recurrences are complicated, the initial values for our sequence are again easy to deduce. If $k_1, \ldots, k_{2^b}$ are all less than c, then there can be no c-runs. Thus, we have

$$U^{c,b}_{k_1,\ldots,k_{2^b}} = \binom{k_1 + \cdots + k_{2^b}}{k_1, \ldots, k_{2^b}} \tag{9.7}$$

whenever $k_1, \ldots, k_{2^b} < c$.

Similar to the way that we derived recurrences and used dynamic programming to compute values of $U^c_{n,k}$, it is possible to derive recurrences and use dynamic programming to compute values of $U^{c,b}_{k_1,\ldots,k_{2^b}}$. There are, however, several problems that arise in doing so. The first is that the derivation of the recurrences is very complicated—much more complicated than any problem which should be solved manually. Even if we were to make the illegal choice $b = \log 3$ so as to work with ternary strings, the resulting recurrence would have 16 terms, as opposed to the 5 terms in the recurrence for $U^c_{n,k}$. For 4-ary strings, the number of terms increases to 47.

To this end, I have written code that applies recurrences (9.5) and (9.6) repeatedly until all proxy sequence terms have been eliminated. Thus was I able to produce the 47-term recurrence relation for the sequence $U^{c,2}_{k_1,k_2,k_3,k_4}$.

The resulting recurrences and initial values from (9.7) can be used to "efficiently" compute values of $U^{c,b}_{k_1,\ldots,k_{2^b}}$. This application of dynamic programming is "efficient" in so far as the algorithmic complexity is polynomial in n for any choice of b. However, for the 4-ary case with $b = 2$, the algorithmic complexity is $O(n^4)$, as opposed to our $O(n^2)$ dynamic programming algorithm for $U^c_{n,k}$. In general, for fixed b, the algorithmic complexity of dynamic programming is $O\left(n^{2^b}\right)$, which becomes bad quickly—even $O(n^4)$ is not fast.

Thus, although the values of $U^{c,b}_{k_1,\ldots,k_{2^b}}$ can be computed much faster than brute force in this way, computer runtime and memory requirements are significant factors and cannot be ignored. I have used my dynamic programming code to compute various values of $U^6_{512,k}$ and $U^{3,2}_{k_1,k_2,k_3,k_4}$ when $k_1 + k_2 + k_3 + k_4 = 256$. The latter corresponds to viewing a 512-bit message as a length 256 sequence of 2-blocks. Some of the results are presented in table 9.1.

For $n = 512$, for example, $U^6_{n,n/2}$ would require roughly 495 bits to represent, whereas $U^{3,2}_{n/8,n/8,n/8,n/8}$ would require roughly 481 bits. These

n	$U^{6}_{n,n/2}$	$U^{3,2}_{n/8,n/8,n/8,n/8}$	2^n
0	1	1	1
64	$8.03 \cdot 10^{17}$	$3.15 \cdot 10^{16}$	$1.84 \cdot 10^{19}$
128	$3.53 \cdot 10^{36}$	$3.87 \cdot 10^{34}$	$3.40 \cdot 10^{38}$
192	$1.78 \cdot 10^{55}$	$7.12 \cdot 10^{52}$	$6.28 \cdot 10^{57}$
256	$9.49 \cdot 10^{73}$	$1.55 \cdot 10^{71}$	$1.16 \cdot 10^{77}$
320	$5.22 \cdot 10^{92}$	$3.70 \cdot 10^{89}$	$2.14 \cdot 10^{96}$
384	$2.93 \cdot 10^{111}$	$9.37 \cdot 10^{107}$	$3.94 \cdot 10^{115}$
448	$1.67 \cdot 10^{130}$	$2.47 \cdot 10^{126}$	$7.27 \cdot 10^{134}$
512	$9.61 \cdot 10^{148}$	$6.74 \cdot 10^{144}$	$1.34 \cdot 10^{154}$

Table 9.1 Numerous values of $U^{6}_{n,n/2}$ and $U^{3,2}_{n/8,n/8,n/8,n/8}$ obtained using dynamic programming.

figures correspond to key shortenings by 3.3% and 6.0%, respectively. This still leaves us with keys that are too long to be practical. This can be improved further by using even larger block sizes. It is not clear exactly how much of an improvement this would yield, because deducing values such as those in Table 9.1 for larger block sizes is extremely computationally intensive. Even if the required key length continues to be shortened linearly, however, the maximum block size $b = 36$ would result in a key that is roughly 97.8% shorter than the ones required for the message space $M = \{0, 1\}^n$. This key shortening is impressive, but for $n = 2^{42}$, the required key length is still impractically large at about 97 billion bits.

Chapter 10

Conclusion

This paper succeeds in developing a plausibly deniable encryption scheme for use with personal data storage. The primary method used is message space restriction. By limiting the possible plaintexts of a user to those which convey meaningful information, it becomes possible to map one plaintext to another (effectively hiding one plaintext within another) using a key shorter than those usually needed to safely use an XOR cipher.

The scheme developed here offers perfect deniability. Any plaintext in our message space M can be encrypted or decrypted to any other plaintext in M. Thus, a user's real plaintext can be hidden within any plausible plaintext of his or her choosing. It is even possible to hide multiple different real plaintexts within the same dummy plaintext, and this is true even if all of the plaintexts involved have no blank space and fill their containing hard drive. Thus, this scheme doubles as an effective means of data compression.

Our scheme also has several other desirable properties. For instance, it has the same provable security as the XOR cipher. As long as the steganographic key in our scheme is not used for any other purposes, an adversary will be unable to deduce anything about a user's real plaintext even if the first layer of encryption is broken to reveal the dummy plaintext. Nor will it even be possible for an adversary to conclude that a hidden plaintext exists.

Furthermore, our scheme is versatile. Though it was developed with personal data storage in mind, it can safely be used for communication between mutually trusting parties too. This is not one of the more practical uses for the scheme; it requires the two communicating parties to have at least as many shared secret bits as there are bits in the messages they wish to securely exchange. However, for purposes of communication, our scheme achieves the same security and perfect deniability as the XOR cipher and requires a lesser number of shared secret bits.

Even if we set aside questions of security and privacy, some of the combinatorial results of this paper can stand on their own. The results of chapters 7, 8, and 9 constitute a solution to a large generalization of problems considered by Bloom (1996) and Nyblom (2012).

With all of that said, our scheme in its current state faces difficulty when it comes to practical use. The XOR cipher allows for perfect deniability but is not practical because one of the encryption keys required for the scheme must be as long as the message itself. The scheme developed here shortens this necessary key length by potentially numerous orders of magnitude. However, this is not nearly enough; the key needed for this scheme needs to be shortened several more orders of magnitude for the scheme to be realistically usable.

Another important point is that even if a sufficiently small message space M is created for use with this scheme, we still need functions $f : M \to \{1, \ldots, |M|\}$ and $f^{-1} : \{1, \ldots, |M|\} \to M$ as well as efficient algorithms for computing both f and f^{-1}. The creation of algorithms for this purpose is disregarded here as an implementation concern. However, the fact remains that without such algorithms, this scheme cannot be used.

Yet another potential concern is whether all sensible plaintexts of fixed length can be subtly changed so as to have the even-split property and also satisfy the “no long runs” requirement. We have justified our assumption that this is possible and explained some methods that can be used to achieve this form in practice. However, if this is not always possible, then our scheme may fail for some sensible plaintexts.

Overall, our encryption scheme is not quite on the brink of being practical, but neither is it far from that point. Our work is an important step toward achieving practical private encryption with perfect deniability.

Chapter 11

Future Work

The key size reductions achieved using our methods are not quite enough for our scheme to be practical. There are several ways we could attempt to reduce the necessary key sizes further.

We mention in chapter 4 that a family $m_1, \ldots, m_r$ of messages might all be plausible plaintexts corresponding to the same information, differing only in the order in which they present different ideas or in their representations of blank space. Although the message space restriction techniques used in this paper will eliminate some of this redundancy, our resulting message space M likely still contains a high degree of redundant content. One potential method of shrinking our message space (and hence our key size) further is to cut down even more on this redundancy, ideally including in our message space exactly one message corresponding to each plausible collection of information.

Alternately or additionally, we could partition our message space M into disjoint message spaces $M_1, \ldots, M_k$ whose union is M. Our scheme can still be used with any one of these smaller message spaces as long as a user can indicate which specific message space they wish to use. Although specification of a smaller message space might amount to yet another key that counteracts the resulting smaller key size, it may also be able to hard code the index of the correct message space into one's computer or hard drive so that this additional key is stored elsewhere than a user's memory.

Although our modified XOR cipher that maps one plaintext to another has our message space M both as its message space and its ciphertext space, there may be benefits to using a different ciphertext space. For example, we might use a ciphertext space C that contains only plausible plaintexts and is small enough that

$$\frac{|M|}{|C|} = 2^d, \tag{11.1}$$

for some integer d. This way, each ciphertext in our ciphertext space can be mapped to by some 2^d different plaintexts in M even when the same key is used to encrypt all of them. The primary draw of this approach is that it shortens the necessary key length by d bits. This improvement may be nullified somewhat, since a user will need to store some other information in their memory in order to make their computer decrypt their ciphertext to the correct plaintext. However, this additional information will at most be d bits long, so that this ciphertext space shrinking is never detrimental. Furthermore, even if a user's effective key under this modified scheme is not actually shortened, d bits of an essentially random key are replaced by at most d bits of a user's choosing. Thus, longer keys may become more practical, since at least some of the key can be chosen by the user by virtue of being easy to remember.

One last promising direction for future research lies in work done by Juels and Ristenpart (2014). Juels and Ristenpart introduced *honey encryption*, which allows a ciphertext to be decrypted to numerous plausible, "bogus" plaintexts when certain incorrect decryption keys are used. The primary purpose of honey encryption is to make brute force attacks more difficult—an attacker will have to actually exhaust the key space in their attack, since there may yet be a hidden plaintext even if a sensible plaintext has already been found.

It is not a given that honey encryption can be used to make a good plausibly deniable encryption scheme. The authors explain in their paper's "Related Work" section that honey encryption schemes in general do not provide deniability. Even so, it is conceivable that their work can be modified to construct a deniable encryption scheme. After all, the core principle of being able to decrypt a ciphertext to multiple sensible plaintexts is the same.

Appendix A

Example Algorithm Construction

Given a message space M that we have constructed, an actual implementation of our encryption scheme would require a function $f : M \to \{1, \ldots, |M|\}$ as well as efficient algorithms to compute f and f^{-1}. We have not constructed such algorithms here, and instead have left those algorithms as implementation concerns. But because efficient algorithms for f and f^{-1} would be so crucial for anyone wishing to use our scheme, we use this appendix to demonstrate how, generally, such algorithms might be created.

We create specific algorithms in the case where M is chosen to be the set of messages of length n that have ESP. That is, we consider only messages with equal numbers of 0s and 1s. Recall that this message space M is still far too large for a scheme using M to be practical—the algorithms created here are purely illustrative.

We first demonstrate an efficient algorithm for computing binomial coefficients, since computation of binomial coefficients will be necessary for our algorithms. The factorial expansion of a binomial coefficient is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{A.1}$$

Observe that this is equivalent to the following:

$$\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1}. \tag{A.2}$$

Therefore, to compute $\binom{n}{k}$, we can compute $\frac{n}{k}$ and then recursively multiply this result by $\binom{n-1}{k-1}$. The number of recursive calls made by this algorithm

m	$f(m)$
0011	1
0101	2
0110	3
1001	4
1010	5
1100	6

Table A.1 The selected function f when $n = 4$.

is bounded by k, which can itself be forced to be $\frac{n}{2}$ or less. Therefore, this algorithm computes $\binom{n}{k}$ using $\mathcal{O}(n)$ operations. We are now ready to create our algorithms for f and f^{-1}.

A.1 Choosing and Efficiently Computing f

Note that any function $f : M \to \{1, \ldots, |M|\}$ is, in theory, usable for our scheme. We choose f as follows. Order the messages in M from least to greatest, with their sizes determined by their values when viewed as base 2 numbers. For a given $m \in M$, the value of $f(m)$ is then equal to the index of m within our ordering of M. Refer to Table A.1 for an example in the case where $n = 4$.

Given any $m \in M$, we need to be able to efficiently compute $f(m)$. Observe that this can be done by examining all binary strings of length n to count the strings with ESP that are "less" than m. However, this exhaustive search takes $\mathcal{O}(2^n)$ operations, which is not efficient.

Notably, though, we can efficiently compute $f(m)$ by efficiently counting the messages in M that are less than m. Specifically, we will add 1 to the number of messages in M that are less than m. This gives the index of m within M, hence giving us $f(m)$.

First, in $\mathcal{O}(n)$ time, we search the bits of m to determine the indices $a_1, \ldots, a_{n/2}$ of the 1s in m, where these indices are listed in order with a_1 corresponding to the leftmost 1. All messages in M with their first 1s appearing after index a_1 are less than m, so we now count these messages. The number of such messages is

$$\binom{n - a_1}{\frac{n}{2}}, \tag{A.3}$$

since the number of such messages is equal to the number of ways to distribute our $\frac{n}{2}$ 1s to the $n - a_1$ indices after index a_1.

Quantity A.3, however, is not necessarily the correct value for $f(m)$. It correctly counts the messages less than m that have their first 1s appearing after index a_1, but there may be messages that have their first 1s at index a_1 and yet are smaller than m. We must account for these messages too. To do so, we recurse on the substring of m beginning at index a_2, which is the second 1 in m. This substring has length $n - a_2 + 1$ and has $\frac{n}{2} - 1$ bits equal to 1. The number of binary strings with $\frac{n}{2} - 1$ bits equal to 1 that are smaller than this substring by virtue of having their first 1 appear later is

$$\binom{n - a_2}{\frac{n}{2} - 1}, \tag{A.4}$$

by reasoning analogous to that used to obtain A.3. Continuing our recursive calls, we obtain the following expression for $f(m)$:

$$f(m) = 1 + \sum_{i=1}^{n/2} \binom{n - a_i}{\frac{n}{2} - (i-1)} \tag{A.5}$$

The summation in A.5 has $\frac{n}{2}$ terms, each of which can be computed in $O(n)$ time using our binomial coefficient algorithm. Thus, this algorithm for finding $f(m)$ requires $O(n^2)$ operations.

A.2 Efficiently Computing f^{-1}

Our algorithm for computing m from $f(m)$ is slower but simpler than our algorithm for f. Given the value $f(m)$, we compute values of

$$\binom{n - a}{\frac{n}{2}}, \tag{A.6}$$

starting with $a = 1$ and ending no later than $a = n$. The index a_1 of the first 1 in m is then the largest value of a such that the computed binomial coefficient is less than $f(m)$.

After the appropriate index of the first 1 is found, we recurse using the new function value

$$f(m) - \binom{n - a_1}{\frac{n}{2}}, \tag{A.7}$$

since the first 1 appearing at index a_1 makes m automatically larger than $\binom{n-a_1}{n/2}$ other messages in M. To find the index a_2 of the second 1 in m, for instance, we now find the maximum value of a such that

$$\binom{n-a}{\frac{n}{2}-1} < f(m) - \binom{n-a_1}{\frac{n}{2}}. \tag{A.8}$$

The index a_2 is equal to this maximum value of a. We then recurse again using the new value

$$f(m) - \binom{n-a_1}{\frac{n}{2}} - \binom{n-a_2}{\frac{n}{2}-1}, \tag{A.9}$$

and continue until we have deduced the values of all of the indices $a_1, \ldots, a_{n/2}$. By learning the indices of all 1s in m, we uniquely determine m.

Observe that finding any one of the indices a_i using this algorithm requires the computation of at most n binomial coefficient, and each binomial coefficient computation takes $O(n)$ time. Calculating any one of the indices therefore requires $O(n^2)$ time. Since there are $\frac{n}{2}$ such indices, our algorithm for f^{-1} has runtime $O(n^3)$.

There are still many concerns that would arise in any implementation of our scheme, and it is likely that there are faster algorithms for the computation of our selected f and f^{-1}. Nonetheless, we have demonstrated generally how algorithms for f and f^{-1} could be created.

Bibliography

Bloom, David M. 1996. Probabilities of clumps in a binary sequence (and how to evaluate them without knowing a lot). *Mathematics Magazine* .

Canetti, Ran, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. 1996. Deniable encryption. Cryptology ePrint Archive, Report 1996/002.

Dresden, Gregory P. B., and Zhaohui Du. 2014. A simplified binet formula for k-generalized fibonacci numbers. *Journal of Integer Sequences* .

Dürmuth, Markus, and David Mandell Freeman. 2011. Deniable encryption with negligible detection probability: An interactive construction. Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology.

Greene, Tim. 2009. The history of steganography. Online; last viewed 27 April, 2015. URL http://www.networkworld.com/article/2870165/lan-wan/the-history-of-steganography.html.

Juels, Ari, and Thomas Ristenpart. 2014. Honey encryption: Security beyond the brute-force bound. Cryptology ePrint Archive, Report 2014/155.

Klonowski, Marek, Przemyslaw Kubiak, and Miroslaw Kutylowski. 2008. Practical deniable encryption. Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science.

Nyblom, M. A. 2012. Enumerating binary strings without r-runs of ones. *International Mathematical Forum* .

O'Neill, Adam, Chris Peikert, and Brent Waters. 2011. Bi-deniable public-key encryption. Cryptology ePrint Archive, Report 2011/352.

Y., Andrew. 2012. Hidden volume. Online; last viewed 26 March, 2015. URL http://www.andryou.com/truecrypt/docs/hidden-volume.php.