): ITU standard for public key infrastructure authorization and privige management.

Enc: Standard for encrypting XML documents.

3

Cryptography Primer

CONTENTS

3.1 Introduction

In Chapter 1 we discuss the history of cryptology, cryptography and cryptanalysis. *Cryptology* is the science of communicating using secret codes. It is subdivided into *cryptography*, writing in codes, and *cryptanalysis*, deciphering codes. As shown in Chapter 1, cryptography is of great historical, theoretical, and practical interest. It is an essential tool for developing secure systems.

This chapter provides a brief overview of cryptology. It does not deal with research issues in cryptology in any depth. It provides students with a basic understanding of what cryptology is, and is not. At the end, the student should have a practical understanding of how cryptography, and cryptanalysis work, and are used.

There is an apocryphal quote, which many attribute to Peter G. Neumann

of SRI:[1] "If you think cryptography is the answer to your problems, then you don't understand cryptography and you don't understand your problems." In the realm of security, there is no magic bullet or "pixie dust" that solves all our problems. That said, cryptography is an important tool. You need to understand what it is and how it is used. Just as important, you need to know what it is not and what its limitations are.

Cryptography consists of *encryption* and *decryption*. *Clear-text* is the original version of the message, which is easily read. *Cipher-text* is an obscured version of the message, that unauthorized people should not be able to convert back into clear-text. Clear-text and cipher-text can be expressed in different domains. Encryption maps clear-text to cipher-text, while decryption maps cipher-text to clear-text. The range of possible values for clear-text (cipher-text) is the *message space* (*cipher-space*). These two ranges need not be the same.

As shown in Figure 3.1 [358], encryption and decryption mappings are found by executing algorithms that take as input both a text message (in clear-text or cipher-text) as well as a *cryptographic key*. The encryption and decryption algorithms may, or may not, be identical. Similarly, encryption and decryption may, or may not, use the same cryptographic key. When encryption and decryption use the same key, the cipher is called a *symmetric key* cryptosystem. When the two keys are different, the cipher is called an *asymmetric key* or *public key* cryptosystem [287]. The range of possible values for keys is the *key space*.

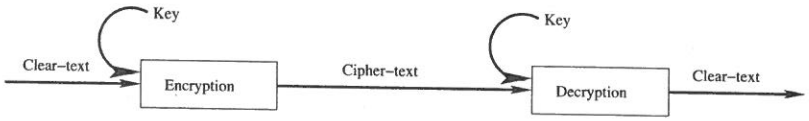

FIGURE 3.1
Illustration of encryption and decryption processes.

In general, cryptographic algorithms are not kept secret. It is felt that publishing cryptographic algorithms makes them more secure, since peer-review finds flaws in the algorithms. When flaws are known, they can be corrected. The use of secret algorithms is disparagingly called *security through obscurity*. Security through obscurity is found to produce less robust and secure systems. "Security should not depend on the secrecy of the design or the ignorance of the attacker" [439].

On the other hand, to keep the message secure some information must be hidden from potential attackers. The approach in Figure 3.1 shows the use of the cryptographic key, which is the one item that needs to be kept secret. Ideally, if the key is kept confidential, attackers can have free access to the cipher-text and be able to determine neither the clear-text message nor the value of the key.

There are several classes of cryptanalysis attacks that are used to attack cryptographic systems [358]. This classification depends mainly on the information (in addition to the cryptography algorithm) available to the cryptanalyst. The four classical classes of cryptanalysis are:

1. In *cipher-text only attacks* the cryptanalyst only has cipher-text. The analyst tries to find as many message cipher-texts as possible. If the crytpanalyst can discover the cryptographic keys, then they can convert all past and future cipher-texts into clear-text.
2. *Known clear-text attacks* are easier, since the analyst has access to some clear-text messages. In these attacks, analysts are interested in using these example clear-text to cipher-text mappings to derive either the cryptographic key or an algorithm for inferring the cryptographic key.
3. In *chosen clear-text attacks*, the analysts not only know some clear-text inputs, they can choose the clear-text inputs that will be encrypted. For example, a secret document could be leaked in order to perform this attack. The analyst's goals are the same as with the known clear-text attack.
4. In *adaptive chosen clear-text attacks* analysts can also modify the choice of clear-text to be used based on results from analysis of previous chosen clear-text attacks.

We will not discuss in detail these less typical classes of cryptanalysis [358]:

1. In *chosen cipher-text attacks* the analyst can choose the cipher-text to be decrypted. This may be useful in black-box analysis of an encryption device.
2. A *chosen-key attack* does not mean that the analyst gets to choose keys, rather it occurs when the analyst gathers some knowledge about relationships between keys. For example, if an initialization vector is used as a seed for a pseudo-random number generator.
3. *Lead-pipe cryptanalysis*[2] [296, 390, 391] occurs when pressure is applied to humans to force them to reveal a cryptographic key. This may include force, bribery, social engineering, etc.

A survey of banking systems [47] shows that most civilian cryptosystems do not fail due to sophisticated cryptanalysis. Rather, they fail due to either implementation errors or management mistakes. In practical terms, the threat model commonly used for civilian cryptographic systems is not the most realistic one. Social engineering and lead-pipe cryptanalysis are more common attacks than sophisticated mathematical analysis. For military and diplomatic applications, the situation is likely to be different.

[1] I heard him attribute the quote to others with the caveat that they in turn attribute it to still other people. The quote is widely accepted, but of apparently unknown origin.

[2] This can also be called rubber hose cryptanalysis.

In the rest of this chapter[3], we first briefly discuss the main classes of cryptographic algorithms. We discuss each algorithm and mention known problems. We also discuss cryptanalysis of the substitution and Vigenère ciphers. This illustrates the general ideas behind cryptanalysis without being too onerous. Cryptanalysis of block ciphers and public key cryptography is outside the scope of this book. We then explain one-time pads; why they are secure; and why they are not very practical for most applications. This leads naturally into an introduction to key management. This is followed by a discussion of how to best verify the correctness of security algorithms. The chapter ends with discussions of steganography, obfuscation, and homomorphic encryption. Although steganography and obfuscation are not encryption, we will see that they have different assumptions, but are used for similar applications. Both steganography and obfuscation can be considered examples of security through obscurity. Homomorphic encryption and obfuscation are discussed together, in spite of the fact that one is encryption and the other is not, mainly because their application domains are very similar.

3.2 Substitution Ciphers and Frequency Analysis

The simplest cipher is a substitution cipher. This cipher replaces each letter of the alphabet with another letter. In the Caesar cipher, each letter is shifted three to the right. The letter "a" becomes "d," and "b" becomes "e." The end of the alphabet wraps around, with "x" mapping to "a'," "y" to "b," and "z" to "c" [358, 388]. If we number the letters from 1 to 26, this can be expressed mathematically as a mapping from the input character a_i to the cipher character a_c using modulo 26[4] addition:

$$a_c = a_i + 3(mod 26) \tag{3.1}$$

If both characters are ASCII values, this can be written in the C programming language as:

```
char CaesarCipher(char input)
{
 char answer;
 if((input > 64) && (input < 91)){          /* Capital letters */
   answer = (((input - 64) + 3) % 26 ) + 64);
 }else if((input > 96) && (input < 122)){ /* Lower case   */
   answer = (((input - 96) + 3) % 26 ) + 96);
 }else answer = input;                          /* Non-alphabetic*/
 return( input );
}
```

The ROT13 cipher is another cipher of this class, where each letter is shifte 13 positions:

$$a_c = a_i + 13(mod 26) \tag{3.2}$$

Verify that applying ROT13 a second time to a message gives you the origin clear-text. A slight generalization of these two ciphers could use an arbitrar key k, with $k <= 26$. This would be a symmetric key encryption approac where encryption is:

$$a_c = a_i + k(mod 26) \tag{3.3}$$

and decryption is:

$$a_i = a_c - k(mod 26) \tag{3.4}$$

Cryptanalysis of this approach is easy. Since there are only 26 possible key it would be possible to write a program that uses all possible key value and compare the results with words in file /usr/share/dict/linux.words. If th input message is in English, the key could be found in seconds.

A more general encryption algorithm could use any one-to-one mappin from (to) a clear-text symbol to (from) a cipher-text symbol. For Englis instead of there being 26 possible keys, there would be 26! (approximatel $4 * 10^{26}$) possible keys [287]. An exhaustive search of this key space is mo expensive. This kind of simple substitution cipher that uses a single alphab when encrypting (decrypting) is called a *mono-alphabetic* cipher.

Mono-alphabetic ciphers are typically broken using frequency analy sis [388]. We compute the frequency of each character in the cipher-text an compare them to the frequencies for a typical English text. In most cases, fc reasonably long texts, this comparison is enough to provide us with the ke with no further calculation[5]. This saves us from having to perform a brut force, exhaustive search of the $4 * 10^{26}$ alternatives in the key space.

This cipher was secure for Caesar not because of its complexity. It wa secure in large part because many people were illiterate and unable to rea clear-text.

[3]We suggest that both instructor and students use cryptool software when studying this chapter [19]. Its visualizations of cryptography algorithms are useful in understanding the approaches. In addition, Cryptool's implementation of cryptography algorithms and cryptanalysis tools help teachers and students create experiments.

[4]Modulo arithmetic is simply normal arithmetic where there is a maximum number. For example, in arithmetic modulo n all multiples of n are set to zero and any numbers larger than n are set to their remainder when divided by n. For example, in modulo 8 arithmetic 10 is equal to 2, as is 18.

[5]This could be problematic if the clear-text message was not representative of the la guage as a whole. For example, Georges Perec's 300-page French novel *La disparition* an Ernest Vincent Wright's English novel *Gadsby* were both written without a single use the letter "e", the most common letter in both languages.

3.3 Vigenère Cipher and Cryptanalysis

We now describe Blaise de Vigenère's cipher from 1586 that we mention in Chapter 1. The Vigenère cipher is an example of a *poly-alphabetic* cipher where multiple alphabets are used.

To understand the cipher, consider Table 3.1 [388]. In Vigenère's approach, the key is a word or phrase. For a simple example, let's use the phrase *goldbug* as a key. We use this key to encrypt the message *EdgarAllenPoe*.

The message is encrypted letter by letter. We consider each character of the message and the key in turn. If the message is longer than the key, which is usually the case, we cycle through the key as many times as necessary.

Each letter in the message is replaced with the element in Table 3.1 corresponding to the column of the current key character and the row corresponding to the current message character. In our example, the first character of the encrypted message is k which is the character in column g and row e. The second character of the encrypted message is r, found at column o and row d. By repeating this simple process we encrypt the entire message, giving *KrrdsUrrsySpy*. Decryption is performed by simply looking in the column specified by the key value at each step; finding the letter in the encrypted message; and replacing that letter with the letter corresponding to that row.

This can also be expressed mathematically [287]. For a message of length i, a key of length t, a clear-text sequence of characters $a_0, a_1, \ldots, a_{i-1}$, the encrypted text sequence of characters $c_0, c_1, \ldots, c_{i-1}$, and key sequence of characters $k_0, k_i, \ldots, k_t$, we have Equation 3.5.

$$c_i = a_i + k_{imodt}(mod26) \tag{3.5}$$

Cryptanalysis of poly-alphabetic ciphers is done by analyzing the cipher-text and finding ways to divide the problem into a set of mono-alphabetic ciphers.

To illustrate this, we downloaded a copy of Dostoevsky's *The Brothers Karamazov* from Gutenberg.org and encrypted it using the key *CRYPTONOMICON*. Figure 3.2 shows the auto-correlation of the resulting cipher-text[6]. We note the clear peaks at every 13 characters, which shows us that the key used in the Vigenère cipher is 13 characters long.

The symbol frequencies that were used to break the Caesar cipher cause these peaks to occur, since the same symbols occur more frequently when the key vlaues are the same. We can then separate the cipher-text into 13 unrelated Caesar ciphers. Solving them separately allows us to find the original key, which we use to find the clear-text message.

[6]This figure was produced using the cryptool software suite [19].

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
b	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a
c	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b
d	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c
e	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d
f	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e
g	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f
h	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g
i	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h
j	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i
k	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j
l	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k
m	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l
n	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m
o	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n
p	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
q	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
r	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
s	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
t	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
u	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
v	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
w	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	u
x	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	u	w
y	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	u	w	x
z	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	u	w	x	y

TABLE 3.1
Vigenère table.

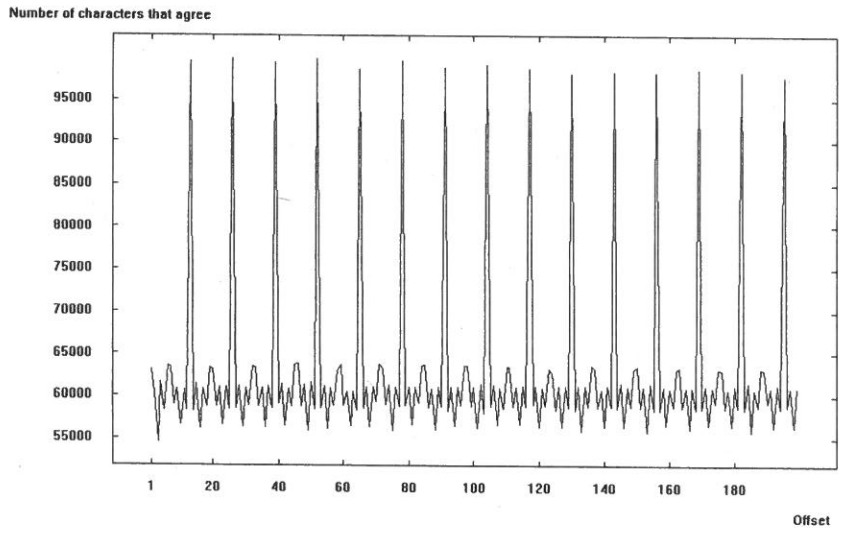

FIGURE 3.2
Example auto-correlation of Vigenère encrypted message.

3.4 Block Ciphers

The main weakness of both mono-alphabetic and poly-alphabetic ciphers is the one-to-one correspondence of clear-text characters to cipher-text characters. This gives cryptanalysts information to exploit. To overcome this, block ciphers encrypt and decrypt blocks of n bits at a time.

Most currently used symmetric key cryptography algorithms are block ciphers. The advent of the Data Encryption Standard (DES) block cipher is widely viewed as the birth of modern day cryptography. Today, as we will discuss, DES is no longer considered adequately strong. It has been replaced with another block cipher, the Advanced Encryption Standard (AES).

This section provides background information followed by summary descriptions of DES and AES. It ends with a brief discussion of cryptanalysis for block ciphers. Our goal is to provide a brief introduction to these tools, so the reader can have a general understanding of how they work. People interested in implementing, improving, or attacking block ciphers need a more in-depth understanding and should refer to our references.

3.4.1 Operations

Block ciphers encrypt clear-text messages by processing them in sequential chunks of n bits. Typically, n is a power of 2. DES uses a block size of 64 bits. AES uses 128-bit blocks [388]. Block sizes smaller than 64 bits would rarely be used, since the cipher would not be significantly different than a cipher processing one character at a time.

Symmetric key ciphers are typically built by combining the following atomic operations [287, 358, 388]:

- Boolean logic – The following operations will be used on bits, bytes, or words: AND, OR, XOR, NOT, NAND, and NOR.[7]
- Substitution[8] – Replacement of one value by another value. This may be specified as a vector. For example, if 0 is to be replaced by 2, 1 replaced by 0, and 2 replaced by 1, the vector would be $[2, 0, 1]$. We note that DES uses least significant bit order in this operation [388].
- Permutation[9] – Changing the bit order of the data. This may also be specified as a vector. For example, if bit 0 is to be put in bit 2, bit 1 placed in bit 0, and bit 2 moved to bit 1, the vector would be $[2, 0, 1]$.
- Shift registers – A set of n registers collected to a common clock controlling the movement of data between, into, and out of the registers [194]. The path of movement between the registers is fixed. New data values of registers may include functions of the contents of other registers. *Linear feedback shift registers* are a common way of producing sequences of pseudo-random numbers.

Symmetric ciphers tend to be a networked combination of these operations. The symmetric key is used to parametrize these operations for transforming clear-text into cipher-text. It is essential that there is an efficient way of computing the inverse of the resulting encryption algorithm.

Feistel structures are a common way of networking these atomic operations [194, 287, 358, 388]. The clear-test input is divided into a sequence of blocks. If the blocks are not full, the last block will be padded. Each input block is divided into a left half L_0 and right half R_0. The Feistel network defines a number of computation rounds. Each round i has the same structure and an associated key K_i. Equation 3.6 gives the computation at each round, where $\oplus$ is the XOR function and $f()$ is a known function repeated at each round.

$$(L_{i-1}, R_{i-1}) \Rightarrow (R_{i-1}, L_{i-1} \oplus f(k_i, R_{i-1})) \tag{3.6}$$

Figure 3.3 is a flowchart for Equation 3.6.

[7] We assume the reader understands these operations, if not refer to [388]

[8] The component that performs a substitution operation is referred to as an *S-box* [358, 388]

[9] The component that performs a permutation operation is referred to as a *P-box* [358, 388]

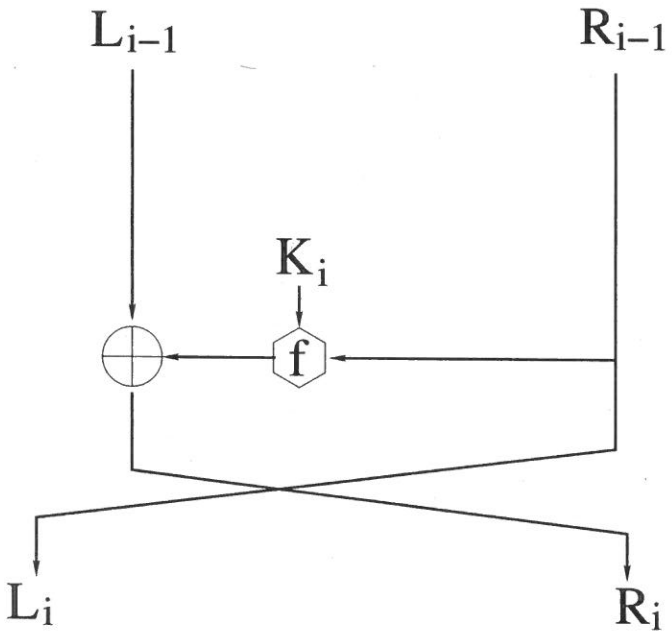

FIGURE 3.3
Diagram illustrating one round of a Feistel structure.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bits	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

TABLE 3.2
Number of key bits shifted at each round of DES.

3.4.2 Data Encryption Standard

The Data Encryption Standard (DES) is a block cipher that uses a 56-bit key K [358]. Any 56-bit number can be a key. There are a small number of known bad keys that must not be used. The key is usually expressed in 64 bits, but every eighth bit is a parity check and ignored by the computations.

The first step in the DES algorithm is a data permutation. The last step is the inverse of this permutation. Since this step has no effect on the security of the algorithm and is frequently skipped by software implementations [358], we ignore it here. Interested parties can refer to [358].

DES has 16 rounds. Each round i uses a different 48-bit sub-key k_i where $1 \leq i \leq 16$. To calculate k_i, k_{i-1} is divided into two 28-bit halves and each half is circularly shifted by 1 or 2 bits each round, following Table 3.2.

After the shift operation, a permutation operation (called the *compressio permutation*) is performed using 48 bits from the shifted key. Because of th rotation operation, a different subset of the bits is used at each round [358 287]. Another permutation operation (called the *expansion permutation*) take 32 bits from the right half of the data and writes a 48-bit output.[10] Th compressed key is XOR-ed with the output of the expansion permutation These processes create a different key for each round and ensure that th dependency of output bits on input bits spread quickly.

The 48 bits resulting from the XOR of the outputs from the compressio and expansion permutations are divided into eight sets of six bits. Thes six bits are input to eight separate *S-Box* circuits. Each S-Box inputs 6-bit and outputs 4-bits. They can be implemented as table look-ups. DES ha eight unique S-Box circuits. The S-Box outputs are concatenated into a singl 32-bit block. The S-Box step is non-linear, difficult to analyze, and largel responsible for DES's security.

The 32-bit block output from the S-Boxes is subjected to a simple per mutation operation using a *P-Box* circuit. The results are then XOR-ed witl the left half (32 bits) of the data for this round. This result becomes the righ half for the next round. The unmodified right half of the data for this roun becomes the left half of the data for the next round. Note that this is a Feiste structure as shown in Figure 3.3.

DES decryption is done using exactly the same algorithm as encryption The only change is that the round keys are used in reverse order.

3.4.3 Advanced Encryption Standard

The Advanced Encryption Standard (AES) was chosen by NIST to be th successor to DES in 2001 [318]. Some of the reasons for this update will b discussed in Section 3.4.5. The original name for AES was Rijndael, whic comes from the names of the two inventors. AES uses 128 bit blocks an supports key sizes of 128, 192, and 256 [388]. For brevity in this book, we onl discuss 128 bit key AES[11].

AES starts processing each 128-bit byte by creating a *state* matrix, a shown in Figure 3.4. Each element $S(i, j)$ in the state matrix is an 8-bit byte Each row of the matrix is a 32-bit word [318, 388].

AES has four basic operations [318, 388]:

1. *SubBytes* – Each element of the state is run through an 8-bit S-Box, which is usually implemented as a look-up table. This means each

[10] Tables of the bit positions used in these permutations will not be provided, becaus this level of detail is not relevant for our discussion. They are easily accessed, either fror references [358, 287] or on-line at http://www.itl.nist.gov/fipspubs/fip46-2.htm. A sampl implementation of DES can be found at http://www.cryptool.org. The visualization prc vided by the cryptool package is useful for understanding DES.

[11] We note that cryptool [19] has an AES implementation for reference and test. It, ur fortunately, does not have a visualization

S(0,0)	S(0,1)	S(0,2)	S(0,3)
S(1,0)	S(1,1)	S(1,2)	S(1,3)
S(2,0)	S(2,1)	S(2,2)	S(2,3)
S(3,0)	S(3,1)	S(3,2)	S(3,3)

FIGURE 3.4
Row and column byte ordering for the AES state.

byte in the state is replaced with another byte value. See [318] for a table giving the replacement values in hexadecimal.

2. *ShiftRows* – The bytes in the last three rows are rotated to the left. The first row is not changed. The second row is rotated one byte to the left; the third row is rotated two bytes; and the fourth row is rotated three bytes.
3. *MixColumns* – Operates on the state column by column. In essence, the matrix multiplication in Figure 3.5 is performed where the operation $\otimes$ is *binary XOR long division modulo 128*. Refer to [318] for details on implementing binary XOR long division modulo 128.
4. *AddRoundKey* – XOR the state with the round key.

$$\begin{bmatrix} S(0,i) \\ S(1,i) \\ S(2,i) \\ S(3,i) \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \otimes \begin{bmatrix} S(0,i) \\ S(1,i) \\ S(2,i) \\ S(3,i) \end{bmatrix}$$

FIGURE 3.5
Matrix algebra representation of the AES MixColumns operation.

When using a 128-bit key, AES has 10 rounds. The AES algorithm is [318]:

1. Do AddRoundKey
2. For each round, except the last:
   - •Do SubBytes
   - •Do ShiftRows
   - •Do MixColumns
   - •Do AddRoundKey
3. Do SubBytes
4. Do ShiftRows
5. Do AddRoundKey

Each round uses a different key. When AES uses a 128-bit key, it has 10

Round	1	2	3	4	5	6	7	8	9	10
Bits	1	2	4	8	16	32	64	128	27	54

TABLE 3.3
Value of first byte of round constant for AES key schedule.

rounds. The 128-bit key is four 32-bit words long. The key scheduler has to generate 47 words to satisfy the algorithm. The first four words generated by the AES key scheduling algorithm are simply the original key. At which point, the following algorithm is performed [318]:

1. Write the last word output by the key scheduler into a temporary variable t.
2. Rotate the bytes of t one byte to the left and overwrite t.
3. Send each byte of t through an S-Box and store in t.
4. XOR the temp variable with a four-byte round constant whose last 3 bytes are zero. The first byte is 2^i modulo(283); these values are listed in Table 3.3.
5. XOR this temp variable with the first word in the last iteration of this algorithm and write this as one of the words to use as a round key.
6. For each of the three other words output in the last iteration of the algorithm, XOR them with the word just output as a round key and output them as the next word to use as a round key.
7. Go back to step 1 until 47 round key words have been output.

AES decryption is similar to DES encryption. The operations are done in reverse order, with a reverse key schedule, and inverse operations for SubBytes, MixColumns, and ShiftRows [388].

3.4.4 ECB and CBC modes

In practice, block ciphers are implemented using one of two modes:

- Electronic code book *ECB* – In ECB mode, each block's cipher-text is calculated by applying the encryption algorithm directly to the clear-text using the cryptographic key. Similarly, the clear-text is recovered from the cipher-text by executing the decryption algorithm with the cryptographic key on the cipher-text one block at a time.
- Cipher block chaining *CBC* - To use CBC mode, both sides need to start with a common *initialization vector* (IV) [12]. The IV is a random bit string as

[12]The IV is sometimes referred to as *salt*

long as a single block. The first block is XOR-ed with the IV before encrypting. Each subsequent block is XOR-ed with the cipher-text of the previous block before encrypting. To decrypt the stream of blocks, the first block of cipher-text is decrypted and then XOR-ed with the IV. Each subsequent block is decrypted and then XOR-ed with the cipher-text of the previous block.

ECB mode is susceptible to replay attacks, where the attacker records cipher-text for use at a later time. For some applications replay attacks are possible, because the application will see that the cipher-text was encrypted with the proper key. Note that the attacker does not need to perform any cryptanalysis for this to work. The use of the IV (salt) makes replay attacks infeasible, as long as both sides use different IV values each time.

3.4.5 Cryptanalysis

In Section 3.1, we list the classes of cryptanalysis. Cipher-text-only attacks are the most damaging attacks to cryptographic protocols.

The simplest attack, conceptually, is a brute force attack, where the attacker tries different possible key combinations until they find one that correctly decrypts the cipher-text. For a key of n-bits, there will be 2^n possible keys. If all keys are equally likely, then there is a 50 percent chance that a brute-force attack will discover the key within 2^{n-1} guesses. Note that each bit added to the key doubles the number of possible keys and doubles the time needed to guess the solution.

It was vulnerability to brute-force search that forced NIST to replace DES with AES. DES was designed with a 56-bit key-length. A custom hardware design was published in 1993 which could find a DES key from known cipher-text and clear-text in 3.5 hours on the average (7 hours worst case) at a cost of around one million dollars. In 1998, the Electronic Frontier Foundation (EFF) published a book detailing a custom hardware and software design that would be able to find the DES key using 8-byte clear-text and cipher-text samples within one week at a cost of around $ 200,000 [163]. The book detailed how to build low-cost devices tailored to quickly performing DES operations so that they could be used in parallel to find the DES key used to encrypt the data. After publication of this document, NIST opened competition to find another cryptography standard. This approach assumed a small amount of known clear-text but if the attacker has some knowledge of the format of the clear-text this approach would still be feasible. It would, however, require more cipher-text samples.

Another feasible attack on DES is *differential cryptanalysis* [388, 71]. The concept behind differential cryptanalysis is that differences in clear-text inputs will cause differences in cipher-text outputs. This is a chosen clear-text attack. Picking inputs correctly allows the attacker to target paths through specific S-boxes. Running a large number of samples through the encryption algorithm

allows the attacker to derive a probability distribution relating input differences to output differences. See [388] for a fuller treatment of how differential cryptanalysis works and [71] for application of differential cryptanalysis to DES. They analyze 2^{36} cypher-texts in 2^{37} time.

Linear cryptanalysis has also been successfully used against DES. It is similar conceptually to differential cryptanalysis [388, 284]. Linear cryptanalysis assumes a linear expression can describe a relationship mapping bits of clear-text and cipher-text data to individual key-bits. Large sets of clear-text are input to the algorithm with different keys and the resulting cipher-text collected. Clear-text and cipher-text combinations are mapped to individual key-bits. Probability density functions are calculated. If the probability differs from 0.5 then the data provides information about the key. In the attack, cipher and clear-text pairs are input to the probability density functions to calculate the probability of different bit combinations for the key. The most likely key is chosen. The work in [284] shows DES can be broken using 2^{45} random plain-text inputs.

It is also possible to combine linear and differential cryptanalysis to aid in inferring keys. One can also do "higher-order" differential cryptanalysis by looking at the differences between differences in cipher-text. All of these approaches can be useful in cryptanalysis of block-ciphers [388].

In addition, all cryptographic approaches are vulnerable to *side-channel attacks*. A side-channel attack takes advantage of the fact that algorithms are implemented on physical machines. Machine instructions will not have identical time, power, space, etc. needs. Measuring these environmental interactions provides the attacker with enough information to break the basic assumptions behind the cryptographic protocols. We discuss side-channels in depth in Chapter 12.

The design of AES took advantage of lessons learned when analyzing DES. AES is very resistant to linear and differential attacks [139]. In spite of this, some possible vulnerabilities may exist in AES. Structural attacks can leverage the flow of information specified by the cipher. For AES, there is structural information that is predictable every three rounds. Algebraic attacks are also possible in theory that create sets of quadratic equations over the Galois field 2^8, which could be solved to infer some key bits using chosen text attacks [139].

AES has been shown to be vulnerable to the injection of physical faults. Power spikes, errors in clock timing, and electromagnetic radiation can all either modify the contents of memory or code execution causing the AES computations to be incorrect. For example if the attacker can set key bit i to zero (one) and observe that the other side accepts (rejects) the resulting cipher-text, then the attacker knows that the correct value of key bit i is zero. These ideas are used in [78, 175] to extract 128-bit AES keys. These attacks are typically countered by physically hardening cryptographic devices. A related concept is the idea of *related key attacks*, where the attackers can inject perturbations on the key and see how that changes the cipher-text. Some researchers have found related key vulnerabilities in AES [75].

3.5 RSA Public Key Cryptography

In public key cryptography the key used to encrypt data is different from the one used to decrypt the cipher-text. This asymmetry makes it easier to maintain system security, since users need only maintain the secrecy of their *private key* and can freely distribute the *public key* without fear of compromising system security. Since public key approaches are typically more expensive computationally than symmetric key cryptography, it is common to use the public key approach to securely exchange a random, temporary, session key that is used by a symmetric key algorithm, like AES, to encrypt/decrypt the data exchange [358].

This section will not provide mathematical details. We provide only brief, higher-level descriptions of the important public key encryption algorithm RSA. Elliptic curve cryptography is also important, but will not be presented, since it would require more mathematical detail than appropriate for this book.

The RSA algorithm's name comes from its inventors: Rivest, Shamir and Adleman. It is based on the difficulty of factoring large numbers [358, 287]. First find two large prime numbers p and q of equal length. Find the product n:

$$n = pq \tag{3.7}$$

Find a number e that is relatively prime to $(p-1)(q-1)$ and $1 < e < (p-1)(q-1)$. The encryption key is e. The decryption key is a number d where:

$$ed = 1 mod (p-1)(q-1) \tag{3.8}$$

which means that

$$d = e^{-1} mod (p-1)(q-1) \tag{3.9}$$

The public key is n and e. The private key is d.

To encrypt a message m:

1. Divide m into j blocks m_j of size 2^i where i is an integer and $2^i < n$.
2. The cipher-text block c_j is set to $c_j = m_j^e mod n$.

Decryption is simply:

$$m_j = c_j^d mod n \tag{3.10}$$

One way to break RSA would be to factor n directly [358, 287], but as long as n is large there are no efficient factoring algorithms. Current suggestions are to use keys of at least 1024 bits [249].

There are some known weaknesses in RSA:

- If similar messages may be sent over time, then it is inadvisable to use a small value for e [287],
- Choose d to be approximately the same size as n [287]
- For small messages, pad the message with random bits [287],
- It is possible for a message to have the same clear-text and cipher-text values. This should be avoided [287],
- Chosen cipher-text attacks can be used to trick users into mistakenly decoding messages [358],
- Re-use of the value n makes it easier to infer d [201],
- RSA is sensitive to related plain-text attacks [201], and
- Like symmetric key algorithms, public key encryption is subject to side channel attacks [238]; see Chapter 12.

A thorough discussion of RSA weaknesses can be found in [201]. Issues related to hardware implementations of public key systems can be found in [63].

It is worth noting that for a public key approach like RSA to provide the same security as a symmetric key approach like AES, the public key approach typically requires a significantly larger key [358]. This is because constraints on the possible public key values reduce the size of the key space for a given key length.

3.6 Hash Functions

A hash function $h = H(M)$ is a mapping of a binary value M to a binary value h, where h has fixed length m. Common examples of hash functions are parity bits and cyclical redundancy checks. The use of hash functions in cryptography is similar to their use for error detection and correction [194]. Cryptographic hash functions are commonly referred to as *one-way hash functions*. A one-way hash function should have the following attributes [358]:

1. Easy to compute (find h given M),
2. Hard to invert (find M given h), and
3. Difficult to create collisions (find M' given M where $H(M') = H(M)$).

It is worth noting that both parity bits and cyclic redundancy checks are not hard to invert. They are, therefore, also prone to collisions.

One important issue in designing a hash function is finding the right value for m. The third property implies m should be large enough that different values of M are unlikely to have the same value. Recall the birthday paradox [424]. The probability that another person has the same birthday as you is $1/365$. The probability that they have a different birthday is $(365-1)/365$ Let's replace 365 with variable d. This lets us express the probability Q_1 that n people all have different birthdays as:

$$Q_1(n,d) = \frac{d!}{(d-n)!\,d^n} \tag{3.11}$$

So the probability $P_2(n,m)$ that there will be at least one collision among n messages M mapped to a hash function of size m is:

$$P_2(n,m) = 1 - Q_1(n,m) = 1 - \frac{m!}{(m-n)!m^n} \tag{3.12}$$

This can be generalized as the *birthday attack*, which states that the expected number of random trials needed to construct a collision to a hash function of range m is:

$$1.2\sqrt{m} \tag{3.13}$$

The two main classes of cryptographic hash functions are [287]:

- *Modification detection codes* (MDCs) – are un-keyed hash functions that are used as *fingerprints* to detect tampering. These are often constructed from block ciphers. NIST is currently evaluating proposals for a new family of cryptographic hash functions, since weaknesses have been found in SHA-1. Current advice is to use SHA-256 or SHA-512.
- *Message authentication codes* (MACs) – are keyed hash functions. They are used both to verify that the data has not been tampered with and the data source. MACs are often constructed from MDCs. MACs use a shared key for authorization, so they are not suitable for non-repudiation.

We will not discuss implementation details, since implementations of standard cryptographic hash functions can be found in Cryptool [19] and OpenSSL [22].

If non-repudiation is needed, then it is possible to use public key cryptography to encrypt a MDC value with the secret key.

3.7 One-time Pads

There is one theoretically secure, and totally impractical, cryptography algorithm. A one-time pad encrypts a message M of length m by XORing it with its key K. The key K is a random bit-string of length m with each bit set to 1 with probability 1/2. K is used only once [194]. This approach goes back to Shannon and is perfectly secure against cipher-text only attacks, since all keys (and therefore cipher-texts) are equally likely. The cipher-text gives no information about the clear-text message, unless the attacker knows K.

This system loses its security, should K be re-used. The main drawback is the need to distribute the keys. You would need a secure, secret channel for distributing information with sufficient bandwidth to handle messages of size m. If you were to have that channel, it would be wise to transmit M directly.

3.8 Key Management

The issue with one-time pads is the difficulty of securely distributing key values. This is a general problem, which is quite different for symmetric and asymmetric key cryptography. We look at each in turn. The primary issue is how to transmit keys between hosts using untrusted channels, without compromising them.

3.8.1 Notation and Communicating Sequential Processes (CSP)

To describe protocols, we use notation from Ryan and Schneider [351]. The notation is based on Hoare's Communicating Sequential Processes (CSP) [203] as modified to create Abadi's [35] *spi calculus* that represents cryptographic protocols. This is a straightforward way of logically representing logic, message passing, cryptographic operations, and parallel operations.

For example, a sending b a message that contains a data element with session key k_{ab} concatenated with a's address, where the data element has been encrypted with key k_b, is represented by:

$$a \rightarrow b : \{k_{ab} \bullet a\}_{k_b} \tag{3.14}$$

CSP can be used to automatically check these protocols for deadlock and similar errors [203]. Theorem proving tools can be used to verify the soundness of protocols against a large number of attack scenarios [35, 351].

3.8.2 Symmetric key distribution

It is possible for two hosts (a,b) to securely establish a key over an open channel [287]. *Shamir's no-key protocol* starts with a and b publishing a prime number p. Node a and b choose random numbers r_a and r_b, respectively. Node a initiates the following protocol to send session key K to b:

- $a \rightarrow b : K^{r_a} mod p$
- $b \rightarrow a : (K^{r_a})^{r_b} mod p$
- $a \rightarrow b : (K^{r_a r_b})^{r_a^{-1}} mod p$

At which point, b exponentiates the last message by r_b^{-1} and retrieves K. Note that the protocol we present does not authenticate the hosts involved and only protects against passive eavesdropping. It is vulnerable to man-in-the-middle attacks.

A major problem with this approach is the lack of authentication. If the two nodes a and b do not already share a key, it is difficult for them to authenticate each other directly. If they both share keys with a trusted key server s, then they can use the Yahalom protocol [351]. In this protocol, n_a

and n_b are random values chosen by a and b, respectively. In the future, we will call random values like these *nonces*. The protocol uses the following keys:

- k_{as} – session key shared by a and s,
- k_{bs} – session key shared by b and s, and
- k_{ab} – session key to be used by a and b, chosen by s.

The following messages are exchanged:

- $a \rightarrow b : a \bullet n_a$
- $b \rightarrow s : b \bullet \{a \bullet n_a \bullet n_b\}_{k_{bs}}$
- $s \rightarrow a : \{b \bullet k_{ab} \bullet n_a \bullet n_b\}_{k_{as}} \bullet \{a \bullet k_{ab}\}_{k_{bs}}$
- $a \rightarrow b : \{a \bullet k_{ab}\}_{k_{bs}} \bullet \{n_b\}_{k_{ab}}$

This allows both parties to know the session key for direct communications. All messages are authenticated and no information is exposed in clear-text. The use of nonces guards against replay attacks.

The main issue with establishing a common key, using symmetric key algorithms, is that both parties need to have a common key-server that verifies their identities.

3.8.3 Asymmetric key distribution and public key infrastructure (PKI)

In many ways, key distribution is much easier when using public key cryptography. The public key can be sent in clear-text without compromising security. A major problem remains, how to authenticate that the party delivering the public key is the node we wish to communicate with.

Standards exist for public key authentication. The International Telecommunications Union[13] developed a number of computer networking standards. These included the X.500 directory services standards designed to support the X.400 electronic mail exchange standards. X.400 has been largely dominated by Internet SMTP technologies. The part of the X.500 standard that is most widely used is the X.509 standard, which defines *public key infrastructure* (PKI) approaches used for distributed authentication.

X.509 defines a format for *public key certificates*. A public key certificate is a data structure with two parts: data and signature [287]. The data part includes at a minimum a public key and string identification for the entity being authenticated. It may also contain, among other items:

- A termination date for certificate validity,
- Additional key information (algorithm and use), and
- Additional information about the entity being authenticated.

[13]The ITU is a special agency of the United Nations, which is responsible for making communications standards.

The signature is made by using the private key of a trusted party (called the *certificate authority* (CA)) to encrypt a cryptographic hash of the data part.

The X.509 key exchange protocol for nodes a and b to authenticate each other is [287]:

- $a \rightarrow b : cert_a \bullet t_a \bullet r_a \bullet b \bullet data_1^* \bullet \{k_1^*\}_{pb} \bullet \{t_a \bullet r_a \bullet b \bullet data_1^* \bullet \{k_1^*\}_{pb}\}_{sa}$
- $b \rightarrow a : cert_b \bullet t_b \bullet r_b \bullet a \bullet r_a \bullet data_2^* \bullet \{k_2^*\}_{pa} \bullet \{t_b \bullet r_b \bullet a \bullet r_a \bullet data_2^* \bullet \{k_2^*\}_{pa}\}_{st}$

where $cert_x$ is a certificate from an accepted certificate authority binding x to its public key, px is x's public key, t_x is an expiration time-stamp generated by x, r_x is a nonce generated by x and sx is x's secret key. Both a and b verify each other's certificates and encrypted data items. This provides strong two-way authentication.

We will revisit PKI security and the use of certificates in Chapter 4. That discussion will illustrate how security flaws remain even when strong cryptography is used.

3.9 Message Confidentiality

The assumptions behind cryptography are:

- Keys, with the exception of asymmetric encryption public keys, are kept private,
- The encryption algorithms execute quickly when the key and data are known,
- Encryption algorithms are difficult to invert,
- Cipher-text provides little information about keys and clear-text, and
- Key size is large enough that a brute-force search of the key space will take long enough that the clear-text is not worth the effort.

We presented examples of how these properties can be justified. It was also shown that these assumptions are not always valid. Cryptanalysis is able to extract information from cipher-text and invert the Caesar and Vigenère ciphers (see Sections 3.2 and 3.3). The EFF (see Section 3.4.5) was able to design machines that allowed for a fast brute-force scan of the DES key-space. Moore's law that computation and storage capacity will double every two years presents a danger to designers of cryptographic systems. It is not enough to show that systems can not be broken with current CPUs, since more powerful computers will be available in the near future.

In addition, cryptography depends on sound protocols. In Section 3.8.1 we refer to a notation for expressing cryptographic protocols that is derived from existing protocol analysis techniques. This approach has been used to produce tools for automatic protocol verification [351] that use mathematical expressions of common attacks [35]. Protocol and attack descriptions can then

be input to theorem provers, which either show that the protocols are sound or they find counterexamples. The counterexamples are traces of legitimate protocol interactions that are vulnerable to attack.

Cryptography is a very important tool, but we need to be clear about the guarantees that it does, and does not, provide. In Chapter 4, we will discuss in depth a number of vulnerabilities that have been found in the Secure Sockets Layer / Transport Layer Security (SSL/TLS) system. In Chapter 12, we will discuss the use of side-channel attacks to break the security guarantees provided by cryptography by attacking these systems at another level of abstraction.

3.10 Steganography

Steganography is a security through obscurity concept. It is the science of hiding information. One piece of data is hidden within another. For example, changing the lower order bits of an image or music file is unlikely to noticeably change the original item. White-space characters in documents or computer source code can also contain hidden information. Comparing the original and modified data items allows the hidden data to be retrieved easily.

Herodotus documented the use of steganography for covert communications. Recent concerns about the use of Steganography for secret communications by terrorist organizations is most likely overstated [4, 16]. In addition to being used for covert communications, steganography is applied to digital watermarking. Information hidden in images, sound, or other data can be used to verify the origin of the data. This is used to protect intellectual property rights.

Note that steganographic information is secure because others are unaware of its presence. Once a hiding place is detected, recovery of the information is usually fairly straightforward. Steganalysis refers to discovering and recovering hidden information. Removal of hidden information is often straightforward, once it has been detected. Typically, the attacker finds features that can be embedded with minimal disruption of its host. Researchers are looking at ways of detecting these disturbances, which limits the volume of information that can be hidden without detection [333].

3.11 Obfuscation and Homomorphic Encryption

To obfuscate is to make a concept so confused and opaque that it is difficult to understand [29]. Obfuscation in software design is not always intentional.

Examples showing effective obfuscation of computer code can be found in the results of the annual obfuscated C code-writing contest [17]. We will revisit obfuscation and computing with encrypted functions in the polymorphic virus project (see Chapter 9).

With obfuscation, object or source code is deliberately scrambled in a manner that keeps it functional but hard to reverse engineer. Sometimes obfuscation is derided as being an inferior form of encryption. Obfuscation tries to hide information without providing the formal complexity guarantees of modern encryption techniques. Many types of obfuscation exist. The obfuscation taxonomy in [120] can be summarized as:

- Layout obfuscation - these are simple, straightforward, irreversible, and fairly ineffective,
  - Replace identifier names in source or intermediate code with arbitrary values,
  - Remove formatting information in source or intermediate files,
  - Remove comments explaining code.
- Data obfuscation - makes data structures less obvious,
  - Store and encode data in unusual forms,
    - Create a transformation that maps a variable into a domain requiring multiple inputs,
    - Promote scalar values to objects or insert into a larger data structure,
    - Convert static data into a procedure that outputs the constant values desired,
    - Change variable encodings into an equivalent domain,
    - Change variable lifetime and scope.
  - Change the grouping of data structures,
    - If the range of two variables is restricted they can be joined into one variable with a larger range,
    - Modify inheritance relations between objects, including inserting useless classes in the class structure,
    - Change the dimensionality of arrays.
  - Reorder program instructions and data structures in a random manner,
    - Data declarations can be randomized,
    - The order of instructions can be randomized, so long as this does not change program meaning,
    - Positions of data in arrays can be randomized.
- Control obfuscation- changes program flow adding execution overhead,
  - Aggregation - combine unrelated computations into one module or separate related computations into multiple modules,
    - Replace a procedure call with an in-line copy of the procedure,

    * Take two separate code components and insert them into a common procedure,
    * Take two separate code components and interleave their statements in a common procedure,
    * Make multiple copies of a single subroutine to make calling patterns less obvious,
    * Modify loop structures by combining, splitting, or unrolling the loop.
  - Ordering - randomize the order computations are performed,
    * Reorder statements,
    * Reorder loops,
    * Reorder expressions.
  - Computations - modify algorithms,
    * Insert dead code that is irrelevant into a program,
    * Use native or virtual machine commands that can not be expressed in the higher-level languages,
    * Make loop exit conditions needlessly complex without affecting the number of times a loop is executed,
    * Parallelize the program,
    * Include an interpreter in the program that executes code native to that non-standard interpreter,
- Preventive transformations,
  - Targeted - explore weaknesses in current decompilers,
  - Inherent - Explore problems with known de-obfuscation techniques.

Obfuscation has been suggested as a method of protecting mobile code from attacks by malicious hosts. If code (data) is modified to remain valid only for a short period of time, the time needed for reverse engineering may be longer than the period of time the code (data) is valid. Unfortunately, it is difficult to quantify the time needed to reverse engineer obfuscated code and robustly establish the code validity time periods [224].

Homomorphic encryption resembles obfuscation in many ways. Code and data are executed on remote nodes. Details of the execution process and results are hidden from the user. An early example of homomorphic encryption was found in RSA [287]. The encryption of the product of two numbers is the same as the product of two numbers encrypted with the same key. Following this, Yao looked for ways of modifying Boolean circuits so that they could be executed without revealing the underlying function [437]. Another approach uses coding theory to express numbers in a format that is secured in a way that could be considered cryptography [273].

Finally, Gentry's Ph.D. dissertation at Stanford produced a fully homomorphic encryption scheme [173]. This scheme makes it possible for a machine to compute arbitrary functions of data using an encoding of the function that accepts encrypted inputs, produces encrypted outputs, and is opaque to the machine executing the function. Like the approach in [273], Gentry's approach introduces a limited amount of noise into each computation. At each step, though, the system can decipher the actual value by re-encrypting the result. This is done by performing more than one layer of encryption on the data. The approach in [173] is still computationally intensive, but holds great promise.

3.12 Problems

1. The instructor will provide you with a text encrypted using the Caesar cipher. Use Cryptool [19] cryptanalysis tools to decipher the text. What is the smallest amount of cipher-text you need to decipher the text? Why?

2. The instructor will provide you with a text encrypted using the Vigenère cipher. Use Cryptool [19] cryptanalysis tools to decipher the text. What is the smallest amount of cipher-text you need to decipher the text? Why?

3. Is there a one-time pad that performs a specific Vigenère cipher for a given text? If so, provide an example. If not, explain why not.

4. Use Cryptool [19] to encrypt two documents where the first 3 kilobytes are identical using DES in ECB mode with the same key. Examine the resulting cipher-texts. Is there any information that could be utilized in a cryptanalysis?

5. Under what conditions could a replay attack be performed on an AES encrypted data stream?

6. Use RSA to design a replacement protocol for Shamir's no-key protocol. Justify the security of your approach by showing that it is at least as secure as Shamir's approach.

7. Do RSA encryption manually for a very small message and with a very small value of p.

8. How many operations will it take to break the encryption in the previous problem? Justify your answer.

9. Is obfuscation necessarily less secure than encryption? Justify your answer.