Chapter 2

Modern symmetric encryption

2.1 Binary numbers and message streams

For all the classical ciphers covered in the previous chapter, we dealt with encryption of messages over the standard English alphabet `A`, `B`, `C`, *etc.* Up to some slight modifications for different languages (and possibly the inclusion of punctuation), this was the alphabet of cryptography for thousands of years, well into the 20th century.

In modern times, however, cryptography takes place almost entirely on computers, and the internal 'alphabet' of computer systems is the binary number system.

The familiar decimal number system we are all used to gets its name from the fact that it is based on powers of 10. For example, given a number like 3,725, each digit indicates increasing powers of 10, from right to left. The rightmost digit is the 1's place, the second rightmost digit is the 10's place, the third rightmost digit is the 100's place, and the 4th rightmost digit is the 1000's place. Thus we have

$$3,725 = 3 \cdot 10^3 + 7 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0.$$

There are 10 possible values for each digit (0 through 9), since larger digits would be redundant (since, for example, $10 \cdot 10^0$ is the same as $1 \cdot 10^1$).

The binary number system, on the other hand, is based on powers of 2. For example, the number 10011 in binary represents

$$1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 2 + 1 = 19$$

Rather than there being a 1's place, 10's place, 100's place, and so on, there is a 1's place, a 2's place, a 4's place, *etc.*. The powers of 2 take the place of the powers of 10. Moreover, in binary, we use only *two* possible values for each

digit, 0 and 1, since using any more would be redundant. (For example, even the number 2 can be represented as 10 in binary).

If we want to represent a decimal number, say 35, in binary, we subtract the largest power of 2 less than 35, which is 32, giving 3, and repeat. The largest power of 2 less than 3 is 2, and subtraction gives us 1, which is itself a power of 2. The binary representation has a 1 for each power of 2 subtracted in this process: 100011 in this case, since we have a 1 in the 32's place, the 2's place, and the 1's place.

The fact that any number can be represented with just two distinct digit values is what makes binary so attractive for use in computers, since two digit types can be much more easily represented electronically (power on/power off, magnetized/demagnetized, etc.) than can 10 different digit types.

Ex. 2.1.1. Determine which decimal number each of the following binary numbers represents: 1011, 101, 100110, 111.

Ex. 2.1.2. Determine which decimal number each of the following binary numbers represents: 11011, 1001, 10110, 1111.

Ex. 2.1.3. Determine which binary number each of the following decimal numbers represents: 14, 29, 82, 4.

Ex. 2.1.4. Determine which binary number each of the following decimal numbers represents: 33, 54, 59, 23.

Of course, for binary numbers to be useful to computers, there needs to be ways to do arithmetic operations on them. It turns out that the ways you learned to do arithmetic with decimal numbers carry over basically directly to binary numbers. For example, consider the addition problem

$$\begin{array}{r} 10011 \\ +101011 \\ \hline \end{array}$$

We can carry out the addition just like we would with decimal numbers, keeping in mind that in binary, 1+1=10. Thus, for example, addition of the rightmost digits results in a carry:

$$\begin{array}{r} {\scriptstyle 1} \\ 10011 \\ +101011 \\ \hline 0 \end{array}$$

Thus in the second-rightmost digit, the addition is 1+1+1, which is 11 in binary, resulting in 1 and a carry of 1:

2.1.1. 11,5,38,7

2.1.3. 1110,11101,1010010,100

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 10011 \\ +101011 \\ \hline 10 \end{array}$$

The rest of the addition is shown below:

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 10011 \\ +101011 \\ \hline 111110 \end{array}$$

And we can check that we get the same answer through decimal arithmetic: the binary number 10011 represents the decimal number 19, the binary number 101011 represents the binary number 43, and the binary number 111110 represents the decimal number 62, which is the sum of 19 and 43.

Ex. 2.1.5. Carry out the following binary arithmetic problems, by adapting the standard methods for the equivalent decimal arithmetic problems. (You can check your answers by converting to decimal numbers).

$$\begin{array}{r} 11001 \\ +01011 \\ \hline \end{array} \qquad \begin{array}{r} 101011 \\ -\ \ 10011 \\ \hline \end{array} \qquad \begin{array}{r} 101101 \\ \times\ \ \ 1101 \\ \hline \end{array}$$

Of course, computers need a way to convert between the binary used for internal calculations and the human-readable character set presented to users. This is done with a character encoding scheme, the most famous of which is ASCII, which converts characters to 7 bit binary numbers (see Table 2.1. Newer character-encoding schemes (*e.g.,* UTF-8) allow the encoding of more characters (accented characters, other non-English characters, special symbols, *etc.*.)

For the purpose of doing cryptography with binary message streams, we don't need to be able to convert all the punctuation marks, and don't need to distinguish between upper- and lower-case letters. Thus we will use the conversion given in Table 2.1, which maps characters to 5-bit binary strings. Note that we have given a character assignment for all possible 5-bit binary number. Using the Table, the string `HELLO` would become 0011100100010110101101110, while the string 011011010101110 breaks up into 01101 10101 01110, giving `NVO`.

Ex. 2.1.6. Write `HELLOCOMPUTER` as a binary stream using Table 2.1.

Ex. 2.1.7. Convert the following to a string of letters using Table 2.1: 001110010001011010110111010010100111010000011001000110110011

2.1.1 The binary one-time pad

One can perform one-time pad encryption and decryption on binary message streams. Assume we want to send the message `HELP`. This converts to the binary string 00111001000101101111. If we have generated the random keystream

Dec	Char	Binary
0	A	00000
1	B	00001
2	C	00010
3	D	00011
4	E	00100
5	F	00101
6	G	00110
7	H	00111
8	I	01000
9	J	01001
10	K	01010
11	L	01011
12	M	01100
13	N	01101
14	O	01110
15	P	01111
16	Q	10000
17	R	10001
18	S	10010
19	T	10011
20	U	10100
21	V	10101
22	W	10110
23	X	10111
24	Y	11000
25	Z	11001
26	.	11010
27	!	11011
28	?	11100
29	☺	11101
30	☹	11110
31	_	11111

Table 2.1: Binary/Character conversion table

USASCII code chart

b7 b6 b5 → Bits					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Column → / Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	−	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

Figure 2.1: An old ASCII code chart. The column determines the first three binary digits, while the row determines the final four. .Thus, for example, < corresponds to 0111100

01010111111011010000 (and given a copy to the intended recipient of the message) we can encrypt the message by adding it to the keystream. When we were working with the alphabet, the addition was carried out modulo 26 for each character. For binary streams, we will add each digit *modulo 2*. Note that $0 + 0 \equiv 0 \pmod 2$, $1 + 0 \equiv 1 \pmod 2$, and, finally, $1 + 1 \equiv 0 \pmod 2$. Thus the encryption is given by:

$$\begin{array}{r} 00111001000101101111 \\ \oplus 01010111111011010000 \\ \hline 01101110111110111111 \end{array}$$

We indicate this bitwise addition modulo 2 by the symbol $\oplus$. (The operation is also sometimes referred to as XOR[1].) As discussed in the section on one-time pads, producing random-seeming streams in a deterministic way is a difficult problem. If we have a way of turning a small key into a large 'random-seeming' binary stream, then we can use such a method as a basis for an encryption

[1]XOR stands for eXclusive OR, since it returns 1 if the first digit is 1, OR the second digit is 1, but not if both are 1.

method which XORs the stream and the binary message string—ciphers of this type are called *stream ciphers*. In the next section, we we discuss Linear Feedback Shift Registers as a way of producing apparently random streams from short keys.

2.2 Linear Feedback Shift Registers

Imagine an office of imitators, who obsess about whether or not to wear hats to work to satisfy social pressures. Jan thinks Alice is the coolest one in the office, and wears a hat to work whenever Alice wore one the previous day. Albert things Jan is the coolest person in the office, and so wears a hat whenever Jan wore one on the previous day. Jeff thinks that Albert is totally awesome, and so wears a hat whenever Albert wore on the previous day. Finally, Alice wears a hat whenever either Jeff or Albert wore a hat on the previous day, but not if they both did. (We will not explore the psychology behind such behavior.)

Let's assume that on the first day of the new year, Albert wears a hat and nobody else does. The following table shows how the days will progress in terms of hat-wearing: (1 indicates that someone is wearing a hat on the given day; 0 indicates that they are not.)

day	Alice	Jan	Albert	Jeff
1	0	0	1	0
2	1	0	0	1
3	1	1	0	0
4	0	1	1	0
5	1	0	1	1
6	0	1	0	1
7	1	0	1	0
8	1	1	0	1
9	1	1	1	0
10	1	1	1	1
11	0	1	1	1
12	0	0	1	1

Consider now the perspective of Jeff's lunch buddy Bobby, who never sees Alice, Jan, and Albert, and does not know about the dedicated attempts being made at imitation. Bobby notices that Jeff sometimes wears a hat and sometimes doesn't, but he knows nothing about how the decision is made for a particular day. In fact, to Bobby, the sequence $0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, \ldots$ indicating when Jeff decides to wear a hat (the rightmost column in the above table) might appear to be random. In spite of this appearance, we know that there was actually no randomness involved in the production of this sequence; instead, it was produced by the deterministic imitations of Jeff and his coworkers.

Linear Feedback Shift Registers can be used to build 'random-seeming' sequences in the same way that Jeff's hat-wearing sequence was produced. Con-

sider, for example, the following 'transition rules':

$$\begin{array}{l} b_3 \leftarrow b_4' \\ b_2 \leftarrow b_3' \\ b_1 \leftarrow b_2' \\ b_4 \leftarrow b_1' + b_2' \end{array} \tag{2.1}$$

These rules tell us how to transform one 4-bit string into another. By convention, the indices are written right-to left. Thus, for example, the string 1011 might correspond to the assignment $b_4' = 1$, $b_3' = 0$, $b_2' = 1$, and $b_1' = 1$. Transforming this string according to the rules given above produces the sequence 0101 (corresponding to $b_4 = 0$, $b_3 = 1$, $b_2 = 0$, and $b_1 = 1$).

In fact, these rules represent the rules of imitation practiced by Jeff and his coworkers. In this case, b_4, b_3, b_2, b_1 are the variables indicating whether Alice, Jan, Albert, and Jeff, respectively, wear hats on a given day. The transition rule $b_2 \leftarrow b_3'$, for example, implies that Albert wears a hat whenever Jan did the day before. Notice the last transition rule $b_4 \leftarrow b_1' + b_2'$: adding the bits b_1' and b_2' modulo 2 gives 1 whenever exactly one of them is 1, but not both. Thus $b_4 \leftarrow b_1' + b_2'$ is exactly Alice's hat-wearing rule: she wears a hat whenever Jeff *or* Albert—but not both—wore one the day before.

In general, a Linear Feedback Shift Register (LFSR) can be specified by a single formula like

$$b_4 \leftarrow b_1' + b_2'$$

This formula tells us that the register has 4-bits. It is understood that the three unspecified rules are the rules $b_3 \leftarrow b_4'$, $b_2 \leftarrow b_3'$, and $b_1 \leftarrow b_2'$. Similarly, the LFSR specified by the rule

$$b_5 \leftarrow b_1' + b_2' + b_4'$$

is a 5-bit LFSR whose complete transition rules are

$$\begin{array}{l} b_4 \leftarrow b_5' \\ b_3 \leftarrow b_4' \\ b_2 \leftarrow b_3' \\ b_1 \leftarrow b_2' \\ b_5 \leftarrow b_1' + b_2' + b_4' \end{array}$$

The table of output values can be filled out beginning with a *seed*, which is just an initial condition for the register. For example, if we seed this 5-bit register

with the string 10110, the output will begin with:

	b_5	b_4	b_3	b_2	b_1
1	1	0	1	1	0
2	1	1	0	1	1
3	1	1	1	0	1
4	0	1	1	1	0
5	0	0	1	1	1
6	0	0	0	1	1
7	0	0	0	0	1
8	1	0	0	0	0
9	1	1	0	0	0
10	1	1	1	0	0
11	1	1	1	1	0
⋮	⋮	⋮	⋮	⋮	⋮

The output stream of any LFSR is taken to be the sequence of states of the bit b_1; for the 5-bit sequence whose table of states is shown above, this sequence would be 01101110000.... LFSR's are very commonly used as part of schemes to generate 'pseudo-random' streams of bits because they are very easy to implement in hardware very easily. In spite of the fact that they produce streams that seem very random, the only operations involved in producing the streams are simple bitwise addition, which can performed very efficiently even on very simple computer chips.

The output of an LFSR can be used for the keystream of a stream cipher. For example, lets encrypt the message `HELP` with the output from the LFSR specified by $b_4 \leftarrow b_1' + b_2' + b_4'$ and seeded with the string 0110. The first step is to use Table 2.1 to translate the message `HELP` into the binary message stream 00111001000101101111. Next, we compute the output stream of the LFSR with the given stream. Here is the table of output states:

	b_4	b_3	b_2	b_1
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	0
5	1	0	1	1
6	1	1	0	1
7	0	1	1	0
8	1	0	1	1
9	1	1	0	1
10	0	1	1	0
11	1	0	1	1
⋮	⋮	⋮	⋮	⋮

The output sequence (the b_1 column) appears to be repeating: 011011011 . . . . And indeed, we can see that the 7th state of the register is the same as the first state; thus, the 8th state will be the same as the second state, the 9th the same as the third state, and so on. Thus we do not need to continue filling out the table to produce a long keystream; we can infer that the output will be 011011011011011 . . . .

To encrypt the binary message stream, we add it digit by digit to the keystream modulo 2:

$$\begin{array}{r} 00111001000101101111 \\ \oplus 01101101101101101101 \\ \hline 01010100101000000010 \end{array}$$

We can convert this back to characters using Table 2.1 to get the ciphertext `KSQC`. The recipient, knowing the LFSR register and seed used, would convert this ciphertext back into binary, find the keystream using the LFSR, and subtract the keystream from the binary ciphertext modulo 2 to get the binary message stream (which is then converted back to characters using Table 2.1. Note that, in fact, *subtracting modulo 2 is the same as adding!* For example, $1-1 \equiv 0 \pmod 2$, but also $1+1 \equiv 0 \pmod 2$. This means that the decryption operation is actually just adding the keystream to the binary ciphertext, and so is the same as the encryption operation. The decryption for our message would look like this:

	K	S	Q	C
	01010	10010	10000	00010
⊕	01101	10110	11011	01101
	00111	00100	01011	01111
	H	E	L	P

Ex. 2.2.1. Encrypt the message `HI` using a stream cipher using the LFSR given by $b_4 \leftarrow b_3' + b_1'$ and seeded with 1011.

Ex. 2.2.2. Encrypt the message `BLUE` using a stream cipher using the LFSR given by $b_5 \leftarrow b_4' + b_1'$ and seeded with 11011.

Ex. 2.2.3. Decrypt the message `?SY`, which was encrypted using a stream cipher with a keystream produced by the LFSR with formula $b4 \leftarrow b_3' + b_1'$ and seed 0110.

Recall that in the example above where `HELP` was encrypted to `KSQC`, the keystream produced by the LFSR being used eventually repeated. In fact this will *always* happen eventually, no matter what LFSR is used and no matter the seed. This is simply because there is a finite number of possible states for the LFSR. For example, consider the 3-bit LFSR register given by $b_3 \leftarrow b_2' + b_1'$, seeded with the string 101. The table of states begins with:

	b_3	b_2	b_1
1	1	0	1
⋮	⋮	⋮	⋮

There are only finitely many possible states (in fact, only 8: 111, 110, 101, 100, 011, 010, 001, 000), thus, eventually, some state will have to be repeated as we fill out the table[2] Once that happens, the table will repeat that section over and over again, since the next row of the table is completely determined by the previous row. The same thing will happen for any LFSR, although for more bits there are more possibilities.

Ex. 2.2.4. What is the longest possible period of repetition of a 4-bit LFSR register? Of a 5-bit LFSR register? Of an n-bit LFSR register? (Be sure to read footnote 2.)

It should be noted that, while LFSR streams 'seem' random, they are not really random at all. For one thing, as already noticed, they eventually will just repeat over and over again. For another, it turns out that given enough of an LFSR stream, it is possible to figure out the rest of it. This is crucial for the application of LFSR's to cryptography, as we will see shortly.

Known-plaintext attack on LFSR stream ciphers

Suppose we have intercepted the message `WIMSUJ`, encrypted with an LFSR stream cipher. As is often the case with LFSRs in cryptographic settings, we assume the formula for the LFSR is public knowledge[3] (in this case, $b_4 \leftarrow b_4' + b_1'$), and that only the seed is kept secret as the key. To mount a known-plaintext attack on the stream cipher, we need to have some known plaintext. In this case, we know that the intercepted message is a name, and, in particular, begins with the letters `MR`.

This allows us to discover the beginning of the keystream used for encryption as follows: The message `WIMSUJ` corresponds to the binary stream 1011001000 . . . , while the letters `MR` correspond to 0110010001 in binary. By subtracting the second string from the first modulo 2, we can get the beginning of the keystream used for encryption. Since subtraction and addition are the same modulo 2, we are just adding the two strings modulo 2:

$$\begin{array}{r} 1011001000 \\ \oplus 0110010001 \\ \hline 1101011001 \end{array}$$

2.2.1. ?R

[2]In fact, the all zero state 000 can never arise, unless the register eventually outputs all 0s and so has a period of repetition of 1. This means that, for a 3 bit register, the longest possible period of repetition is 7.

[3]this is often the case because the LFSR formula is hardcoded (built-into) the hardware used for encryption, and cannot be changed as part of the key.

The result must be the beginning of the keystream used for encryption, thus we know that this string forms the beginning of the right-hand column of the table of LFSR states:

	b_4	b_3	b_2	b_1
1				1
2				1
3				0
4				1
5				0
6				1
7				1
8				0
9				0
10				1
11				
12				

To decrypt more of the ciphertext, we need to be able to fill in more of the b_1 column; but to do this, we need to know the other missing bits in the table. But we can figure out many of these bits from what we know about the right and column! For example, the fact that the $b_1 = 1$ in the second state means that $b_2 = 1$ in the first state (because of the transition rule $b_1 \leftarrow b_2$), as we have indicated in table below on the left:

	b_4	b_3	b_2	b_1
1			1	1
2				1
3				0
4				1
5				0
6				1
7				1
8				0
9				0
10				1

	b_4	b_3	b_2	b_1
1		0	1	1
2			0	1
3				0
4				1
5				0
6				1
7				1
8				0
9				0
10				1

Similarly, the fact that $b_1 = 0$ in the 3rd state means that $b_2 = 0$ in the second state, and, moreover, that $b_3 = 0$ in the first state, as we have indicated in the table above on the right. Continuing in this way, we can fill in much more of the table:

	b_4	b_3	b_2	b_1
1	1	0	1	1
2	0	1	0	1
3	1	0	1	0
4	1	1	0	1
5	0	1	1	0
6	0	0	1	1
7	1	0	0	1
8		1	0	0
9			1	0
10				1

At this point, we can start working down the table from the transition rules (in this case, $b_1 \leftarrow b_2'$, $b_2 \leftarrow b_3'$, $b_3 \leftarrow b_4'$, and $b_4 \leftarrow b_4 \oplus b_1$). For example, the 8th state of the bit b_4 is given by $1 \oplus 1$, which is 0. This, in turn, is then the 9th state of the bit b_3, the 10th state of the bit b_2, and the 11th state of the bit b_1:

	b_4	b_3	b_2	b_1
1	1	0	1	1
2	0	1	0	1
3	1	0	1	0
4	1	1	0	1
5	0	1	1	0
6	0	0	1	1
7	1	0	0	1
8	0	1	0	0
9		0	1	0
10			0	1
11				0

And now we can fill in the 9th state of the bit b_4, and so on. Continuing in this manner we produce:

	b_4	b_3	b_2	b_1
1	1	0	1	1
2	0	1	0	1
3	1	0	1	0
4	1	1	0	1
5	0	1	1	0
6	0	0	1	1
7	1	0	0	1
8	0	1	0	0
9	0	0	1	0
10	0	0	0	1
11	1	0	0	0
12	1	1	0	0
13	1	1	1	0
14	1	1	1	1
15	0	1	1	1
16	1	0	1	1
17		1	0	1
18			1	0
19				1

Since the entire message we are trying to decrypt consists of 6 characters, and so corresponds to a binary string of length 30, we need the first 30 bits of output to decrypt the stream. We can stop filling in the table at this point since we have encountered a repeated state: The 16th state in the above table is the same as the first states. This means that the first 15 digits of the output stream from this LFSR repeat over and over again, giving an output stream of

$$110101100100011\ 110101100100011\ 110101100100011\ \ldots$$

Subtracting (adding) this to the binary ciphertext string gives:

	W	I	M	S	U	J
	10110	01000	01100	10010	10100	01001
⊕	11010	11001	00011	11010	11001	00011
	01100	10001	01111	01000	01101	01010
	M	R	P	I	N	K

And we have found the plaintext, MRPINK. As we have seen, with even just a tiny amount of known-plaintext, a stream cipher based on a single LFSR can be completely broken by a known plaintext attack—the problem, roughly speaking, was that it is to easy to work backwards from a given part of an output stream and determine previously unknown states of the registers. Nevertheless, the simplicity of LFSR's make them very attractive for many forms of cryptography, especially in cases where the encryption or decryption is to be carried out by a dedicated hardware device, and thus there are sophisticated ways of designing stream ciphers which make use of LFSR's, but try to overcome their weakness towards known-plaintext attacks. We cover one such type of cipher in the next section.

Ex. 2.2.5. The message .ZYHM.! was encrypted with a LFSR stream cipher which uses the formula $b_4 \leftarrow b_4' + b_1'$. Knowing that the message begins with the letters 'Mr', determine the key and decrypt the message.

Ex. 2.2.6. The message ☹YWTCQ! was encrypted with a LFSR stream cipher which uses the formula $b_4 \leftarrow b_4' + b_2' + b_1'$. Knowing that the message begins with the letters 'Mr', determine the key and decrypt the message.

2.3 LFSRsum

We've already seen that encryption schemes based on a single LFSR are quite insecure against known-plaintext attacks, and can be easily broken once we know a little of the plaintext.

By combining multiple LFSR's, however, we can attempt to create a system which is more resilient to plaintext attacks. As a first example of this, we will consider a scheme, which we call LFSRsum, which forms an encryption keystream as the modulo-2 sum of two different LFSR output streams.

The LFSRsum system uses a 3-bit LFSR and a 5-bit LFSR in tandem to create a single 'pseudorandom' stream. The 3-bit output stream is defined by the formula $b_3 = b_2' + b_1'$, while the 5-bit LFSR is defined by $b_5 = b_4' + b_2' + b_1'$. We will refer to these registers by LFSR-3 and LFSR-5, respectively.

For example, if we seed LFSR-3 and LFSR-5 with the strings 011 and 10101,

respectively, their output is:

b_3	b_2	b_1
0	1	1
0	0	1
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1

b_5	b_4	b_3	b_2	b_1
1	0	1	0	1
1	1	0	1	0
0	1	1	0	1
0	0	1	1	0
1	0	0	1	1
0	1	0	0	1
0	0	1	0	0

The LFSRsum system works by simply adding these two streams together modulo 2 to form a keystream, which is then added (modulo 2) to the binary message stream to encrypt it. For example, to encrypt the message Y using the seeds given above, the encryption stream used would be

$$\begin{array}{r} 11001 \\ \oplus 10101 \\ \hline 01100 \end{array}$$

This is then added to the plaintext to encrypt:

$$\begin{array}{r} \texttt{Y} \\ 11000 \\ \oplus 01100 \\ \hline 10100 \\ \texttt{U} \end{array}$$

The key for the LFSRsum cipher consists of 6 binary digits. The first 2 digits give the first two digits of the seed for the LFSR-3 register, and the last 4 digits of the key give the first 4 digits of the LFSR-4 register seed. The last digit of the seed for each register is always 1: this means that no matter what key we choose, the registers will never get set to all-0's. Notice that for the example above, the key was 011010.

Ex. 2.3.1. Encrypt the message HELLO using the LFSRsum cipher, using 001100 as a key.

Just like with single LFSR streams, the decryption operation is *exactly the same* as the encryption operation, since adding is the same as subtracting.

Ex. 2.3.2. Decrypt the message DTQRZ_Y which was encrypted with the LFSRsum cipher using the key 001110.

At first glance, this new scheme might seem secure against a known plaintext attack. Assume, for example, that using a known plaintext we have discovered that the beginning of an encryption stream produced by the LFSRsum system using some unknown key is 011110011. Can we figure out the key used, and calculate more of the decryption stream? It seems like we can't since, for example, the fact that the first digit of this keytream is a 1 could reflect the fact that

the first digit of the LFSR-3 output stream is a 1, or it could reflect the fact that the first digit of the LFSR-5 output stream is a 1, and it does not seem that there is any way we can figure out which of these possibilities is correct. In fact, however, we can completely figure out how each register contributed to the encryption stream by solving a system of equations.

Since the 6-bit key used to produce the encryption stream 011110011 is unknown to us, let's represent it with the 6 variables $k_1k_2k_3k_4k_5k_6$, each either a 0 or a 1. The LFSR registers used for the LFSRsum cipher are initialized with these bits:

b_3	b_2	b_1
k_1	k_2	1
⋮	⋮	⋮

b_5	b_4	b_3	b_2	b_1
k_3	k_4	k_5	k_6	1
⋮	⋮	⋮	⋮	⋮

In spite of the fact that we don't know the values of these seed bits, we can nevertheless use the transition rules of the LFSR-3 and LFSR-5 registers to fill in subsequent rows of the tables. Note that modulo 2, we have $1+1 \equiv 0 \pmod 2$, but also, for example, $k_2+k_2 \equiv 2\cdot k_2 \equiv 0 \pmod 2$. Thus, for example, in the LFSR-3 table below, the 5th state of the bit b_3 is simplified to $1+k_1$ from $(1+k_2)+(k_2+k_1)$:

b_3	b_2	b_1
k_1	k_2	1
$1+k_2$	k_1	k_2
k_2+k_1	$1+k_2$	k_1
k_1+1+k_2	k_2+k_1	$1+k_2$
$1+k_1$	k_1+1+k_2	k_1+k_2
1	$1+k_1$	k_1+1+k_2
k_2	1	$1+k_1$
k_1	k_2	1
⋮	k_1	k_2
⋮	⋮	k_1

b_5	b_4	b_3	b_2	b_1
k_3	k_4	k_5	k_6	1
$1+k_6+k_4$	k_3	k_4	k_5	k_6
$k_6+k_5+k_3$	$1+k_6+k_4$	k_3	k_4	k_5
$1+k_5+k_6$	$k_6+k_5+k_3$	$1+k_6+k_4$	k_3	k_4
$k_4+k_5+k_6$	$1+k_5+k_6$	$k_6+k_5+k_3$	$1+k_6+k_4$	k_3
$k_3+k_4+k_5$	$k_4+k_5+k_6$	$1+k_5+k_6$	$k_6+k_5+k_3$	$1+k_6+k_4$
⋮	$k_3+k_4+k_5$	$k_4+k_5+k_6$	$1+k_5+k_6$	$k_6+k_5+k_3$
⋮	⋮	$k_3+k_4+k_5$	$k_4+k_5+k_6$	$1+k_5+k_6$
⋮	⋮	⋮	$k_3+k_4+k_5$	$k_4+k_5+k_6$
⋮	⋮	⋮	⋮	$k_3+k_4+k_5$

Thus we know that, in terms of the key bits $k_1, k_2, k_3, k_4, k_5, k_6$, the keytream generated by the LFSRsum cipher should be the (modulo 2) sum of the two output streams found above; namely, $(1+1)$, (k_2+k_6), (k_1+k_5), $(1+k_2+k_4)$, $(k_1+k_2+k_3)$, $(k_1+1+k_2+1+k_6+k_4)$, $(1+k_1+k_6+k_5+k_4), \ldots$. On the other hand, we *know* (from a known-plaintext attack) that the actual encryption stream is 0111100. This allows us to set up a system of congruences. For example, we know from the tables that the first digit of the encryption stream should be congruent to $1+1 \pmod 2$, and from the known-plaintext attack that it should be 0. This isn't really helpful since it is true regardless of the assignment of variables. For the second digit of the encryption stream, on the other hand, the tables give that the value should be k_2+k_6 (modulo 2), while the known plaintext attack tells us the digit should be 1. This allows us to setup the congruence

$$k_2+k_6 \equiv 1 \pmod 2.$$

In fact, we can setup such a congruence for each digit of the encryption stream that we know: for the third digit, we get $k_1+k_5 \equiv 1 \pmod 2$; for the fourth, we get $1+k_2+k_4 \equiv 1 \pmod 2$. Continuing in this fashion, we will obtain the system

$$\begin{cases} k_2+k_6 \equiv 1 \pmod 2 \\ k_1+k_5 \equiv 1 \pmod 2 \\ 1+k_2+k_4 \equiv 1 \pmod 2 \\ k_1+k_2+k_3 \equiv 1 \pmod 2 \\ k_1+k_2+k_6+k_4 \equiv 0 \pmod 2 \\ 1+k_1+k_6+k_5+k_3 \equiv 0 \pmod 2 \end{cases} \tag{2.2}$$

At this point, we have six unknowns and six congruences, so we can try solving the system. This looks daunting at first, but it is really not that bad! Let's begin by reordering the variables in each congruence:

$$\begin{cases} \quad k_2 \qquad\qquad\qquad + k_6 \equiv 1 \pmod 2 \\ k_1 \qquad\qquad\qquad + k_5 \qquad \equiv 1 \pmod 2 \\ \quad + k_2 \qquad + k_4 \qquad\qquad \equiv 0 \pmod 2 \\ k_1 + k_2 + k_3 \qquad\qquad\qquad \equiv 1 \pmod 2 \\ k_1 + k_2 \qquad + k_4 \qquad\quad k_6 \equiv 0 \pmod 2 \\ k_1 \qquad + k_3 \qquad + k_5 + k_6 \equiv 1 \pmod 2 \end{cases} \tag{2.3}$$

(Note that in the third and sixth congruence, 1 has been added to both sides so that only variables remain and the lefthand sides.) The system can be solved very efficiently by adding congruences. For example, adding the 1st, 2nd, and 6th congruences modulo 2 gives:

$$\begin{array}{r} k_2 \qquad\qquad + k_6 \equiv 1 \pmod 2 \\ k_2 \qquad\qquad + k_6 \equiv 1 \pmod 2 \\ + \quad k_1 \qquad + k_3 \quad + k_5 + k_6 \equiv 1 \pmod 2 \\ \hline k_5 \qquad \equiv 1 \pmod 2 \end{array} \tag{2.4}$$

And we have found that

$$k_5 \equiv 1 \pmod 2. \tag{2.5}$$

Adding this congruence to the second congruence from line (2.3) gives

$$k_1 \equiv 0 \pmod 2. \tag{2.6}$$

Adding the 1st, 2nd, 3rd, and 5th congruences from (2.3) gives

$$\begin{array}{rl} k_2 \qquad\qquad + k_6 \equiv 1 & \pmod 2 \\ k_1 \qquad\qquad + k_5 \qquad \equiv 1 & \pmod 2 \\ k_2 + k_4 \qquad\qquad \equiv 0 & \pmod 2 \\ + \quad k_1 + k_2 + k_4 \qquad k_6 \equiv 0 & \pmod 2 \\ \hline k_2 \qquad + k_5 \qquad \equiv 0 & \pmod 2 \end{array} \tag{2.7}$$

and adding the result to the congruence in (2.5) gives that

$$k_2 \equiv 1 \pmod 2. \tag{2.8}$$

Adding (2.8) to the first congruence from (2.3) gives

$$k_6 \equiv 0 \pmod 2, \tag{2.9}$$

and adding (2.8) to the third congruence from (2.3) gives

$$k_4 \equiv 1 \pmod 2, \tag{2.10}$$

Finally, adding (2.6) and (2.8) to the fourth congruence from (2.3) gives

$$k_3 \equiv 0 \pmod 2,. \tag{2.11}$$

We have thus been able to complete figure out the key, namely, 010110. With this information, we could figure out the rest of the LFSR encryption stream and decrypt the message.

Ex. 2.3.3. The message `FCJUWRMX` was encrypted with the LFSRsum cipher.

(a) The first two letters of the plaintext are 'Mr'. Use this information to determine the first 10 digits of the binary stream used for encryption.

(b) Using the binary digits found in part (a), mount a known plaintext attack on the LFSRsum cipher to determine the key used for encryption, and decrypt the rest of the message. *Hint:* The equations you need to solve to break the cipher are exactly the same as those in in line (2.3), except for the 0s and 1s on the righthand side of each. This means also that the same combinations of congruences can be summed to solve for the separate variables!

Congruences (and equations) in which variables are never multiplied by each other (only added or subtracted) are called *linear*. (For example, $y - 3x = 0$ is a linear equation, since x is not multiplied by y). Note that the congruences we needed to solve to break LFSRsum system were all linear. It turns out that this can always be done efficiently. Even though it seemed like there was a bit of trial and error involved in solving the congruences in the above example (for example, in deciding which congruences to add together), the process can be carried out by the method of *Gaussian elimination*, which we will not cover here, but which is a systematic way of carrying out the appropriate additions/subtractions of congruences to reach a solution. It turns out that linear systems can be solved quickly even when there are lots and lots of congruences; a computer could solve a linear system containing millions of congruences (and variables) in a matter a seconds. This means that even if LFSRsum was modified to use gigantic LFSRs, (and thus take a gigantic key), it would still not be secure against known-plaintext attacks. For nonlinear systems of congruences, however, there is no good general method of solving systems of equations, suggesting that to make a secure cipher based on LFSRs, we need to introduce some 'nonlinearity' to prevent known-plaintext attacks like the one shown above.

2.4 BabyCSS

BabyCSS is in many ways similar to LFSRsum. It uses the same LFSR-3 and LFSR-5 registers (given by the formulas $b_3 \leftarrow b_2' + b_1'$ and $b_5 \leftarrow b_4' + b_2' + b_1'$, respectively), and again uses a 6 bit key, whose first two bits give the first two bits of the LFSR-3 seed, and whose final four bits give the first four bits of the LFSR-5 seed (ss for the LFSRsum system, the last seed-bit of each register is 1).

Instead of adding the two LFSR output streams bitwise modulo-2, BabyCSS combines the two streams in blocks of 5 bits through addition with carrying. To see how this works, let's see how to encrypt the message `HI` with the BabyCSS cipher using the key 011010 (the same used in the first example in the LFSRsum section.) Since the binary message stream for `HI` has 10 digits, we need to compute 10 digits of the BabyCSS encryption stream. The output tables of the LFSR-3 and LFSR-5 registers are:

b_3	b_2	b_1
0	1	1
0	0	1
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1
0	1	1
0	0	1
1	0	0

b_5	b_4	b_3	b_2	b_1
1	0	1	0	1
1	1	0	1	0
0	1	1	0	1
0	0	1	1	0
1	0	0	1	1
0	1	0	0	1
0	0	1	0	0
0	0	0	1	0
1	0	0	0	1
1	0	0	0	0

We add the streams in blocks of 5 with carrying (as opposed to bitwise modulo 2 as was done for LFSRsum). For example, adding the first 5 bits of each output stream gives:

$$\begin{array}{rccccc} \textcircled{1} & & & & 1 & \\ & 1 & 1 & 0 & 0 & 1 \\ + & 1 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 1 & 0 \end{array}$$

The resulting bits 01110 will form the first 5 output bits of the encryption stream. Notice that there was an 'extra' carry digit (circled above). Instead of becoming a sixth digit in the sum, we carry this digit over to the next addition problem. Thus, adding the next 5 digits gives:

$$\begin{array}{rccccc} 1 & 1 & 1 & 1 & & 1 \\ & 0 & 1 & 1 & 1 & 0 \\ + & 1 & 0 & 0 & 1 & 0 \\ \hline & 0 & 0 & 0 & 0 & 1 \end{array}$$

There is again an extra carry digit. Since we do not need to compute any more digits of the encryption stream, the extra carry digit from this addition is simply discarded.

`HI` corresponds to the binary message stream 0011101000. The encryption stream found through the additions above is 0111000001. Encryption works by adding these streams modulo 2:

$$\begin{array}{r|l} \texttt{H} & \texttt{I} \\ 00111 & 01000 \\ \oplus 01110 & 00001 \\ \hline 01001 & 01001 \\ \texttt{J} & \texttt{J} \end{array}$$

Note that in spite of the fact that BabyCSS uses addition with carrying in blocks of 5 to produce its encryption stream, the resulting stream is still added to the message modulo 2 for encryption, and thus, like other stream ciphers in the same mold, encryption and decryption are still exactly the same operation. Adding the encryption stream to the encrypted message recovers the plaintext, since addition and subtraction are the same modulo 2:

$$\begin{array}{r|l} \texttt{J} & \texttt{J} \\ 01001 & 01001 \\ \oplus 01110 & 00001 \\ \hline 00111 & 01000 \\ \texttt{H} & \texttt{I} \end{array}$$

Ex. 2.4.1. Encrypt the message `BLUE` with the BabyCSS cipher using the key 110011.

The value of carrying

Recall that the LFSRsum system can be completely broken by a known plaintext attack by solving a system of linear equations. In this section, we will see why BabyCSS does not suffer the same weakness. As for the LFSRsum cipher, the

output tables of the LFSR-3 and LFSR-5 registers under a key $k_1k_2k_3k_4k_5k_6$ are:

b_3	b_2	b_1
k_1	k_2	1
$1+k_2$	k_1	k_2
k_2+k_1	$1+k_2$	k_1
k_1+1+k_2	k_2+k_1	$1+k_2$
$1+k_1$	k_1+1+k_2	k_1+k_2
1	$1+k_1$	k_1+1+k_2
$\vdots$	$\vdots$	$\vdots$

b_5	b_4	b_3	b_2	b_1
k_3	k_4	k_5	k_6	1
$1+k_6+k_4$	k_3	k_4	k_5	k_6
$k_6+k_5+k_3$	$1+k_6+k_4$	k_3	k_4	k_5
$1+k_5+k_6$	$k_6+k_5+k_3$	$1+k_6+k_4$	k_3	k_4
$k_4+k_5+k_6$	$1+k_5+k_6$	$k_6+k_5+k_3$	$1+k_6+k_4$	k_3
$k_3+k_4+k_5$	$k_4+k_5+k_6$	$1+k_5+k_6$	$k_6+k_5+k_3$	$1+k_6+k_4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Suppose we know from a known-plaintext attack that the first 5 bits of a BabyCSS encryption stream is 01000. Let's form a system of what congruences can we form from this information? We know that the 5th bit of the stream is the sum of the 5th output bits from each of the registers: this gives the congruence

$$k_1 + k_2 + k_3 \equiv 0 \pmod 2.$$

So far so good. But now it gets tricky: the fourth bit of the output stream is the sum of the fourth output bit from each of the registers, *plus* any carry bit from the 5th digit. There is a carry digit resulting from two single binary digits exactly whenever they are both one. Similarly, the *product* of two single binary digits is 1 exactly whenever they are both one. Thus, we can represent the value of the carry digit (1 or 0) from the 5th bits as the product of those bits: $(k_1 + k_2) \cdot k_3$. The fourth bit of the encryption stream is thus given as the sum of this carry bit with the fourth output bits from each of the registers, giving the congruence

$$(k_1 + k_2) \cdot k_3 + 1 + k_1 + k_4 \equiv 0.$$

Note that this congruence is *not linear*, since it involves multiplication of variables. Subsequent congruences will be even more complicated. The fact that the congruences can not be represented with addition alone means that the congruences can not be efficiently solved through Gaussian elimination, so that BabyCSS does not suffer the same weakness as LFSRsum.

Breaking BabyCSS

The LFSRsum cipher could be completely broken by a known-plaintext attack, in the sense that an attacker that knows some corresponding plaintext and ciphertext (and so can figure out some of the encryption stream) can quickly deduce the key used quickly and without any guesswork, allowing her to decrypt the rest of the message.

The BabyCSS cipher is not *as* susceptible to a known-plaintext attack: we have seen that the congruences that would need to be solved to directly deduce the key used to produce a given encryption stream are nonlinear, indicating that there is not likely an efficient way to solve them.

Nevertheless, a known-plaintext attack is still quite effective against the BabyCSS cipher. Note that the BabyCSS cipher has $2^6 = 64$ possible keys, thus a brute-force attack on the cipher would require at most 64 guesses to break the cipher. It turns out that a known-plaintext attack can break the cipher using at most 4 guesses!

To see how this works, suppose we have intercepted the message

`FPSM☹ U`

which was encrypted with the BabyBlock cipher using an unknown key. We know the message contains the name of a double agent, and believe it begins with the letters `MR`. Since BabyCSS encrypts binary message streams by adding them to the BabyCSS encryption stream modulo 2, we can find the beginning of the BabyCSS encryption stream by computing

$$\begin{array}{rl} 0010101111 & (\texttt{FP}) \\ \oplus 0110010001 & (\texttt{MR}) \\ \hline 0100111110 & \end{array}$$

We have already seen that knowing part of the encryption stream is not enough to work backwards to find the seeds of the registers by solving a system of linear congruences.

Imagine, however, that someone tells us that the LFSR-3 register used for this BabyCSS encryption was seeded with 001 (in other words, they have told us that the first two bits of the BabyCSS key are 00). Never mind for the moment how they might know this.

In that case, we know can compute the output table of the LFSR-3 register:

b_3	b_2	b_1
0	0	1
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1
0	1	1
0	0	1
1	0	0
0	1	0

So far, we know that the BabyCSS encryption stream begins with 0100111110, while the LFSR-3 output begins with 1001011100. From these two pieces of information, can we figure out the LFSR-5 output stream? It turns out we can! Since the BabyCSS encryption stream is formed by adding the LFSR-3 and LFSR-5 output streams (in blocks of 5 with carrying), the LFSR-5 output stream can be found by subtracting the LFSR-3 output stream from the BabyCSS stream (in blocks of 5 with borrowing). We arrange the two streams into blocks of 5:

$$\begin{array}{cc} 01001 & 11110 \\ \hline 10010 & 11100 \\ \hline \end{array}$$

and subtract the blocks, starting with the leftmost one:

$$\begin{array}{r} 0\ 1 \\ {}^{1}0\ \cancel{1}\ \cancel{0}{}^{1}0\ 1 \\ -\ 1\ 0\ 0\ 1\ 0 \\ \hline 1\ 0\ 1\ 1\ 1 \end{array}$$

Note that there was an 'extra borrow' from the leftmost digit of the problem; this is borrowed from the rightmost digit of the next block of 5:

$$\begin{array}{r} 0\ 0\ 0\ 1 \\ {}^{1}1\ \cancel{1}\ \cancel{1}\ \cancel{1}\ \cancel{0} \\ -\ 1\ 0\ 0\ 1\ 0 \\ \hline 0\ 0\ 0\ 0\ 1 \end{array}$$

Thus we have found that the LFSR-5 output stream begins with 1011100001.

We can thus fill in the righthand column of the LFSR-5 output table:

b_5	b_4	b_3	b_2	b_1
1	1	1	0	1
0	1	1	1	0
0	0	1	1	1
0	0	0	1	1
0	0	0	0	1
1	0	0	0	0
	1	0	0	0
		1	0	0
			1	0
				1

And thus we have found that the seed for the LFSR-5 register was 11101! Thus the BabyCSS key used was 001110.

Ex. 2.4.2. Decrypt the rest of the intercepted message.

As seen above, we can break the BabyCSS cipher with a known plaintext attack *if* we know the seed of the LFSR-3 cipher. What if we don't know the seed? Guess at it! There are only 4 possible seeds for the LFSR-3 cipher (001, 011, 101, 111). By trying each one, we can see which one gives a sensible decryption. (In fact, improper guesses should usually be able to ruled out based on whether or not the resulting LFSR-5 output table is consistent with the transition rules.)

Ex. 2.4.3. You are trying to break a message which was encrypted with the BabyCSS cipher. The ciphertext is `DXMP`, and you know that the plaintext begins with `MR`.

(a) What does the encryption stream begin with (first 7 digits)?

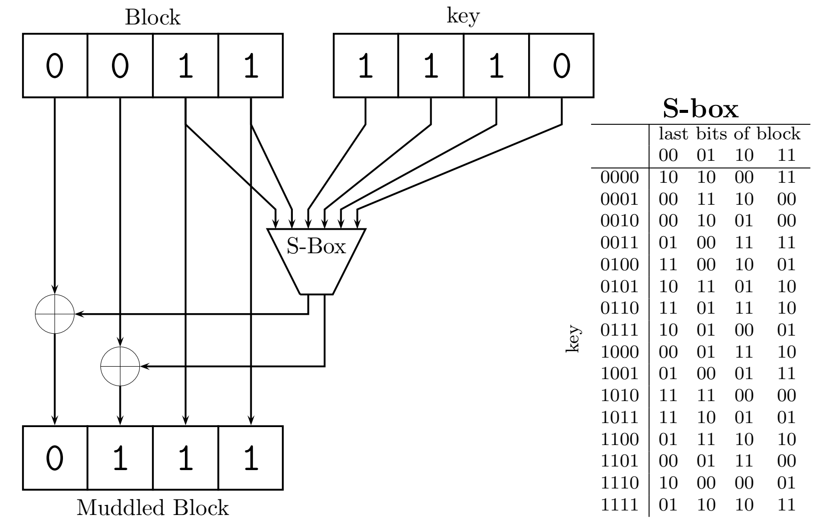
(b) If you guess that the LFSR-3 register is seeded with 111, what does this mean the LFSR-5 output stream was?

(c) What was the LFSR-5 output stream seeded with?

(d) What was the key?

(e) Decrypt the rest of the message.

Ex. 2.4.4. In the known-plaintext example carried out in this section, we had 10 bits of known plaintext (the 10 bits corresponding to the known characters `MR`). Would the same attack work if we just knew 8 of the plaintext bits? What about 4 of the plaintext bits? What is the minimum number of known bits necessary for the known-plaintext attack described in this section to work?



S-box

key	last bits of block 00	01	10	11
0000	10	10	00	11
0001	00	11	10	00
0010	00	10	01	00
0011	01	00	11	11
0100	11	00	10	01
0101	10	11	01	10
0110	11	01	11	10
0111	10	01	00	01
1000	00	01	11	10
1001	01	00	01	11
1010	11	11	00	00
1011	11	10	01	01
1100	01	11	10	10
1101	00	01	11	00
1110	10	00	00	01
1111	01	10	10	11

Figure 2.2: The muddling of block `0011` when the key is `1110`. the $\oplus$'s mean addition modulo 2. The S-box is a table-based substitution.

2.5 BabyBlock

A *block cipher* is one in which the message is broken up into blocks before encryption, and identical blocks all get sent to the same thing. For example, the Hill cipher is a block cipher where the blocks have size 2.

Modern block ciphers are used in situations where the highest levels of security are required. The only publicly known cipher which is approved for encryption of "Top Secret" information at the government level is a block cipher. Block ciphers are also used for secure Internet transactions. We will study a simplified example of a block-cipher to see how this type of cipher achieves security.

The BabyBlock cipher operates on blocks of size 4, and has a key consisting of 4 bits. Consider the situation where we are encrypting the message 'Help' with the key 1110. As with stream ciphers, the actual encryption/decryption is done in 0's and 1's, so we begin by converting the message to a binary string using our character table. H corresponds to 00111, E corresponds to 00100, L corresponds to 01011, and P corresponds to 01111. Thus we are encrypting the message:

$$00111001000101101111$$

Since the BabyBlock cipher operates on blocks of size 4, we begin by breaking the message into blocks of this size, adding extra bits to the end of the message

if we need to:

0011 1001 0001 0110 1111

The BabyBlock cipher encrypts the message one block at a time, so lets begin with the first block Our key is 1110.

We use the key to *muddle* the blocks. This process is shown in Figure 2.2. The last two bits of the block are not changed by the muddling operation, but are used with the key to produce an bit pair using an *S-box.* An S-box is just a substitution table. Given the last two bits of the block and the key, the S-box table gives a pair of bits; these bits are added to the first 2 bits of the block being encrypted to make the last two bits of the muddled block.

Ex. 2.5.1. The second 4-bit block from the plaintext HELP is 1001. Muddle this block using the key 1110

Notice that the last two bits are never changed by the muddling operation. We'll get back to this in a moment. For now, the obvious question is: how do we unmuddle a block? By muddling it! Muddling a muddled block, you'll get back the original block. Why is that? The S-Box will give the same output as before (since the last two bits haven't changed and the key is the same), and adding mod 2 is the same as subtracting mod 2, so when we add the S-Box output to the muddled bits, it will change them to the original bits.

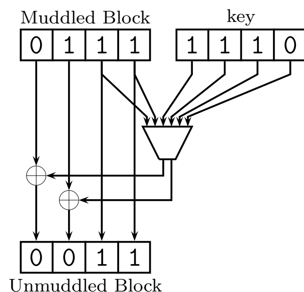


Figure 2.3: Unmuddling is the same as muddling!

If we're going to use the muddling operation we've defined to encrypt things, we're going to have to do something about the fact that the last two bits never change.

BabyBlock deals with this problem by muddling a block repeatedly, but swapping the first and last pairs of digits between each time. For example: we've already seen that the plaintext 0011 gets muddled to 1011. Now we switch the first and last pair of digits in the string to get 1110. And now we can muddle this string. But we don't use the same key! After each muddling operation, we shift the digits of the key to the left. BabyBlock does this 3 times. The operation is shown in Figure 2.4. The ciphertext for the first block of our plaintext turns out to be 1110.

We've already defined the encryption side of BabyBlock: muddle, swap first and last pairs, muddle, swap first and last pairs, and muddle. Notice that each of these operations reverses itself. Muddling reverses muddling, and swapping pairs reverses swapping pairs. Because of this, encryption and decryption for BabyBlock are exactly the same, except you have to use the keys in reverse order: first use the key shifted to the left by 2, then use the key shifted to the left by 1, then use the original key.

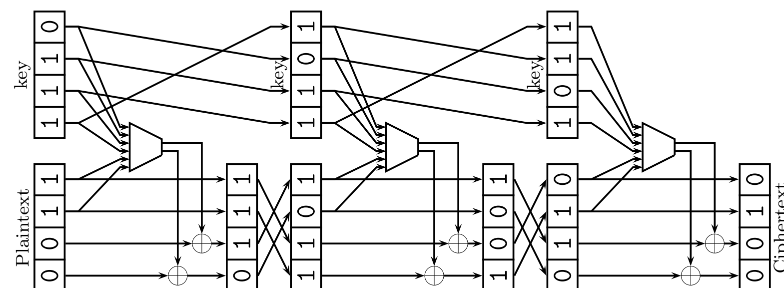


Figure 2.4: BabyBlock

Ex. 2.5.2. Decrypt the ciphertext 1110 with the key 1110 and check that you can recover the plaintext 0011. (Remember you need to start your decryption with the key rotated two positions to the left!)

Ex. 2.5.3. Encrypt the next block from the plaintext message help (1001) with the key 1110.

Security of BabyBlock

So far, we know how encryption with BabyBlock works, but it is perhaps unclear why its designed as it is. Keep in mind that modern ciphers are expected to be resistant to known-plaintext attacks. Let's see how this works in the case of BabyBlock.

Imagine we are given a string 1110 and are told that, when muddled with a certain key, it turns into 0110. Can we figure out what the key is?

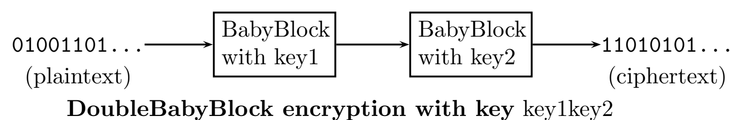
We know that the input to the S-Box was the (unknown) key and the bits 10, since those are the last 2 bits of the original string. And since the output of the S-box gives 01 (the first 2 bits of the muddled string) when added to 11 (the first 2 bits of the original string), we know that the output of the S-Box is 10. Is this enough information to figure out what the key is? Looking at the S-Box table, we see that there are four possible keys which could give rise to the output 10 when the last two bits of the block being muddled are 10: 0001, 0100, 1100, 1111, are all possibilities. Even though we knew a plaintext and a ciphertext, we haven't been able to figure out the key. This is because the S-box is not *invertible.* This means that, even if we know the output and one of the inputs, we still may not be able to figure out the other input. In spite of the fact that the S-Box is not invertible, it is still possible to decrypt messages: this is because messages are not encrypted directly with the S-Box; instead, the S-Box is only used to choose a pair of bits which will be added to plaintext bits to encrypt them. Since addition can be reversed, the encryption can be reversed, and it is possible to decrypt messages so long as we know the key.

We've seen that knowing a plaintext isn't enough to figure out the key used for the muddling operation, but it *did* narrow the choices down to just 4 keys. This is a good reason not to encrypt messages just by muddling them! (Of course, another good reason is that half the bits remain unencrypted!) When the muddling operation is repeated, however, there's not even an quick way of narrowing down to four keys. The goal for a well-designed block cipher is that there isn't a way to break the cipher much faster than just trying all possible keys. Note, however, that BabyBlock is *not* a well-designed block cipher, but is just a simplified example showing the general structure of such schemes.

There are a number of advanced techniques developed to attack block ciphers. Some poorly designed block ciphers have turned out to be very insecure, but those in most widespread use (i.e., AES) appear to be essentially as secure as the size of their keyspace (guessing the key is basically your best option). Most of the advanced techniques used to attack block ciphers are out of the scope of this course, unfortunately, but we will examine one type of attack which has had important ramifications for the use of block ciphers.

Meet-in-the-middle

As described in this section, the BabyBlock cipher has a 4-bit key, meaning that there are a total of just 16 possible keys. If we wanted more security, we might try to encrypt a message *twice* with the BabyBlock cipher, using a different key for each step. We'll call this procedure DoubleBabyBlock.



DoubleBabyBlock encryption with key key1key2

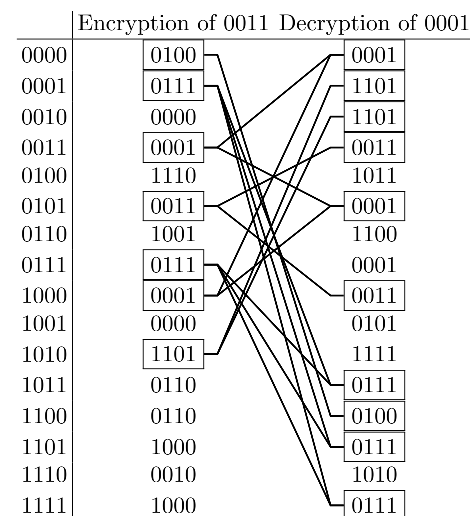
Since two 4-bit keys are used for DoubleBabyBlock, we can view DoubleBabyBlock as a cipher with a single 8-bit key which is just the concatenation of the two (if key1 is 1100 and key2 is 1010, the single 8-bit key is 11001010). There are a total of 256 possible 8-bit keys, which means that a brute force attack on DoubleBabyBlock would take up to 256 guess for the key (and 128 guesses on average).

It turns out, however, that DoubleBabyBlock does *not* provide much more security than BabyBlock, as there is a clever way to avoid having to make 256 guesses.

The **Meet-in-the-middle** attack is a known-plaintext attack. Assume we know that the plaintext HELP became the ciphertext REHN under encryption by the DoubleBabyBlock cipher with unknown key (after conversion to and from binary). The idea of the meet-in-the-middle attack is simple. If a plaintext is encrypted with a key *key1* and then a key *key2* to produce a ciphertext, then *decrypting* that ciphertext with *key2* must produce the same result as *encrypting* the plaintext with *key1*; both of these operations 'meet in the middle', at the

halfway point between the original plaintext and the doubly-encrypted ciphertext. It turns out that, by checking which pairs of keys meet up in the middle like this, we can make break the cipher with far fewer guesses than would be required for a brute force attack (which would require 256 guesses, since there are 256 possible 8-bit keys).

To illustrate the Meet-in-the-middle attack, assume we know that the first letter of the a 4-letter plaintext message is `H`, and the 4-letter ciphertext is `CYU!`. Converting to binary gives that the first block of 4 of the plaintext is `0011`, the first block of the ciphertext is `0001`. The brute force attack would now try all 256 possible key pairs to see which one encrypts `0011` to `0001`. Instead, we will try all 16 possible 4-bit keys to *encrypt* the plaintext `0011` using the regular BabyBlock algorithm, and all 16 possible 4-bit keys to *decrypt* `0001` using the regular algorithm, and see which pair of guesses give the same thing:



	Encryption of 0011	Decryption of 0001
0000	0100	0001
0001	0111	1101
0010	0000	1101
0011	0001	0011
0100	1110	1011
0101	0011	0001
0110	1001	1100
0111	0111	0001
1000	0001	0011
1001	0000	0101
1010	1101	1111
1011	0110	0111
1100	0110	0100
1101	1000	0111
1110	0010	1010
1111	1000	0111

For example, encrypting 0011 with 0000 gives the same result as decrypting 0001 with the key 1100. This means that one possible key combination that would encrypt 0011 to 0001 is 0000 followed by 1100, corresponding to the 8-bit DoubleBabyBlock key 00001100. Altogether, there are 17 8-bit keys formed by matched pairs in the above table; the 8-bit key which will decrypt the rest of the message `CYU!` must be among them. Note that it took calculating 32 BabyBlock encryptions/decryptions to make the table, and it will take at most 17 more DoubleBabyBlock decryptions to find the true key and decrypt the message. Since a BabyBlock encryption/decryption takes half as long as a DoubleBabyBlock decryption, we will be able to break the cipher with the equivalent

of at most 33 DoubleBabyBlock decryptions. Although this is quite tedious to do by hand, it is much better than having to try all 256 possible decryptions!