

Syllabus for Cryptography

Cryptography Math 178. Fall 2011. TTh 11:50 - 1:35, O'Connor 107, Section 73398

Professor Ed Schaefer, 309 O'Connor Hall, 554-6899, eschaefer@scu.edu
math.scu.edu/~eschaefer/index.html (or Google "Ed Schaefer").

Office Hours: Tuesday 9:20 - 10:20, Wednesday 9:55 - 10:55, Thursday 3 - 4:05, Friday 9:55 - 10:55.

Midterm, Thursday November 3. The final is on Tuesday, December 6 at 1:30. Your grade is broken up as: Homework 20%, Midterm 30%, Final 50%.

Cryptography is about hiding information. As technology becomes increasingly involved in communication, cryptography becomes increasingly important. There are issues of security relating to cellular phones, e-mail, ATM pin numbers, making purchases over the internet, electronic cash, financial transactions over phone lines, communications among spies and among world leaders, keeping hackers out of bank accounts, authenticating electronic signatures and more.

In this course, we will study symmetric key and public key cryptography. Symmetric key cryptography is the classical situation in which the two proper users (the sender and the receiver) must agree on the key for a cryptosystem ahead of time, in order to be able to encrypt and decrypt messages. This will include classical cryptosystems, stream ciphers and block ciphers. Then we will study public key cryptography, invented in the mid 1970's by Whitfield Diffie and Martin Hellman. It enables two proper users to encrypt and decrypt messages without agreeing on a key ahead of time. It also enables digital signatures and verifying that a message has not been tampered with.

For those interested, I recommend further reading from RSA's FAQ on line.

End syllabus

Cryptography computer labs manual

The program CryptoSoft was written by Cameron Wong (who took this class in 2009) and me. It contains programs for this class. GP-PARI was written by 4 Frenchmen who are both number theorists and computer scientists. The Windows version of GP-PARI is an afterthought (ah UNIX!), so if you have a home computer/laptop, you may have trouble running it at home, if so, see me. You can go on-line to download a UNIX version. I'm not sure whether there is a MAC version.

CryptoSoft: This program has 3 options at first: Encode/decode, Encrypt, Decrypt.

Encode/Decode At *Input Type* you can choose Plaintext, Binary, Integer or Hex.

If you choose Plaintext then you can type the characters a - z, A - Z, 0 - 9, comma, space and period. This field will accept no other characters **including** enter/carriage return. If you then hit *convert* it will turn this plaintext into the corresponding ASCII string, which will appear in the field Binary. ASCII is an international standard encoding of characters into bytes (strings of 8 bits). The ASCII encoding can be found on a later handout in this document. This bit string is the base 2 representation of an integer, that will show up in the field Integer. This bit string can also be encoded as a Hex string. Hex is an encoding of nibbles (strings of 4 bits). The Hex encoding can also be found on a later handout in this document.

If you choose Binary then you can type 0's and 1's. It will also accept arrays of 0's and 1's like [0,1,1,0,1,1,1,1,0,1,0,0,0,1]. Then convert. It turns it into an integer, as above. If the length of the bit string is a multiple of 8 then it will do an ASCII decoding. If the length is a multiple of 4 then it will do a Hex decoding.

If you choose Integer then you can type an integer (without commas). Then convert. Its base 2 representation will appear in binary with 0's added to the beginning so the bit string will have length a multiple of 8. The ASCII decoding and Hex encoding of this bit string will appear in those fields.

If you choose Hex then you can type 0 - 9, A - F. Then convert. The Hex decoding will appear in Binary. That bit string is decoded to Plaintext and turned into an Integer (as described above).

If you pick Encrypt it will encrypt with Simplified AES in Cipher Block Chaining Mode with initialization vector 1100101111101000 (which will make sense to you about a month into the course - until then, think of it as a black box encryption). First you enter a key in binary, integer or hex form. Then enter plaintext, which again must only consist of a - z, A-Z, 0 - 9, comma, period and space. Decrypt is similar.

PARI is essentially a calculator. Annoyingly the basic Windows version does not allow you to use a mouse to copy, cut and paste things. It can be run through Emacs and perhaps the Unix version has this functionality (you'll have to explore that on your own, if you like). So you will sometimes need to read things into it and write things to files. The 2 most important things to know about PARI are to never end the name of an output file with *.out*, always end them with *.txt* like *a.txt* and use only letters and a period in the filename. The second is that you type $\backslash q$ to get out. If PARI gets hung up because you ask it to do something too hard (like factor a 100 digit number) then you will have to kill it.

PARI does all the usual operations $+$, $-$, $*$, $/$, \wedge for plus, minus, times, divides, and powers. You can use parentheses too. Below will be a short PARI session, what I type is boldface.

```
? 2+2
%1 = 4
? 3^5
%2 = 243
? %1+%2
%3 = 247
```

Comment: Mod(a,m) means $a \pmod m$; you must use a capital M. Back to PARI:

```
? Mod(78,5)
%4=Mod(3,5)
```

Comment: To do powers mod m, do the power outside. To reduce $2^{1000000} \pmod{77}$ do the following:

```
? Mod(2,77)^1000000
%5 = Mod(23,77)
? Mod(2,7)^-1
%6 = Mod(4,7)
```

Comment: nextprime(a) gives the next prime $\geq a$. I like to give things names other than the ones they already have with the % signs. This is a matter of taste. When numbers have names you don't have to retype them, if you use the same name for 2 different numbers, it will only remember the last one (like in BASIC). Back to PARI:

```
? a=nextprime(100)
%7 = 101
? b=nextprime(200)
%8 = 211
? n=a*b
%9 = 21311
? gcd(a,b)
%10 = 1
```

Comment: If $d = \text{Mod}(10,35)$ then the output of lift(d) is 10, it pulls the c out of $\text{mod}(c,n)$.

```
? d=Mod(100,2345)
%11=Mod(100,2345)
? e=lift(d)
%12=100
? d
%12=Mod(100,2345)
? e
%13=100
```

Do not start a variable name with a number. r2 is OK, 2r is not. Do not use math operations in variable names. a+b is not an OK VARIABLE name.

To find out about PARI's other functions if you are interested, type ?

Reading from and writing to files

Say the file c.txt already exists in your directory (may be output from *letters to numbers*, for example). It has 3 numbers on 3 lines and looks like the following:

12
15
17

Let's start a new session of PARI:

```
? \r c.txt
%1=12
%2=15
%3=17
? %1*%3
%4=204
```

In PARI if you type `\w d.txt` after the line `%n=m` it will write `m` to the file `d.txt`.

Example:

```
? a=16
%1=16
? \w d.txt
? b=14
%2=14
? \w d.txt
? n=a*b
%3=224
? \w d.txt
```

Then there will be a file called `d.txt` left in your directory that will have 3 lines, the numbers 16, 14 and 224.

Warning: there are already files called `n.txt` and `e.txt` on your CD so avoid these names for files that you create.

You can type `\l` while in PARI and everything thereafter will appear in a file called `pari.log`. If you want to play with PARI then type `?` and explore. You can program within PARI. A document explaining how is at my website:
math.scu.edu/~eschaefer/index.html.

End computer labs manual

Encoding handout

ASCII

00100000	32	8	00111000	56	P	01010000	80	h	01101000	104	
!	00100001	33	9	00111001	57	Q	01010001	81	i	01101001	105
"	00100010	34	:	00111010	58	R	01010010	82	j	01101010	106
#	00100011	35	;	00111011	59	S	01010011	83	k	01101011	107
\$	00100100	36	<	00111100	60	T	01010100	84	l	01101100	108
%	00100101	37	=	00111101	61	U	01010101	85	m	01101101	109
&	00100110	38	>	00111110	62	V	01010110	86	n	01101110	110
'	00100111	39	?	00111111	63	W	01010111	87	o	01101111	111
(00101000	40	@	01000000	64	X	01011000	88	p	01110000	112
)	00101001	41	A	01000001	65	Y	01011001	89	q	01110001	113
*	00101010	42	B	01000010	66	Z	01011010	90	r	01110010	114
+	00101011	43	C	01000011	67	[01011011	91	s	01110011	115
,	00101100	44	D	01000100	68	\	01011100	92	t	01110100	116
-	00101101	45	E	01000101	69]	01011101	93	u	01110101	117
.	00101110	46	F	01000110	70	^	01011110	94	v	01110110	118
/	00101111	47	G	01000111	71	_	01011111	95	w	01110111	119
0	00110000	48	H	01001000	72	`	01100000	96	x	01111000	120
1	00110001	49	I	01001001	73	a	01100001	97	y	01111001	121
2	00110010	50	J	01001010	74	b	01100010	98	z	01111010	122
3	00110011	51	K	01001011	75	c	01100011	99	{	01111011	123
4	00110100	52	L	01001100	76	d	01100100	100		01111100	124
5	00110101	53	M	01001101	77	e	01100101	101	}	01111101	125
6	00110110	54	N	01001110	78	f	01100110	102	~	01111110	126
7	00110111	55	O	01001111	79	g	01100111	103			

Hex encodings

0000	0	0001	1	0010	2	0011	3
0100	4	0101	5	0110	6	0111	7
1000	8	1001	9	1010	A	1011	B
1100	C	1101	D	1110	E	1111	F

Alphabet encodings

a	0	00000	b	1	00001	c	2	00010	d	3	00011	e	4	00100
f	5	00101	g	6	00110	h	7	00111	i	8	01000	j	9	01001
k	10	01010	l	11	01011	m	12	01100	n	13	01101	o	14	01110
p	15	01111	q	16	10000	r	17	10001	s	18	10010	t	19	10011
u	20	10100	v	21	10101	w	22	10110	x	23	10111	y	24	11000
z	25	11001												

End of Encoding handout

The Block Cipher AES

Introduction

The U.S. government in the early 1970's wanted an encryption process on a small chip that would be widely used and safe. In 1975 they accepted IBM's Data Encryption Standard Algorithm (DES). DES is a symmetric-key cryptosystem which has a 56-bit key and encrypts 64-bit plaintexts to 64-bit ciphertexts. By the early 1990's, the 56-bit key was considered too short. So people started using Triple-DES. That is using DES three times (with two different keys, first, second, first, so we have 112 key bits) to encrypt a 64-bit plaintext. (Surprisingly, Double-DES with two different keys is not much safer than DES, as I'll explain in the Cryptanalysis course). In 1997 DES was brute forced in 24 hours by 500000 computers. However, DES was not designed with Triple-DES in mind. Undoubtedly there would be a more efficient algorithm with the same level of safety as Triple-DES. So in 1997, the National Institute of Standards and Technology (NIST) solicited proposals for replacements of DES. In 2001, NIST chose 128-bit block Rijndael with a 128-bit key to become the Advanced Encryption Standard (AES). (If you don't speak Dutch, Flemish or Afrikaans, then the closest approximation to the pronunciation is Rine-doll). Rijndael is a symmetric-key block cipher designed by Joan Daemen and Vincent Rijmen.

Simplified AES

Simplified AES was designed by Mohammad Musa, Steve Wedig (two former Crypto students) and me in 2002. It is a method of teaching AES. We published the article *A simplified AES algorithm and its linear and differential cryptanalyses* in the journal *Cryptologia* in 2003. We will learn the linear and differential cryptanalyses in the Cryptanalysis Course.

The Finite Field

Both the key expansion and encryption algorithms of simplified AES depend on an S-box that itself depends on the finite field with 16 elements.

Let $\mathbf{F}_{16} = \mathbf{F}_2[x]/(x^4 + x + 1)$. The word nibble refers to a four-bit string, like 1011. We will frequently associate an element $b_0x^3 + b_1x^2 + b_2x + b_3$ of \mathbf{F}_{16} with the nibble $b_0b_1b_2b_3$.

The S-box

The S-box is a map from nibbles to nibbles. It can be inverted. (For those in the know, it is one-to-one and onto or bijective.) Here is how it operates. First, invert the nibble in \mathbf{F}_{16} . The inverse of $x + 1$ is $x^3 + x^2 + x$ so 0011 goes to 1110. The nibble 0000 is not invertible, so at this step it is sent to itself. Then associate to the nibble $N = b_0b_1b_2b_3$ (which is the output of the inversion) the element $N(y) = b_0y^3 + b_1y^2 + b_2y + b_3$ in $\mathbf{F}_2[y]/(y^4 + 1)$. Let $a(y) = y^3 + y^2 + 1$ and $b(y) = y^3 + 1$ in $\mathbf{F}_2[y]/(y^4 + 1)$. Doing multiplication and addition is similar to doing so in \mathbf{F}_{16} except that we are working modulo $y^4 + 1$ so $y^4 = 1$ and $y^5 = y$ and $y^6 = y^2$. The second step of the S-box is to send the nibble $N(y)$ to $a(y)N(y) + b(y)$. So the nibble 1110 = $y^3 + y^2 + y$ goes to $(y^3 + y^2 + 1)(y^3 + y^2 + y) + (y^3 + 1) = (y^6 + y^5 + y^4) + (y^5 + y^4 + y^3) + (y^3 + y^2 + y) + (y^3 + 1) = y^2 + y + 1 + y + 1 + y^3 + y^3 + y^2 + y + y^3 + 1 = 3y^3 + 2y^2 + 3y + 3 = y^3 + y + 1 = 1011$. So S-box(0011) = 1011.

Note that $y^4 + 1 = (y + 1)^4$ is reducible over \mathbf{F}_2 so $\mathbf{F}_2[y]/(y^4 + 1)$ is not a field and not all of its non-zero elements are invertible; the polynomial $a(y)$, however, is. So $N(y) \mapsto$

$a(y)N(y) + b(y)$ is an invertible map. If you read the literature, then the second step is often described by an affine matrix map.

We can represent the action of the S-box in two ways (note we do not show the intermediary output of the inversion in \mathbf{F}_{16}^*). These are called look up tables.

nib	S-box(nib)	nib	S-box(nib)	
0000	1001	1000	0110	
0001	0100	1001	0010	
0010	1010	1010	0000	
0011	1011	1011	0011	
0100	1101	1100	1100	or
0101	0001	1101	1110	$\left[\begin{array}{cccc} 9 & 4 & 10 & 11 \\ 13 & 1 & 8 & 5 \\ 6 & 2 & 0 & 3 \\ 12 & 14 & 15 & 7 \end{array} \right]$
0110	1000	1110	1111	
0111	0101	1111	0111	

The left-hand side is most useful for doing an example by hand. For the matrix on the right, we start in the upper left corner and go across, then to the next row and go across etc. The integers 0 - 15 are associated with their 4-bit binary representations. So 0000 = 0 goes to 9 = 1001, 0001 = 1 goes to 4 = 0100, ..., 0100 = 4 goes to 13 = 1101, etc.

Keys

For our simplified version of AES, we have a 16-bit key, which we denote $k_0 \dots k_{15}$. That needs to be expanded to a total of 48 key bits $k_0 \dots k_{47}$, where the first 16 key bits are the same as the original key. Let us describe the expansion. Let $\text{RC}[i] = x^{i+2} \in \mathbf{F}_{16}$. So $\text{RC}[1] = x^3 = 1000$ and $\text{RC}[2] = x^4 = x + 1 = 0011$. If N_0 and N_1 are nibbles, then we denote their concatenation by N_0N_1 . Let $\text{RCN}[i] = \text{RC}[i]0000$ (this is a byte, a string of 8 bits). So $\text{RCN}[1] = 10000000$ and $\text{RCN}[2] = 00110000$. These are abbreviations for *round constant*. We define the function RotNib to be $\text{RotNib}(N_0N_1) = N_1N_0$ and the function SubNib to be $\text{SubNib}(N_0N_1) = \text{S-box}(N_0)\text{S-box}(N_1)$; these are functions from bytes to bytes. Their names are abbreviations for *rotate nibble* and *substitute nibble*. Let us define an array (vector, if you prefer) W whose entries are bytes. The original key fills $W[0]$ and $W[1]$ in order. For $2 \leq i \leq 5$,

$$\begin{aligned} \text{if } i \equiv 0 \pmod{2} & \text{ then } W[i] = W[i-2] \oplus \text{RCN}(i/2) \oplus \text{SubNib}(\text{RotNib}(W[i-1])) \\ \text{if } i \not\equiv 0 \pmod{2} & \text{ then } W[i] = W[i-2] \oplus W[i-1] \end{aligned}$$

The bits contained in the entries of W can be denoted $k_0 \dots k_{47}$. For $0 \leq i \leq 2$ we let $K_i = W[2i]W[2i+1]$. So $K_0 = k_0 \dots k_{15}$, $K_1 = k_{16} \dots k_{31}$ and $K_2 = k_{32} \dots k_{47}$. For $i \geq 1$, K_i is the round key used at the end of the i -th round; K_0 is used before the first round.

Recall \oplus denotes bit-by-bit XORing.

Key Expansion Example

Let's say that the key is 0101 1001 0111 1010. So $W[0] = 01011001$ and $W[1] = 01111010$. Now $i = 2$ so we $\text{Rotnib}(W[1])=1010\ 0111$. Then we $\text{SubNib}(1010\ 0111)=0000$

0101. Then we XOR this with $W[0] \oplus \text{RCON}(1)$ and get $W[2]$.

$$\begin{array}{r} 0000 \ 0101 \\ 0101 \ 1001 \\ \oplus \ \underline{1000} \ \underline{0000} \\ 1101 \ 1100 \end{array}$$

So $W[2] = 11011100$.

Now $i = 3$ so $W[3] = W[1] \oplus W[2] = 0111 \ 1010 \oplus 1101 \ 1100 = 1010 \ 0110$. Now $i = 4$ so we $\text{Rotnib}(W[3])=0110 \ 1010$. Then we $\text{SubNib}(0110 \ 1010)=1000 \ 0000$. Then we XOR this with $W[2] \oplus \text{RCON}(2)$ and get $W[4]$.

$$\begin{array}{r} 1000 \ 0000 \\ 1101 \ 1100 \\ \oplus \ \underline{0011} \ \underline{0000} \\ 0110 \ 1100 \end{array}$$

So $W[4] = 01101100$.

Now $i = 5$ so $W[5] = W[3] \oplus W[4] = 1010 \ 0110 \oplus 0110 \ 1100 = 1100 \ 1010$.

The Simplified AES Algorithm

The simplified AES algorithm operates on 16-bit plaintexts and generates 16-bit ciphertexts, using the expanded key $k_0 \dots k_{47}$. The encryption algorithm consists of the composition of 8 functions applied to the plaintext: $A_{K_2} \circ SR \circ NS \circ A_{K_1} \circ MC \circ SR \circ NS \circ A_{K_0}$ (so A_{K_0} is applied first), which will be described below. Each function operates on a state. A state consists of 4 nibbles configured as in Figure 1. The initial state consists of the plaintext as in Figure 2. The final state consists of the ciphertext as in Figure 3.

$b_0b_1b_2b_3$	$b_8b_9b_{10}b_{11}$
$b_4b_5b_6b_7$	$b_{12}b_{13}b_{14}b_{15}$

Figure 1

$p_0p_1p_2p_3$	$p_8p_9p_{10}p_{11}$
$p_4p_5p_6p_7$	$p_{12}p_{13}p_{14}p_{15}$

Figure 2

$c_0c_1c_2c_3$	$c_8c_9c_{10}c_{11}$
$c_4c_5c_6c_7$	$c_{12}c_{13}c_{14}c_{15}$

Figure 3

The Function A_{K_i} : The abbreviation A_K stands for *add key*. The function A_{K_i} consists of XORing K_i with the state so that the subscripts of the bits in the state and the key bits agree modulo 16.

The Function NS : The abbreviation NS stands for *nibble substitution*. The function NS replaces each nibble N_i in a state by $\text{S-box}(N_i)$ without changing the order of the nibbles. So it sends the state

$$\begin{array}{|c|c|} \hline N_0 & N_2 \\ \hline N_1 & N_3 \\ \hline \end{array} \text{ to the state } \begin{array}{|c|c|} \hline \text{S-box}(N_0) & \text{S-box}(N_2) \\ \hline \text{S-box}(N_1) & \text{S-box}(N_3) \\ \hline \end{array}.$$

The Function SR : The abbreviation SR stands for *shift row*. The function SR takes the state

$$\begin{array}{|c|c|} \hline N_0 & N_2 \\ \hline N_1 & N_3 \\ \hline \end{array} \text{ to the state } \begin{array}{|c|c|} \hline N_0 & N_2 \\ \hline N_3 & N_1 \\ \hline \end{array}.$$

The Function MC : The abbreviation MC stands for *mix column*. A column $[N_i, N_j]$ of the state is considered to be the element $N_i z + N_j$ of $\mathbf{F}_{16}[z]/(z^2 + 1)$. As an example,

if the column consists of $[N_i, N_j]$ where $N_i = 1010$ and $N_j = 1001$ then that would be $(x^3 + x)z + (x^3 + 1)$. Like before, $\mathbf{F}_{16}[z]$ denotes polynomials in z with coefficients in \mathbf{F}_{16} . So $\mathbf{F}_{16}[z]/(z^2 + 1)$ means that polynomials are considered modulo $z^2 + 1$; thus $z^2 = 1$. So representatives consist of the 16^2 polynomials of degree less than 2 in z .

The function MC multiplies each column by the polynomial $c(z) = x^2z + 1$. As an example,

$$\begin{aligned} & [((x^3 + x)z + (x^3 + 1))(x^2z + 1)] = (x^5 + x^3)z^2 + (x^3 + x + x^5 + x^2)z + (x^3 + 1) \\ & = (x^5 + x^3 + x^2 + x)z + (x^5 + x^3 + x^3 + 1) = (x^2 + x + x^3 + x^2 + x)z + (x^2 + x + 1) \\ & = (x^3)z + (x^2 + x + 1), \end{aligned}$$

which goes to the column $[N_k, N_l]$ where $N_k = 1000$ and $N_l = 0111$.

Note that $z^2 + 1 = (z + 1)^2$ is reducible over \mathbf{F}_{16} so $\mathbf{F}_{16}[z]/(z^2 + 1)$ is not a field and not all of its non-zero elements are invertible; the polynomial $c(z)$, however, is.

The simplest way to explain MC is to note that MC sends a column

$$\begin{array}{|c|} \hline b_0b_1b_2b_3 \\ \hline b_4b_5b_6b_7 \\ \hline \end{array} \text{ to } \begin{array}{|cccc|} \hline b_0 \oplus b_6 & b_1 \oplus b_4 \oplus b_7 & b_2 \oplus b_4 \oplus b_5 & b_3 \oplus b_5 \\ \hline b_2 \oplus b_4 & b_0 \oplus b_3 \oplus b_5 & b_0 \oplus b_1 \oplus b_6 & b_1 \oplus b_7 \\ \hline \end{array}.$$

The Rounds: The composition of functions $A_{K_i} \circ MC \circ SR \circ NS$ is considered to be the i -th round. So this simplified algorithm has two rounds. There is an extra A_K before the first round and the last round does not have an MC ; the latter will be explained in the next section.

Decryption

Note that for general functions (where the composition and inversion are possible) $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$. Also, if a function composed with itself is the identity map (i.e. gets you back where you started), then it is its own inverse; this is called an involution. This is true of each A_{K_i} . Although it is true for our SR , this is not true for the real SR in AES, so we will not simplify the notation SR^{-1} . Decryption is then by $A_{K_0} \circ NS^{-1} \circ SR^{-1} \circ MC^{-1} \circ A_{K_1} \circ NS^{-1} \circ SR^{-1} \circ A_{K_2}$.

To accomplish NS^{-1} , multiply a nibble by $a(y)^{-1} = y^2 + y + 1$ and add $a(y)^{-1}b(y) = y^3 + y^2$ in $\mathbf{F}_2[y]/(y^4 + 1)$. Then invert the nibble in \mathbf{F}_{16} . Alternately, we can simply use one of the S-box tables in reverse.

Since MC is multiplication by $c(z) = x^2z + 1$, the function MC^{-1} is multiplication by $c(z)^{-1} = xz + (x^3 + 1)$ in $\mathbf{F}_{16}[z]/(z^2 + 1)$.

Decryption can be done as above. However to see why there is no MC in the last round, we continue. First note that $NS^{-1} \circ SR^{-1} = SR^{-1} \circ NS^{-1}$. Let St denote a state. We have $MC^{-1}(A_{K_i}(St)) = MC^{-1}(K_i \oplus St) = c(z)^{-1}(K_i \oplus St) = c(z)^{-1}(K_i) \oplus c(z)^{-1}(St) = c(z)^{-1}(K_i) \oplus MC^{-1}(St) = A_{c(z)^{-1}K_i}(MC^{-1}(St))$. So $MC^{-1} \circ A_{K_i} = A_{c(z)^{-1}K_i} \circ MC^{-1}$.

What does $c(z)^{-1}(K_i)$ mean? Break K_i into two bytes $b_0b_1 \dots b_7, b_8, \dots b_{15}$. Consider the first byte

$$\begin{array}{|c|} \hline b_0b_1b_2b_3 \\ \hline b_4b_5b_6b_7 \\ \hline \end{array}$$

to be an element of $\mathbf{F}_{16}[z]/(z^2 + 1)$. Multiply by $c(z)^{-1}$, then convert back to a byte. Do the same with $b_8 \dots b_{15}$. So $c(z)^{-1}K_i$ has 16 bits. $A_{c(z)^{-1}K_i}$ means XOR $c(z)^{-1}K_i$ with the current state. Note when we do MC^{-1} , we will multiply the state by $c(z)^{-1}$ (or more easily, use the equivalent table that you will create in your homework). For $A_{c(z)^{-1}K_1}$, you will first multiply K_1 by $c(z)^{-1}$ (or more easily, use the equivalent table that you will create in your homework), then XOR the result with the current state.

Thus decryption is also

$$A_{K_0} \circ SR^{-1} \circ NS^{-1} \circ A_{c(z)^{-1}K_1} \circ MC^{-1} \circ SR^{-1} \circ NS^{-1} \circ A_{K_2}.$$

Recall that encryption is

$$A_{K_2} \circ SR \circ NS \circ A_{K_1} \circ MC \circ SR \circ NS \circ A_{K_0}.$$

Notice how each kind of operation for decryption appears in exactly the same order as in encryption, except that the round keys have to be applied in reverse order. For the real AES, this can improve implementation. This would not be possible if MC appeared in the last round.

Encryption Example

Let's say that we use the key in the above example 0101 1001 0111 1010. So $W[0] = 0101\ 1001$, $W[1] = 0111\ 1010$, $W[2] = 1101\ 1100$, $W[3] = 1010\ 0110$, $W[4] = 0110\ 1100$, $W[5] = 1100\ 1010$,

Let's say that the plaintext is my name 'Ed' in ASCII: 01000101 01100100 Then the initial state is (remembering that the nibbles go in upper left, **then lower left**, then upper right, then lower right)

0100	0110
0101	0100

Then we do A_{K_0} (recall $K_0 = W[0]W[1]$) to get a new state:

0100	0110	=	0001	0001
\oplus 0101	\oplus 0111		1100	1110
0101	0100			
\oplus 1001	\oplus 1010			

Then we apply NS and SR to get

0100	0100	$\rightarrow SR \rightarrow$	0100	0100
1100	1111		1111	1100

Then we apply MC to get

1101	0001
1100	1111

Then we apply A_{K_1} , recall $K_1 = W[2]W[3]$.

1101	0001	=	0000	1011
\oplus 1101	\oplus 1010		0000	1001
1100	1111			
\oplus 1100	\oplus 0110			

Then we apply NS and SR to get

$$\begin{array}{|c|c|} \hline 1001 & 0011 \\ \hline 1001 & 0010 \\ \hline \end{array} \quad \rightarrow SR \rightarrow \quad \begin{array}{|c|c|} \hline 1001 & 0011 \\ \hline 0010 & 1001 \\ \hline \end{array}$$

Then we apply A_{K_2} , recall $K_2 = W[4]W[5]$.

$$\begin{array}{|c|c|} \hline 1001 & 0011 \\ \oplus 0110 & \oplus 1100 \\ \hline 0010 & 1001 \\ \oplus 1100 & \oplus 1010 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1111 & 1111 \\ \hline 1110 & 0011 \\ \hline \end{array}$$

So the ciphertext is 11111110 11110011.

The Real AES

For simplicity, we will describe the version of AES that has a 128-bit key and has 10 rounds. Recall that the AES algorithm operates on 128-bit blocks. We will mostly explain the ways in which it differs from our simplified version. Each state consists of a four-by-four grid of bytes.

The finite field is $\mathbf{F}_{2^8} = \mathbf{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$. We let the byte $b_0b_1b_2b_3b_4b_5b_6b_7$ and the element $b_0x^7 + \dots + b_7$ of \mathbf{F}_{2^8} correspond to each other. The S-box first inverts a byte in \mathbf{F}_{2^8} and then multiplies it by $a(y) = y^4 + y^3 + y^2 + y + 1$ and adds $b(y) = y^6 + y^5 + y + 1$ in $\mathbf{F}_2[y]/(y^8 + 1)$. Note $a(y)^{-1} = y^6 + y^3 + y$ and $a(y)^{-1}b(y) = y^2 + 1$.

The real ByteSub is the obvious generalization of our NS - it replaces each byte by its image under the S-box. The real ShiftRow shifts the rows left by 0, 1, 2 and 3. So it sends the state

$$\begin{array}{|c|c|c|c|} \hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_1 & B_5 & B_9 & B_{13} \\ \hline B_2 & B_6 & B_{10} & B_{14} \\ \hline B_3 & B_7 & B_{11} & B_{15} \\ \hline \end{array} \quad \text{to the state} \quad \begin{array}{|c|c|c|c|} \hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_5 & B_9 & B_{13} & B_1 \\ \hline B_{10} & B_{14} & B_2 & B_6 \\ \hline B_{15} & B_3 & B_7 & B_{11} \\ \hline \end{array}.$$

The real MixColumn multiplies a column by $c(z) = (x+1)z^3 + z^2 + z + x$ in $\mathbf{F}_{2^8}[z]/(z^4 + 1)$. Also $c(z)^{-1} = (x^3 + x + 1)z^3 + (x^3 + x^2 + 1)z^2 + (x^3 + 1)z + (x^3 + x^2 + x)$. The MixColumn step appears in all but the last round. The real AddRoundKey is the obvious generalization of our A_{K_i} . There is an additional AddRoundKey with round key 0 at the beginning of the encryption algorithm.

For key expansion, the entries of the array W are four bytes each. The key fills in $W[0], \dots, W[3]$. The function RotByte cyclically rotates four bytes 1 to the left each, like the action on the second row in ShiftRow. The function SubByte applies the S-box to each byte. $RC[i] = x^i$ in \mathbf{F}_{2^8} and $RCON[i]$ is the concatenation of $RC[i]$ and 3 bytes of all 0's. For $4 \leq i \leq 43$,

$$\begin{array}{ll} \text{if } i \equiv 0 \pmod{4} & \text{then } W[i] = W[i-4] \oplus RCON(i/4) \oplus \text{SubByte}(\text{RotByte}(W[i-1])) \\ \text{if } i \not\equiv 0 \pmod{4} & \text{then } W[i] = W[i-4] \oplus W[i-1]. \end{array}$$

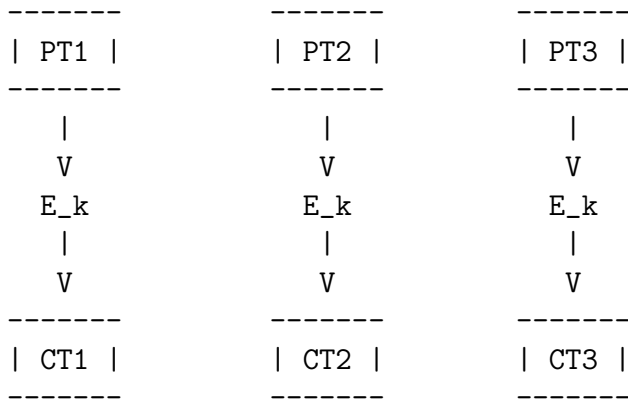
The i -th key K_i consists of the bits contained in the entries of $W[4i] \dots W[4i+3]$.

AES as a product cipher

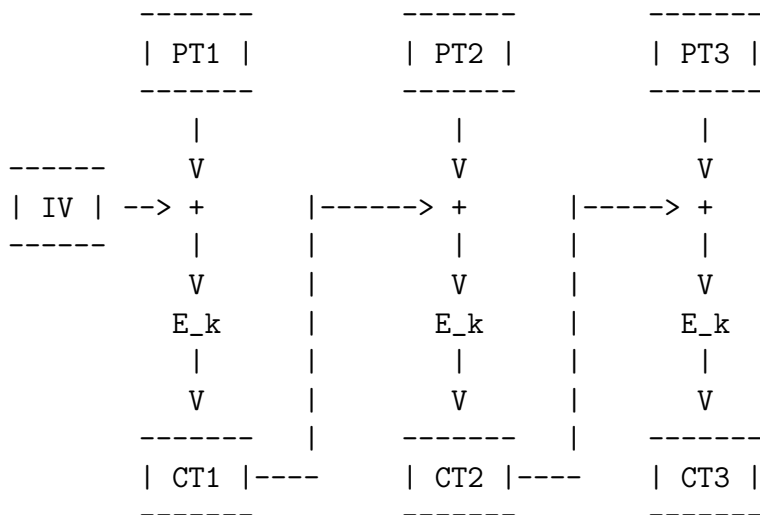
Note that there is transposition by row using ShiftRow. Though it is not technically transposition, there is dispersion by column using MixColumn. The substitution is accomplished with ByteSub and AddRoundKey makes the algorithm key-dependent.

Modes of operation for a block cipher

There are four modes of operation on an AES chip (this is true for any block cipher). The standard mode is the *electronic code book* (ECB) mode. It is the most straightforward but has the disadvantage that for a given key, two identical plaintexts will correspond to identical ciphertexts. If the number of bits in the plaintext message is not a multiple of 128, then padding (extra bits, probably starting with a 1, since normal ASCII characters don't) are added so that the last plaintext block has 128 bits as well.

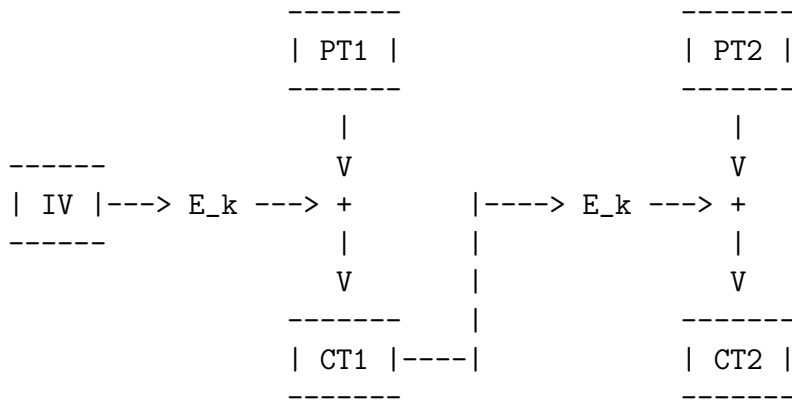


The next mode is the *cipherblock chaining* (CBC) mode. This is the most commonly used mode. Alice and Bob must agree on a non-secret 128-bit initialization vector (IV) of random bits. There should be a new IV for each session.

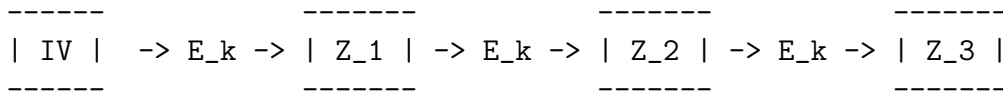


The next mode is the *cipher feedback* (CFB) mode. IV again denotes a non-secret 128-bit initialization vector that the two users must agree upon ahead of time and that should

change each time. If the plaintext is coming in slowly, the ciphertext can be sent as soon as as the plaintext comes in. With the CBC mode, one must wait for a whole 128 bit plaintext block before computing the ciphertext. This is also a good mode of you do not want to pad the plaintext.



The last mode is the *output feedback* (OFB) mode. It is a way to create a keystream for a stream cipher. Below is how you create the keystream. IV again denotes a non-secret 128-bit initialization vector that the two users must agree upon ahead of time.



The keystream is the concatenation of $Z_1Z_2Z_3\dots$. As usual, this will be XORed with the plaintext. (In the diagram you can add PT_i 's, CT_i 's and \oplus 's.)

Analysis of Simplified AES

We want to look at attacks on the ECB mode of simplified AES.

The enemy intercepts a matched plaintext/ciphertext pair and wants to solve for the key. Let's say the plaintext is $p_0 \dots p_{15}$, the ciphertext is $c_0 \dots c_{15}$ and the key is $k_0 \dots k_{15}$. There are 15 equations of the form

$$f_i(p_0, \dots, p_{15}, k_0, \dots, k_{15}) = c_i$$

where f_i is a polynomial in 32 variables, with coefficients in \mathbf{F}_2 which can be expected to have 2^{31} terms on average. Once we fix the c_j 's and p_j 's (from the known matched plaintext/ciphertext pair) we get 16 non-linear equations in 16 unknowns (the k_i 's). On average these equations should have 2^{15} terms.

Everything in simplified AES is a linear map except for the S-boxes. Let us consider how they operate. Let us denote the input nibble of an S-box by $abcd$ and the output nibble as

$efgh$. Then the operation of the S-boxes can be computed with the following equations

$$\begin{aligned}e &= acd + bcd + ab + ad + cd + a + d + 1 \\f &= abd + bcd + ab + ac + bc + cd + a + b + d \\g &= abc + abd + acd + ab + bc + a + c \\h &= abc + abd + bcd + acd + ac + ad + bd + a + c + d + 1\end{aligned}$$

where all additions are modulo 2. Alternating the linear maps with these non-linear maps leads to very complicated polynomial expressions for the ciphertext bits.

Solving a system of linear equations in several variables is very easy. However, there are no known algorithms for quickly solving systems of non-linear polynomial equations in several variables.

Design Rationale

The quality of an encryption algorithm is judged by two main criteria, security and efficiency. In designing AES, Rijmen and Daemen focused on these qualities. They also instilled the algorithm with simplicity and repetition. Security is measured by how well the encryption withstands all known attacks. Efficiency is defined as the combination of encryption/decryption speed and how well the algorithm utilizes resources. These resources include required chip area for hardware implementation and necessary working memory for software implementation. Simplicity refers to the complexity of the cipher's individual steps and as a whole. If these are easy to understand, proper implementation is more likely. Lastly, repetition refers to how the algorithm makes repeated use of functions.

In the following two sections, we will discuss the concepts security, efficiency, simplicity, and repetition with respect to the real AES algorithm.

Security

As an encryption standard, AES needs to be resistant to all known cryptanalytic attacks. Thus, AES was designed to be resistant against these attacks, especially differential and linear cryptanalysis. To ensure such security, block ciphers in general must have diffusion and non-linearity.

Diffusion is defined by the spread of the bits in the cipher. Full diffusion means that each bit of a state depends on every bit of a previous state. In AES, two consecutive rounds provide full diffusion. The ShiftRow step, the MixColumn step, and the key expansion provide the diffusion necessary for the cipher to withstand known attacks.

Non-linearity is added to the algorithm with the S-Box, which is used in ByteSub and the key expansion. The non-linearity, in particular, comes from inversion in a finite field. This is not a linear map from bytes to bytes. By linear, I mean a map that can be described as map from bytes (i.e. the 8-dimensional vector space over the field \mathbf{F}_2) to bytes which can be computed by multiplying a byte by an 8×8 -matrix and then adding a vector.

Non-linearity increases the cipher's resistance against cryptanalytic attacks. The non-linearity in the key expansion makes it so that knowledge of a part of the cipher key or a round key does not easily enable one to determine many other round key bits.

Simplicity helps to build a cipher's credibility in the following way. The use of simple steps leads people to believe that it is easier to break the cipher and so they attempt to do so. When many attempts fail, the cipher becomes better trusted.

Although repetition has many benefits, it can also make the cipher more vulnerable to certain attacks. The design of AES ensures that repetition does not lead to security holes. For example, the round constants break patterns between the round keys.

Efficiency

AES is expected to be used on many machines and devices of various sizes and processing powers. For this reason, it was designed to be versatile. Versatility means that the algorithm works efficiently on many platforms, ranging from desktop computers to embedded devices such as cable boxes.

The repetition in the design of AES allows for parallel implementation to increase speed of encryption/decryption. Each step can be broken into independent calculations because of repetition. ByteSub is the same function applied to each byte in the state. MixColumn and ShiftRow work independently on each column and row in the state respectively. The AddKey function can be applied in parallel in several ways.

Repetition of the order of steps for the encryption and decryption processes allows for the same chip to be used for both processes. This leads to reduced hardware costs and increased speed.

Simplicity of the algorithm makes it easier to explain to others, so that the implementation will be obvious and flawless. The coefficients of each polynomial were chosen to minimize computation.

AES vs RC4. Block ciphers more flexible, have different modes. Can turn block cipher into stream cipher but not vice versa. RC4 1.77 times as fast as AES. Less secure.

End of AES handout

RSA Signature example handout

Signatures with RSA. Remember: When sending, small n then big n .

Let's say that $n_G = 221$, $e_G = 187$, $d_G = 115$ and $n_T = 209$, $e_T = 191$, $d_T = 131$.

T wants to sign and encrypt the plaintext message 97 for G. What does he do? In sending you work with the small n then the big n .

First sign: $97^{d_T} \pmod{n_T} = 97^{131} \pmod{209} = 108$

Then encrypt: $108^{e_G} \pmod{n_G} = 108^{187} \pmod{221} = 56$

T sends 56 to G, the enemy sees 56 (which is what it looks like to the enemy)

G receives 56. The receiver works with the big n and then the small n (since he's undoing).

$56^{d_G} \pmod{n_G} = 56^{115} \pmod{221} = 108$

$108^{e_T} \pmod{n_T} = 108^{191} \pmod{209} = 97$.

Now G wants to sign and encrypt the plaintext message 101 for T. In sending, work with small n then big n .

First encrypt: $101^{e_T} \pmod{n_T} = 101^{191} \pmod{209} = 112$

Then sign: $112^{d_G} \pmod{n_G} = 112^{115} \pmod{221} = 31$

G sends 31 to T, the enemy sees 31.

T receives 31. The receiver works with the big n and then the small n .

$31^{e_G} \pmod{n_G} = 31^{187} \pmod{221} = 112$

$112^{d_T} \pmod{n_T} = 112^{131} \pmod{209} = 101$.

Certificates and SSL handout

Certificates

There's still a problem not solved yet. How does Amazon know that my public key actually belongs to Ed Schaefer. Maybe Eve will completely impersonate me and say "this public key belongs to Ed Schaefer" and then purchase stuff under my name. How can Amazon be sure that the person Ed Schaefer and the public key said to belong to Ed Schaefer are actually connected? This is solved using certificates. In real life, the first time I contact Amazon, I say "this is Ed Schaefer and this is my public key". Why should Amazon believe me? There's a trusted Certification Authority (CA) who acts like an electronic notary. I can show ID to the CA and so can Amazon. The CA has a public key and digitally signs a message like "the public key n_E, e_E belongs to Ed Schaefer who works at SCU". That's my certificate. The CA also signs a message like "the public key n_A, e_A belongs to Amazon.com". That's Amazon's certificate. When I first contact Amazon, Amazon uses the CA's public key to verify the CA's signature on my certificate and I do the same for Amazon's certificate. Now, since I trust the CA, I feel confident that the Amazon is really the owner of Amazon's public key and Amazon believes that the real owner of Ed Schaefer's public key is Ed Schaefer.

Of course it's impractical for everyone to visit the CA with ID. So the CA can certify sub-CA's. For example, someone at SCU could go to the CA and show her ID and get her certificate. Then people at SCU could show ID to SCU's sub-CA. Then my certificate would be signed by the sub-CA. If I contact Amazon, then Amazon will check SCU's sub-CA's signature on my certificate. Then Amazon will check the CA's signature on the sub-CA's certificate. Similarly, Amazon might be under a sub-CA. In practice, there are different levels of certification. For a serious one, you really do show ID. You can get a less serious certificate that basically says "the e-mail address `eschaefer@hotmail.com` and the public key n_E, e_E are connected". This less serious one does not certify that the e-mail address is provably connected to person Ed Schaefer.

The world's largest and most important CA is the company Verisign which is located in Mountain View.

Secure Sockets Layer

So how does cryptography actually happen on the web? The process is called the Secure Sockets Layer (SSL). When you see `https:`, the *s* tells use that SSL is being used. There are several different protocols for doing it. The following is a simplification of how Netscape does it.

1. When Bob first connects to Amazon, Bob sends Amazon his certificate, an RSA public key $n_{\text{Bob}}, e_{\text{Bob}}$ and a public key $g^{a_{\text{Bob}}}$ to be used with the DSS signature system. The p for the DSS signature system is standard and used by everyone using Netscape. Recall DSS is a rip-off of ElGamal, so we will just use ElGamal notation. The certificate links Bob's name with his public keys.
2. Amazon uses Bob's RSA public key to encrypt an AES key and a key for a MAC. So Amazon concatenates the two keys and turns it into a number. That's easy, it could just

be the bit string considered as a base 2 number. Then Amazon raises that number to the power e_{Bob} modulo n_{Bob} . Amazon sends this to Bob.

3. Bob decrypts the message by raising it to his d_{Bob} modulo n_{Bob} and gets the AES key and the MAC key.

4. Bob then encrypts a message for Amazon. This message would include his order, his credit card number, his address, etc. This message would be encrypted with AES using the AES key sent by Amazon. It is this encrypted message that Bob sends to Amazon.

5. Bob finds the MAC of the plaintext message he just encrypted using the MAC key sent by Amazon. Bob uses DSS to sign this MAC. This looks like $S = \text{MAC}, r, x$. Bob then encrypts the signature with AES. So Bob takes $S = \text{MAC}, r$, and x and encrypts them using AES. Bob sends this encrypted, signed MAC to Amazon.

6. Amazon then receives the message from 4 and decrypts it using AES and gets the plaintext message.

7. Amazon then finds the MAC of the plaintext message using the MAC key.

8. Amazon then decrypts the signed, encrypted MAC sent by Bob using AES. Amazon then gets $S = \text{MAC}, r, x$.

9. Amazon then verifies that the MAC gotten in 7 is the same as the MAC in 8. This verifies that the message has not been tampered with.

10. Amazon then checks the signature in 8. Recall the signature is checked by ensuring that $(g^{a_{\text{Bob}}})^r \cdot r^x \pmod{p}$ and $g^S \pmod{p}$ are the same. Now Amazon knows that the signed MAC came from Bob.

Notes. Instead of using RSA, finite field Diffie Hellman and elliptic curve Diffie Hellman (now the most common of the three) could be used. For the Diffie Hellmans, Bob and Amazon would need to trade Diffie Hellman public keys at the beginning. Instead of AES, many protocols are still using triple-DES. If Bob gets cheap after he sends the order, he can not deny that he sent it. Such a denial is called repudiation. Only Bob could have signed the MAC and anyone can verify that it was Bob, because the signature only works using Bob's public key (so only Bob could have made it using his private key).

Actually, Bob could deny it if he publishes his private key. Then he could say that anyone could have faked the signature. On the other hand, if Amazon can prove that he published his private key after the order, or if Bob does not want to publish his private key, then he can't repudiate the order.

End SSL handout