ENCRYPTION THROUGH THREE-DIMENSIONAL SEPARATION AND RECOMBINATION OF DATA

NEW MEXICO ADVENTURES IN

SUPERCOMPUTING CHALLENGE

FINAL REPORT

APRIL 2, 2003

TEAM 010

ALBUQUERQUE ACADEMY

Project Members:

Preston Dell

Eric Searle

Sean Smock

Hugh Wimberly

Teacher:

Jim Mims

Project Mentor:

Jim Mims

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Our project was started with the goal of improve existing methods of encryption—as technology is progressing, it is becoming more and more important that the government and companies are able to securely transmit sensitive information over conventional electronic channels. Therefore, it is of extreme importance that these agencies have secure and reliable encryption software; this was our aim, that we would develop a method of encryption that would surpass current programs or improve cryptography by diversifying current methods.

We faced a few very important problems in attempting to satiate this need for a more secure encryption program. We needed to satisfy three basic requirements: that the program use a key to encrypt data so that it did not have a structure prone to decryption by common standards, that knowing the algorithm wouldn't compromise the integrity of the code or the encrypted data, and that the encrypted data had a unique final state after transformation and that all this could be done with relative expediency.

To accomplish all this, we wrote and developed multiple test programs in C++ after carefully researching past and current methods of encryption and their relative benefits and drawbacks. Although true benchmarking would require testing against classified decryption programs and skilled cryptologists, according to the founding mathematics we have even exceeded our goal as far as we are concerned with the likelihood of the data being decrypted.

Through these processes, we have added a new method of encryption; using an algorithm and the plain text as a whole to encrypt itself by selective translation: in other words, taking the original message and altering it, using its size to increase the number of permutations in the final cipher text.

INTRODUCTION

Encryption and Cryptography are by no means new concepts. The earliest recorded cryptographic methods come from the ancient Egyptians nearly 4000 years ago. However, until the last few hundred years, cryptography was mostly a recreational activity, used primarily in puzzles, and most types were basic letter substitution or rearrangement. In the last few centuries though, cryptography began to gain a military importance. Generals began to use encrypted messages to send information to commanders, so that intercepted messengers would not lead to opposing generals knowing accurately what future troop movements would be. During both World War I and II, encryption and decryption played a key role, and some battles were lost or won because of intercepted messages that were decrypted and messages that were not decrypted in time.

Now, however, the modern world looks more and more to the power of encryption. The internet has allowed the entire world to be connected, and allows millions of data transfers are executed on a daily basis. For some people, encryption is merely a method of privacy, a way to keep secret about what they are doing over the internet. However, the minor commercial aspects of cryptography for internet and digital media pale in comparison to the requirements of major business, banking, and government. Corporate secrets and information is passed along the digital pipeline at the speed of light, with literally millions of bits of information flowing by every second. Billions of dollars worth in business and banking transactions are completed every day, and the government uses the internet for its own purposes. In all these cases, faulty encryptions or unreliable methods can lead to catastrophic developments. Leaked corporate secrets, stolen money and classified government information are all examples of what poor encryption

techniques can do to allow cyber-terrorists to wreak havoc upon society through the virtual world, while at the same time, computing power increases at an exponential rate, out-dating encryption standards even as they come into common use. To alleviate these concerns, we decided to create and implement a new form of encryption that would bypass current methods of decryption and wouldn't be susceptible to attacks of massively parallel operation that could be preformed by quantum computers.

METHODS

The first part of the process was of course to research our problem. The most time intensive portion of our project, research was extremely difficult. At the outset of the project, we vaguely understood that this problem had never before been attempted and that what we were trying was essentially innovative. However, there were precedents: even the ENIGMA machine was integrated into our project in some sense, since it provided part of the inspiration for our character substitution algorithm. These were heavily used, especially as documentation exists detailing the relative strength and weakness of various methods of encryption. We were able to borrow what had historically been very strong, and especially in our implementation of the public-key system, programs that have never been technically defeated. (There are still some loopholes and drawbacks, which will be discussed in a later section)

In our research, first person resources were extremely useful, but exceptionally hard to find. There aren't a plethora of cryptographers ready to offer advice to the next high-school student, but we were able to contact and meet a representative from NSA (The National Security Agency, a governmental branch specializing in espionage and counter-espionage, one of the

world's leading cryptography centers) who was willing to discuss our program with us as well as discuss cryptography through history and current implementations. We are still in progress with our project and he has offered to review the applicability of our program to actual problems in cryptography, and tell us how well our program fares under standard decryption algorithms.

The second part of our scientific process entailed the mathematics. Quite a bit of our program is based upon the theoretical mathematics associated with probability, group theory and NP-complete and -incomplete problems. This allowed us to create a program that was theoretically impossible to decrypt without the key and given any limited possible computing power. Much of this was accomplished with the aid of math professors at our school who were eager to work with us and enjoyed the challenges we presented them with. Some of the mathematics were worked out prior to the development of the program, but much of it was worked out afterwards to facilitate an analysis of the efficiency and reliability of the project.

Learning the necessary coding languages was another small difficulty. All three members of our team were enrolled in Computer Science classes at the beginning of the year, so by the time we needed to start writing code for our project, we had already gained a significant understanding the syntax and skills that we needed to accomplish our project. We decided to construct our base code and the structure for our heavy computational work in C++, based on the fact that C++ is more efficient in processing raw data and extensive computations. To quickly perform the required operations, we needed a language that would not balk at performing several thousand operations per second.

PROJECT DESCRIPTION

After we had competed the necessary research and preparation, we laid the groundwork in deciding exactly what shape our program would take. We decided first to transpose multiple methods that had been used in past and present encryption algorithms. The primary element we were going to reuse was character substitution, because without character substitution, decryption boils down to unscrambling words: we decided to mimic AES, combining it with elements of also common RSA, simply because it they are current and effective methods of character substitution. The next step was to create a new method of encryption—we worked through some simple methods that worked more or less before we decided on a shuffling function, one that would use the size of the data as an aid in encryption. Again, we devised methods that would not use a ridiculously large key size that worked to a greater of lesser extent before realizing the power of a Rubik's cube: each individual block on the cube would represent a portion of the original data, and by twisting and turning the cube we would shuffle the data portions. This was just a beginning idea, but our concept grew and developed from this seed.

We decided simultaneously to borrow from various public-key cryptography systems, and throughout our project we have worked out multiple ways of generating a key through this system that would allow a non-discrete number of transformations on our data set that would not be in any set or recognizable pattern. This provided some of the greatest challenges for us, especially finding appropriate equations and functions that would suit our purposes.

The actual implementation of the project took quite a while, and is still on ongoing process, but was one of the easier parts to do. We are all concurrently taking an advanced C++ course, and so we didn't need to learn very much more to write the code for the program. Getting

the computer to act in accordance with our desires is always a difficult process when the desired operations are sufficiently complex, and this was tiring especially in the numerous debugging stages, but we were able to apply the computer sufficiently rigorously.

The actual transformation from the plain text to cipher text goes as follows:

1) The plain text, in a static data stream, is fed into a three dimensional array with equal height, width and depth.
2) Along each axis of the array, or along each dimension, the elements of one layer (from the inside out) are rotated to some degree. After this has been done several times, the array still contains all of the original elements but they are shuffled, much like a twisted Rubik's Cube. This is the transformation portion of the program.
3) The elements are fed one at a time into another static data stream, where they are altered through character substitution according the proper key, in a manner similar to existing methods. This is the transmutation section of the program.

However, this is not sufficient in the greater scope of the program. Before this is accomplished, a key must be generated that is sufficiently secure that the encryption isn't rendered useless. The data is encrypted as above, then transmitted, and decrypted by reversing the encryption above according to the private key.

Our first slightly successful version of the code ran an array of a mere 64 elements, allowing only an outer layer horizontal-clockwise rotation on the four layers, but did manage to prove the ability to the program to shuffle data even given only a few transformations. However, we immediately moved on to more complete programs.

Our second program used a 100x100x100 array, which allowed us to store nearly a

megabyte of information in an encoded format. However, this also only rotated the outer horizontal layers clockwise; we continued to modify it to fit our needs. We soon changed the code so that it would rotate along all three axes, and eventually so that it employed a recursive pattern that allowed it to rotate inner layers in a step-wise fashion.

However, to that point the rotation was determined by a random number or a user generated sequence. Both these were infeasible, given our project goals. Both assumed a symmetric-key encryption with a several-hundred digit number, and didn't allow a character substitution. We changed our focus at this point and started developing key-transmission code, based on the AES open-source code and our own modifications. Additionally, we started down a new path, using the key that the public-key system generated to generate a unique sequence of moves along all axes as well as providing a method for character substitution at the end. For this we used a non-recursive, non-repetitive bounded function with a domain of all the bounded numbers to generate all the integers within the same bounds, in the case of the 100-sided array, zero to three-hundred ninety-six. Combining the programs has not yet been done; as of yet, this is still one more step, although this it will be a trivial matter once we are ready to fully integrate them.

MATH MODEL

Our project requires some relatively advanced mathematics to analyze, but the basis is quite simple. First of all, the transmutations in the first part of our project are simply re-organization of our data elements. Essentially, the data is rotated inside the array to the point that each key can create a unique combination. Since data elements can be anywhere in a given layer (there is no cross-layer manipulation), the function for the number of permutations is P, where P is determined as follows: P = 1 * ([4-1]^3 - [4-3]^3)! * ([6-1]^3 - [6-3]^3)! * . . . ([2r-1]^3 - [2r-3]^3)! ,where r is the number of rows. Notice that for a 100x100x100 array this number greatly exceeds 1.503e+1,192,594 (that is from n = 46 to n =51) Note that this does indeed mean that a 5x5x5 array has more possible permutations than the estimated number of electrons in the universe. (P = 1 * 26! * 99!)

Obviously, this is more than the number of keys, so not all of these combinations can be realized. The difficulty has been reduced exactly to the key size, where the cipher text can only be decrypted by analyzing every single key individually and looking at the resulting probably plain text to see if it was the correct key. For a 100 digit key, this is essentially an impossible task, so we have achieved a sufficient level of difficulty. Again note that this is only true with a reliable key substitution, since a character frequency recognition algorithm could theoretically decrypt a message of sufficient length into a scrambled message; also, without a character substitution the message is scrambled but no more. With a long enough message unscrambling becomes difficult, but especially with a short message, a human can with enough work unscramble character jumbles into words.

ANALYSIS

Finding a method to uniquely encrypt a data set was ridiculously easy, but finding one that was not prone to backtracking and other methods of decryption, especially those that implement hashing and multiple parallel operations (quantum computers, in specific) was excessively difficult. Furthermore, there might have been possibilities that have eluded us, and decryption techniques that are classified and that only the NSA and other secret organizations have access to. All of these mean that there is a high likelihood that there is a better way than brute-force reversal of the algorithm to decrypt encoded messages. However, the brute-force method is so difficult that even if a better algorithm could reduce the difficulty of decoding exponentially by powers of ten to the hundreds, there would still be no way of conceivably decoding the message in any reasonable amount of time (e.g., billions of years, at minimum, with current computing power) so we feel relatively secure that our code is safe to these kinds of attacks.

For our code, we used the common AES (advanced encryption standard) key-generation system with slight modification. Although this cannot be mathematically broached, the code transmission is (and we would like to point out that there is no code that has open transmission that is not subject to this failure) vulnerable to attacks that impersonate the sender and receiver, a deception algorithm or crack, that allows the open keys to be decoded and recoded, resent to the original receiver. However, these attacks are extremely difficult and would have to take place in the space of a single second to work. The only other possibility would be to implement a symmetric-key encryption system. Although our code currently is easily adaptable to any symmetric-key system, we do not desire this since it precludes any easily accessible online

transaction, which is the primary focus of our project. The last and final objection to the AES public-key system is that it is the only part of our program subject to quantum attacks; in other words, a quantum computer could conceivably quickly decrypt the private key and use that to decrypt the data sent. In the end, we will be forced to use a better key-encryption system if we want to exceed the capabilities of current encryption standards, simply because this fault is the weakest link in our chain, and by breaking it, a hacker renders useless all the other defenses of our program.

Our project contains the problem that it currently cannot send blocks of data larger than a megabyte. Although this does not prohibit most text for example, any pictures or digital media are difficult to send. They would have to be broken into megabyte sections and sent individually, either in the same transmission with the same encryption key (which conceivably could allow decryption in less than a year) or in separate transmissions that all individually generate a key to follow. The first approach is fine when the data is time sensitive but no longer will need to be secure in the space of a year (although using the same key does not necessarily decrease the decryption time) and the second approach is better when fast transactions are not needed, although it would take a little less than twice the time of the first method, of course assuming the same transmission speeds. The main problem with these approaches is not security but the fact that the programs or data beings sent would have to be separated by some other program since ours is not equipped to do this. If we deem this to be necessary, we could include a separate program in our final software that would allow this split of data.

Other Applications

Our encryption program has other possible uses that could be as conceivably important. By using eight simultaneous 100-depth arrays filled each block with a separate binary code, we could send streaming data by sending only the coordinates of the proper binary stream. This code would be actually be more secure than our original program. Although we could not implement a character substitution in this instance, it would be unnecessary. The only attack that could decrypt the algorithm would be one that dealt with the initial public-key transmission. The primary strength of this would be that it can send data in real-time and has no limit to how much data can be sent.

CONCLUSION

For this project, we chose an extremely difficult problem to tackle. It was ambitious if us, and we may be proven incorrect, but based on mathematical evidence, we believe that our program is stronger than currently existing encryption standards. We have not thoroughly tested its application, which will be difficult to even begin, although trivial to finish. This is the final step of our program, and we hope to accomplish it within the next month. The only other thing we need to consider is the vulnerability of the current key and whether we should attempt to design a better key encryption system, which has been a daunting problem to even the best computer scientists and cryptologists in the world.

ACKNOWLEDGMENTS

First, our thanks to Jim Mims, our computer science instructor and sponsor, for suggesting the project originally as well as working with us on programming aspects.

Further thanks to Dr. Authur Payne and Dr. Agustin Kintanar, for directing us towards useful sources and helping us with necessary derivations of equations we used in physics.

Thanks to Don Smith for helping us with the necessary math and granting us access to various calculus textbooks dealing with probability, group theory and NP-completeness and -incompleteness.

We heavily used existing knowledge of encryption algorithms in use, using the existing Rijndael AES open-source code in C.

Much of the knowledge that we gained came from Frederic Rasio, Ph.D., Dept. of Cryptography, MIT and his paper key generation systems.

The online publications by MIT, CalTech and University of Michigan of the doctorate work of their graduate students was extremely helpful in navigating our topic.

References

1. Decrypted Secrets: Methods and Maxims of Cryptology, Third Edition, Bauer

2. Applied Cryptography, Bruce Schneir, Wiley.3. Lucy, L.B. 1997, AJ, 82, 1013

3. A.J. Menezes, P.C. van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

4. Writing Secure Code, Michael Howard and David LeBlanc, Microsoft Corporation.

5. Web Hacking: Attacks and Defense, Stuard McClure, Addision Wesley.

6. Cryptography for Internet and Database Applications, Nick Galbreath, Wiley.

7. E. Biham and A. Shamir, Differential Cryptoanalysis of the Data Encryption Standard, Springer-Verlag, 1993.

8. B. Schneir, Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish), Cambridge Security Workshop Proceedings, Springer-Verlag, 1994.

9. A.J. Menezes, P.C. van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.

10. D. Stachel, I. Svoboda, and H. Feuss, Acta Crystallogr., Sect. C, 1995, 51, 1049

11. AES Algorithm by Rijndael
http://csrc.nist.gov/encryption/aes/rijndael

12. RSA Security: Some Papers
http://www.rsasecurity.com/rsalabs/technotes

13. David Kahn, The Codebreakers, The Macmillan Company, 1967.

The following is a brief example of a large array, using user inputs for simplicity. This program differs only in that it doesn't use a bounded function to determine rotation (instead of user input) and that it only shuffles (rather than apply character substitution to) the data.

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>


//====================================
//   Global Variables
//====================================

//Initialize 4 x 4 x 4 matrix
// 4x     int matrix[4][4][4];
int matrix[100][100][100];

//Initialize 64 character array for INITIAL INPUT
char input1[1000000];

//Initialize 68 character array for ENCODED OUTPUT (including 4-digit code)
char output1[1000004];

//Initialize 4 character array for CODE
int code[4];

ofstream outfile;



//====================================
//   Function definitions
//====================================

//Initialize SWAP function
void swap(int &, int &);

//Initialize INITIAL INPUT function
void initinput();

//Initialize POPULATE 3D MATRIX function
void poparray();
```

```
//Initialize PRINT MATRIX function
void printarray();

//Initialize RECEIVE CODE function
void applyrubix();

//Initialize RUBIX CUBE ENCODING function
void rubix(int level, int times);

//Initialize ENCODED OUTPUT function
void popoutput();

//Initialize PRINT INITIAL INPUT/ENCODED OUTPUT function
void printfinals();



//====================================
//   Main function
//====================================

int main()
{
	outfile.open ("h:\\sample.txt", ios::out);

	initinput();	//Receive INITIAL INPUT (2D)
	poparray();		//Populate 3D MATRIX

	system("cls");

	//printarray();  //Print 3D MATRIX on screen
	applyrubix();   //Receive CODE and apply RUBIX CUBE ENCODING

	system("cls");

	//printarray();  //Print 3D MATRIX on screen

	system("pause");
	system("cls");

	popoutput();    //Populate ENCODED OUTPUT (2D)
	printfinals();  //Print INITIAL INPUT and ENCODED OUTPUT on screen

	return 0;
```

```
}


//====================================
//  Function to SWAP two integers
//====================================

void swap(int &x, int &y)       //Figure it out
{
        int a = 0;
        a = x;
        x = y;
        y = a;
}


//====================================
//  Function to receive INITIAL INPUT
//====================================

void initinput()
{
        for (int i = 0; i < 1000000; i++)
        {
                system("cls");
                cout << "Characters remaining: " << 1000000 - i << endl;
                for (int b = 0; b < i; b++)
                        cout << input1[b];
                cin >> input1[i];
                if (input1[i] == 126)
                        i = 1000000;

        }
}


//====================================
//   Function to populate 3D matrix with INITIAL INPUT
//   (2D - 3D object)
//====================================

void poparray()
{
        int a = 0;
```

```
	for (int i=0;i<100;i++)
	{
		for (int j=0;j<100;j++)
		{
			for (int k=0;k<100;k++)
			{
				matrix[i][j][k] = input1[a];
				a = a+1;
			}
		}
	}
}


//====================================
//  Function to print the 3D matrix
//====================================

void printarray()
{
	for (int i=0;i<100;i++)
	{
		for (int j=0;j<100;j++)
		{
			for (int k=0;k<100;k++)
			{
				cout << matrix[i][j][k] << "  ";

			}
			cout << endl;
		}
		cout << endl << "--------" << endl;
	}
}


//====================================
//  Function to RECIEVE CODE
//====================================

void applyrubix()
{
	int times;
	int level;
```

```
	for (int a = 0; a < 100; a++)
	{
		level = a;
			//Set level
		cout << "How many times on level " << a << "?" << endl;
			//Ask for code digit according to level
		cin >> times;
		//Input code digit according to level
		rubix(level,times);
		//Apply RUBIX CUBE ENCODING according to the level and how many times
		code[a] = times;
		//Put code digit in appropriate array place holder
	}
	cout << endl << "=======" << endl;
}


//====================================
//  Function to apply RUBIX CUBE encoding
//  (3D encoding)
//====================================

void rubix(int level, int times)
{
	for (int d = 0 ; d<times ; d++)
	{
		for (int a = 0; a<99;a++)
		{
			swap(matrix[level][0][a],matrix[level][0][a+1]);
		}
		for (int b = 0; b<99;b++)
		{
			swap(matrix[level][b][99],matrix[level][b+1][99]);
		}
		for (int c = 99; c>0;c--)
		{
			swap(matrix[level][99][c],matrix[level][99][c-1]);
		}
		for (int d = 99; d>1;d--)
			swap(matrix[level][d][0],matrix[level][d-1][0]);
	}
}
```

```
//====================================
//  Function to populate 2D array with ENCODED OUTPUT
//  (3D - 2D object)
//====================================

void popoutput()
{
	int a = 0;
	for (int i=0;i<100;i++)
	{
		for (int j=0;j<100;j++)
		{
			for (int k=0;k<100;k++)
			{

				output1[a] = matrix[i][j][k];
					//Populate ENCODED OUTPUT array
				a = a+1;
			}
		}
	}
}


//====================================
//  Function to print INITIAL INPUT and ENCODED OUTPUT
//====================================

void printfinals()
{
	/* NEW
	cout << "Original input: ";
		for (int k = 0; k < 1000000; k++)

			cout << input1[k];
		// OLD  for (int aa=0;aa<64;aa++)
		// OLD outfile << input1[aa];


	cout << endl << "Encoded output: ";
		for (int j = 0; j < 1000000; j++)
			cout << output1[j];
		for (int l = 0; l < 4; l++)
		cout << endl;*/
```

```
        // OLD  outfile << endl;
                for (int bb=0; bb<1000000;bb++)
                        outfile << output1[bb];
                outfile << endl << endl <<
"**************************************************" << endl;


}
```

Here is the corresponding example code also using a user input that decrypts the encrypted (shuffled) data stream:

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>

//Globals
int matrix[100][100][100];
char input1[1000000];
int code[4];
char output1[1000000];



//Functions
void poparray();
void printarray();
void swap(int &, int &);
void antirubix(int level, int times);
void popoutput();


int main()
{

        ifstream  infile;
   char    letter;
   infile.open
   ("h:\\sample.txt",ios::in);
        int ii = 0;
   while (infile >> letter)
        {
                input1[ii] = letter;
```

```
		ii = ii + 1;
}


int times;
int level;

/*for (int i = 0; i < 64; i++)
{
		system("cls");
		cout << "Characters remaining: " << 64 - i << endl;
		for (int b = 0; b < i; b++)
			cout << input1[b];
		cin >> input1[i];
}
cout << endl;*/
cout << "Input code:";
for (int q=0;q<100;q++)
		cin >> code[q];


system("pause");


poparray();
system("cls");
printarray();
for (int a = 0; a < 100; a++)
{
		level = a;
		//cout << "How many times on level " << a << "?" << endl;
		//cin >> times;
		times = 396 - code[a];
		antirubix(level,times);
		//code[a] = times;
}
cout << endl << "=======" << endl;
system("pause");
system("cls");
printarray();
system("pause");
system("cls");
popoutput();
cout << "Encoded input: ";
```

```
	for (int k = 0; k < 1000000; k++)
		cout << input1[k];
	cout << endl << "Decoded output: ";

	for (int j = 0; j < 1000000; j++)
		cout << output1[j];
	cout << endl;
	return 0;



}
void poparray()
{
	int a = 0;
	for (int i=0 ; i<100 ; i++)
	{
		for (int j=0 ; j<100 ; j++)
		{
			for (int k=0 ; k<100 ; k++)
			{
				matrix[i][j][k] = input1[a];
				a = a+1;
			}
		}
	}
}
void popoutput()
{
	int a = 0;
	for (int i=0;i<100;i++)
	{
		for (int j=0;j<100;j++)
		{
			for (int k=0;k<100;k++)
			{
				output1[a] = matrix[i][j][k];
				a = a+1;
			}
		}
	}
}

void printarray()
```

```
{
	for (int i=0;i<100;i++)
	{
		for (int j=0;j<100;j++)
		{
			for (int k=0;k<100;k++)
			{
				cout << matrix[i][j][k] << "  ";

			}
			cout << endl;
		}
		cout << endl << "--------" << endl;
	}
}
void swap(int &x, int &y)
{
	int a = 0;
	a = x;
	x = y;
	y = a;
}
void antirubix(int level, int times)
{
	for (int d = 0;d<times;d++)
	{
		for (int a = 0; a<99;a++)
		{
			swap(matrix[level][0][a],matrix[level][0][a+1]);
		}
		for (int b = 0; b<99;b++)
		{
			swap(matrix[level][b][99],matrix[level][b+1][99]);
		}
		for (int c = 99; c>0;c--)
		{
			swap(matrix[level][99][c],matrix[level][99][c-1]);
		}
		for (int d = 99; d>1;d--)
			swap(matrix[level][d][0],matrix[level][d-1][0]);
	}
}
```

The following is example code for the public-key generation system, uncommented (the size has

been reduced to avoid formatting errors):

```
#include "aesopt.h"

#if defined(BLOCK_SIZE) && (BLOCK_SIZE & 7)
#error An illegal block size has been specified.
#endif

/* Subroutine to set the block size (if variable) in bytes, legal
   values being 16, 24 and 32.
*/

#if !defined(BLOCK_SIZE)

aes_rval aes_blk_len(unsigned int blen, aes_ctx cx[1])
{
#if !defined(FIXED_TABLES)
    if(!tab_init) gen_tabs();
#endif

    if((blen & 7) || blen < 16 || blen > 32)
    {
        cx->n_blk = 0; return aes_bad;
    }

    cx->n_blk = blen;
    return aes_good;
}

#endif

/* Initialise the key schedule from the user supplied key. The key
   length is now specified in bytes - 16, 24 or 32 as appropriate.
   This corresponds to bit lengths of 128, 192 and 256 bits, and
   to Nk values of 4, 6 and 8 respectively.

   The following macros implement a single cycle in the key
   schedule generation process. The number of cycles needed
   for each cx->n_col and nk value is:

    nk =            4  5  6  7  8
    ------------------------------
    cx->n_col = 4   10  9  8  7  7
    cx->n_col = 5   14 11 10  9  9
    cx->n_col = 6   19 15 12 11 11
    cx->n_col = 7   21 19 16 13 14
    cx->n_col = 8   29 23 19 17 14
*/

#define ke4(k,i) \
{   k[4*(i)+4] = ss[0] ^= ls_box(ss[3],3) ^ rcon_tab[i]; k[4*(i)+5] = ss[1] ^= ss[0]; \
    k[4*(i)+6] = ss[2] ^= ss[1]; k[4*(i)+7] = ss[3] ^= ss[2]; \
```

```
}
#define kel4(k,i) \
{   k[4*(i)+4] = ss[0] ^= ls_box(ss[3],3) ^ rcon_tab[i]; k[4*(i)+5] = ss[1] ^= ss[0]; \
    k[4*(i)+6] = ss[2] ^= ss[1]; k[4*(i)+7] = ss[3] ^= ss[2]; \
}

#define ke6(k,i) \
{   k[6*(i)+ 6] = ss[0] ^= ls_box(ss[5],3) ^ rcon_tab[i]; k[6*(i)+ 7] = ss[1] ^= ss[0]; \
    k[6*(i)+ 8] = ss[2] ^= ss[1]; k[6*(i)+ 9] = ss[3] ^= ss[2]; \
    k[6*(i)+10] = ss[4] ^= ss[3]; k[6*(i)+11] = ss[5] ^= ss[4]; \
}
#define kel6(k,i) \
{   k[6*(i)+ 6] = ss[0] ^= ls_box(ss[5],3) ^ rcon_tab[i]; k[6*(i)+ 7] = ss[1] ^= ss[0]; \
    k[6*(i)+ 8] = ss[2] ^= ss[1]; k[6*(i)+ 9] = ss[3] ^= ss[2]; \
}

#define ke8(k,i) \
{   k[8*(i)+ 8] = ss[0] ^= ls_box(ss[7],3) ^ rcon_tab[i]; k[8*(i)+ 9] = ss[1] ^= ss[0]; \
    k[8*(i)+10] = ss[2] ^= ss[1]; k[8*(i)+11] = ss[3] ^= ss[2]; \
    k[8*(i)+12] = ss[4] ^= ls_box(ss[3],0); k[8*(i)+13] = ss[5] ^= ss[4]; \
    k[8*(i)+14] = ss[6] ^= ss[5]; k[8*(i)+15] = ss[7] ^= ss[6]; \
}
#define kel8(k,i) \
{   k[8*(i)+ 8] = ss[0] ^= ls_box(ss[7],3) ^ rcon_tab[i]; k[8*(i)+ 9] = ss[1] ^= ss[0]; \
    k[8*(i)+10] = ss[2] ^= ss[1]; k[8*(i)+11] = ss[3] ^= ss[2]; \
}

#if defined(ENCRYPTION_KEY_SCHEDULE)

aes_rval aes_enc_key(const unsigned char in_key[], unsigned int klen, aes_ctx cx[1])
{   aes_32t    ss[8];

#if !defined(FIXED_TABLES)
    if(!tab_init) gen_tabs();
#endif

#if !defined(BLOCK_SIZE)
    if(!cx->n_blk) cx->n_blk = 16;
#else
    cx->n_blk = BLOCK_SIZE;
#endif

    cx->n_blk = (cx->n_blk & ~3) | 1;

    cx->k_sch[0] = ss[0] = word_in(in_key     );
    cx->k_sch[1] = ss[1] = word_in(in_key +  4);
    cx->k_sch[2] = ss[2] = word_in(in_key +  8);
    cx->k_sch[3] = ss[3] = word_in(in_key + 12);

#if (BLOCK_SIZE == 16) && (ENC_UNROLL != NONE)

    switch(klen)
```

```
    {
    case 16:    ke4(cx->k_sch, 0); ke4(cx->k_sch, 1);
                ke4(cx->k_sch, 2); ke4(cx->k_sch, 3);
                ke4(cx->k_sch, 4); ke4(cx->k_sch, 5);
                ke4(cx->k_sch, 6); ke4(cx->k_sch, 7);
                ke4(cx->k_sch, 8); kel4(cx->k_sch, 9);
                cx->n_rnd = 10; break;
    case 24:    cx->k_sch[4] = ss[4] = word_in(in_key + 16);
                cx->k_sch[5] = ss[5] = word_in(in_key + 20);
                ke6(cx->k_sch, 0); ke6(cx->k_sch, 1);
                ke6(cx->k_sch, 2); ke6(cx->k_sch, 3);
                ke6(cx->k_sch, 4); ke6(cx->k_sch, 5);
                ke6(cx->k_sch, 6); kel6(cx->k_sch, 7);
                cx->n_rnd = 12; break;
    case 32:    cx->k_sch[4] = ss[4] = word_in(in_key + 16);
                cx->k_sch[5] = ss[5] = word_in(in_key + 20);
                cx->k_sch[6] = ss[6] = word_in(in_key + 24);
                cx->k_sch[7] = ss[7] = word_in(in_key + 28);
                ke8(cx->k_sch, 0); ke8(cx->k_sch, 1);
                ke8(cx->k_sch, 2); ke8(cx->k_sch, 3);
                ke8(cx->k_sch, 4); ke8(cx->k_sch, 5);
                kel8(cx->k_sch, 6);
                cx->n_rnd = 14; break;
    default:    cx->n_rnd = 0; return aes_bad;
    }
#else
    {   aes_32t i, l;
        cx->n_rnd = ((klen >> 2) > nc ? (klen >> 2) : nc) + 6;
        l = (nc * cx->n_rnd + nc - 1) / (klen >> 2);

        switch(klen)
        {
        case 16:    for(i = 0; i < l; ++i)
                        ke4(cx->k_sch, i);
                    break;
        case 24:    cx->k_sch[4] = ss[4] = word_in(in_key + 16);
                    cx->k_sch[5] = ss[5] = word_in(in_key + 20);
                    for(i = 0; i < l; ++i)
                        ke6(cx->k_sch, i);
                    break;
        case 32:    cx->k_sch[4] = ss[4] = word_in(in_key + 16);
                    cx->k_sch[5] = ss[5] = word_in(in_key + 20);
                    cx->k_sch[6] = ss[6] = word_in(in_key + 24);
                    cx->k_sch[7] = ss[7] = word_in(in_key + 28);
                    for(i = 0; i < l; ++i)
                        ke8(cx->k_sch,  i);
                    break;
        default:    cx->n_rnd = 0; return aes_bad;
        }
    }
#endif
```

```
    return aes_good;
}

#endif

#if defined(DECRYPTION_KEY_SCHEDULE)

#if (DEC_ROUND != NO_TABLES)
#define d_vars  dec_imvars
#define ff(x)   inv_mcol(x)
#else
#define ff(x)   (x)
#define d_vars
#endif

#if 1
#define kdf4(k,i) \
{   ss[0] = ss[0] ^ ss[2] ^ ss[1] ^ ss[3]; ss[1] = ss[1] ^ ss[3]; ss[2] = ss[2] ^ ss[3]; ss[3] = ss[3]; \
    ss[4] = ls_box(ss[(i+3) % 4], 3) ^ rcon_tab[i]; ss[i % 4] ^= ss[4]; \
    ss[4] ^= k[4*(i)];   k[4*(i)+4] = ff(ss[4]); ss[4] ^= k[4*(i)+1]; k[4*(i)+5] = ff(ss[4]); \
    ss[4] ^= k[4*(i)+2]; k[4*(i)+6] = ff(ss[4]); ss[4] ^= k[4*(i)+3]; k[4*(i)+7] = ff(ss[4]); \
}
#define kd4(k,i) \
{   ss[4] = ls_box(ss[(i+3) % 4], 3) ^ rcon_tab[i]; ss[i % 4] ^= ss[4]; ss[4] = ff(ss[4]); \
    k[4*(i)+4] = ss[4] ^= k[4*(i)]; k[4*(i)+5] = ss[4] ^= k[4*(i)+1]; \
    k[4*(i)+6] = ss[4] ^= k[4*(i)+2]; k[4*(i)+7] = ss[4] ^= k[4*(i)+3]; \
}
#define kdl4(k,i) \
{   ss[4] = ls_box(ss[(i+3) % 4], 3) ^ rcon_tab[i]; ss[i % 4] ^= ss[4]; \
    k[4*(i)+4] = (ss[0] ^= ss[1]) ^ ss[2] ^ ss[3]; k[4*(i)+5] = ss[1] ^ ss[3]; \
    k[4*(i)+6] = ss[0]; k[4*(i)+7] = ss[1]; \
}
#else
#define kdf4(k,i) \
{   ss[0] ^= ls_box(ss[3],3) ^ rcon_tab[i]; k[4*(i)+ 4] = ff(ss[0]); ss[1] ^= ss[0]; k[4*(i)+ 5] = ff(ss[1]); \
    ss[2] ^= ss[1]; k[4*(i)+ 6] = ff(ss[2]); ss[3] ^= ss[2]; k[4*(i)+ 7] = ff(ss[3]); \
}
#define kd4(k,i) \
{   ss[4] = ls_box(ss[3],3) ^ rcon_tab[i]; \
    ss[0] ^= ss[4]; ss[4] = ff(ss[4]); k[4*(i)+ 4] = ss[4] ^= k[4*(i)]; \
    ss[1] ^= ss[0]; k[4*(i)+ 5] = ss[4] ^= k[4*(i)+ 1]; \
    ss[2] ^= ss[1]; k[4*(i)+ 6] = ss[4] ^= k[4*(i)+ 2]; \
    ss[3] ^= ss[2]; k[4*(i)+ 7] = ss[4] ^= k[4*(i)+ 3]; \
}
#define kdl4(k,i) \
{   ss[0] ^= ls_box(ss[3],3) ^ rcon_tab[i]; k[4*(i)+ 4] = ss[0]; ss[1] ^= ss[0]; k[4*(i)+ 5] = ss[1]; \
    ss[2] ^= ss[1]; k[4*(i)+ 6] = ss[2]; ss[3] ^= ss[2]; k[4*(i)+ 7] = ss[3]; \
}
#endif

#define kdf6(k,i) \
{   ss[0] ^= ls_box(ss[5],3) ^ rcon_tab[i]; k[6*(i)+ 6] = ff(ss[0]); ss[1] ^= ss[0]; k[6*(i)+ 7] = ff(ss[1]); \
```

```
    ss[2] ^= ss[1]; k[6*(i)+ 8] = ff(ss[2]); ss[3] ^= ss[2]; k[6*(i)+ 9] = ff(ss[3]); \
    ss[4] ^= ss[3]; k[6*(i)+10] = ff(ss[4]); ss[5] ^= ss[4]; k[6*(i)+11] = ff(ss[5]); \
}
#define kd6(k,i) \
{   ss[6] = ls_box(ss[5],3) ^ rcon_tab[i]; \
    ss[0] ^= ss[6]; ss[6] = ff(ss[6]); k[6*(i)+ 6] = ss[6] ^= k[6*(i)]; \
    ss[1] ^= ss[0]; k[6*(i)+ 7] = ss[6] ^= k[6*(i)+ 1]; \
    ss[2] ^= ss[1]; k[6*(i)+ 8] = ss[6] ^= k[6*(i)+ 2]; \
    ss[3] ^= ss[2]; k[6*(i)+ 9] = ss[6] ^= k[6*(i)+ 3]; \
    ss[4] ^= ss[3]; k[6*(i)+10] = ss[6] ^= k[6*(i)+ 4]; \
    ss[5] ^= ss[4]; k[6*(i)+11] = ss[6] ^= k[6*(i)+ 5]; \
}
#define kdl6(k,i) \
{   ss[0] ^= ls_box(ss[5],3) ^ rcon_tab[i]; k[6*(i)+ 6] = ss[0]; ss[1] ^= ss[0]; k[6*(i)+ 7] = ss[1]; \
    ss[2] ^= ss[1]; k[6*(i)+ 8] = ss[2]; ss[3] ^= ss[2]; k[6*(i)+ 9] = ss[3]; \
}

#define kdf8(k,i) \
{   ss[0] ^= ls_box(ss[7],3) ^ rcon_tab[i]; k[8*(i)+ 8] = ff(ss[0]); ss[1] ^= ss[0]; k[8*(i)+ 9] = ff(ss[1]); \
    ss[2] ^= ss[1]; k[8*(i)+10] = ff(ss[2]); ss[3] ^= ss[2]; k[8*(i)+11] = ff(ss[3]); \
    ss[4] ^= ls_box(ss[3],0); k[8*(i)+12] = ff(ss[4]); ss[5] ^= ss[4]; k[8*(i)+13] = ff(ss[5]); \
    ss[6] ^= ss[5]; k[8*(i)+14] = ff(ss[6]); ss[7] ^= ss[6]; k[8*(i)+15] = ff(ss[7]); \
}
#define kd8(k,i) \
{   aes_32t g = ls_box(ss[7],3) ^ rcon_tab[i]; \
    ss[0] ^= g; g = ff(g); k[8*(i)+ 8] = g ^= k[8*(i)]; \
    ss[1] ^= ss[0]; k[8*(i)+ 9] = g ^= k[8*(i)+ 1]; \
    ss[2] ^= ss[1]; k[8*(i)+10] = g ^= k[8*(i)+ 2]; \
    ss[3] ^= ss[2]; k[8*(i)+11] = g ^= k[8*(i)+ 3]; \
    g = ls_box(ss[3],0); \
    ss[4] ^= g; g = ff(g); k[8*(i)+12] = g ^= k[8*(i)+ 4]; \
    ss[5] ^= ss[4]; k[8*(i)+13] = g ^= k[8*(i)+ 5]; \
    ss[6] ^= ss[5]; k[8*(i)+14] = g ^= k[8*(i)+ 6]; \
    ss[7] ^= ss[6]; k[8*(i)+15] = g ^= k[8*(i)+ 7]; \
}
#define kdl8(k,i) \
{   ss[0] ^= ls_box(ss[7],3) ^ rcon_tab[i]; k[8*(i)+ 8] = ss[0]; ss[1] ^= ss[0]; k[8*(i)+ 9] = ss[1]; \
    ss[2] ^= ss[1]; k[8*(i)+10] = ss[2]; ss[3] ^= ss[2]; k[8*(i)+11] = ss[3]; \
}

aes_rval aes_dec_key(const unsigned char in_key[], unsigned int klen, aes_ctx cx[1])
{   aes_32t    ss[8];
    d_vars

#if !defined(FIXED_TABLES)
    if(!tab_init) gen_tabs();
#endif

#if !defined(BLOCK_SIZE)
    if(!cx->n_blk) cx->n_blk = 16;
#else
    cx->n_blk = BLOCK_SIZE;
```

```
#endif

    cx->n_blk = (cx->n_blk & ~3) | 2;

    cx->k_sch[0] = ss[0] = word_in(in_key     );
    cx->k_sch[1] = ss[1] = word_in(in_key +  4);
    cx->k_sch[2] = ss[2] = word_in(in_key +  8);
    cx->k_sch[3] = ss[3] = word_in(in_key + 12);

#if (BLOCK_SIZE == 16) && (DEC_UNROLL != NONE)

    switch(klen)
    {
    case 16:    kdf4(cx->k_sch, 0); kd4(cx->k_sch, 1);
                kd4(cx->k_sch, 2); kd4(cx->k_sch, 3);
                kd4(cx->k_sch, 4); kd4(cx->k_sch, 5);
                kd4(cx->k_sch, 6); kd4(cx->k_sch, 7);
                kd4(cx->k_sch, 8); kdl4(cx->k_sch, 9);
                cx->n_rnd = 10; break;
    case 24:    cx->k_sch[4] = ff(ss[4] = word_in(in_key + 16));
                cx->k_sch[5] = ff(ss[5] = word_in(in_key + 20));
                kdf6(cx->k_sch, 0); kd6(cx->k_sch, 1);
                kd6(cx->k_sch, 2); kd6(cx->k_sch, 3);
                kd6(cx->k_sch, 4); kd6(cx->k_sch, 5);
                kd6(cx->k_sch, 6); kdl6(cx->k_sch, 7);
                cx->n_rnd = 12; break;
    case 32:    cx->k_sch[4] = ff(ss[4] = word_in(in_key + 16));
                cx->k_sch[5] = ff(ss[5] = word_in(in_key + 20));
                cx->k_sch[6] = ff(ss[6] = word_in(in_key + 24));
                cx->k_sch[7] = ff(ss[7] = word_in(in_key + 28));
                kdf8(cx->k_sch, 0); kd8(cx->k_sch, 1);
                kd8(cx->k_sch, 2); kd8(cx->k_sch, 3);
                kd8(cx->k_sch, 4); kd8(cx->k_sch, 5);
                kdl8(cx->k_sch, 6);
                cx->n_rnd = 14; break;
    default:    cx->n_rnd = 0; return aes_bad;
    }
#else
    {   aes_32t i, l;
        cx->n_rnd = ((klen >> 2) > nc ? (klen >> 2) : nc) + 6;
        l = (nc * cx->n_rnd + nc - 1) / (klen >> 2);

        switch(klen)
        {
        case 16:
                for(i = 0; i < l; ++i)
                    ke4(cx->k_sch, i);
                break;
        case 24:    cx->k_sch[4] = ss[4] = word_in(in_key + 16);
                cx->k_sch[5] = ss[5] = word_in(in_key + 20);
                for(i = 0; i < l; ++i)
                    ke6(cx->k_sch, i);
```

```
            break;
        case 32:    cx->k_sch[4] = ss[4] = word_in(in_key + 16);
            cx->k_sch[5] = ss[5] = word_in(in_key + 20);
            cx->k_sch[6] = ss[6] = word_in(in_key + 24);
            cx->k_sch[7] = ss[7] = word_in(in_key + 28);
            for(i = 0; i < l; ++i)
               ke8(cx->k_sch,  i);
            break;
        default:    cx->n_rnd = 0; return aes_bad;
        }
#if (DEC_ROUND != NO_TABLES)
        for(i = nc; i < nc * cx->n_rnd; ++i)
           cx->k_sch[i] = inv_mcol(cx->k_sch[i]);
#endif
    }
#endif

    return aes_good;
}

#endif
```

And a similar example of a character substitution and encryption, using a single small length key without any hash sequence to ensure data integrity:

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

#include "aes.h"

#define BLOCK_LEN   16
#define READ_ERROR  -7
#define WRITE_ERROR -8

#ifdef LINUX
#define file_len(x) (unsigned long)x.__pos
#else
#define file_len(x) (unsigned long)x
#endif

//  A Pseudo Random Number Generator (PRNG) used for the
//  Initialisation Vector. The PRNG is George Marsaglia's
//  Multiply-With-Carry (MWC) PRNG that concatenates two
//  16-bit MWC generators:
//      x(n)=36969 * x(n-1) + carry mod 2^16
```

```
//      y(n)=18000 * y(n-1) + carry mod 2^16
//  to produce a combined PRNG with a period of about 2^60.
//  The Pentium cycle counter is used to initialise it. This
//  is crude but the IV does not really need to be secret.

void cycles(volatile unsigned __int64 *rtn)
{
#if defined(_MSC_VER)
    __asm   // read the Pentium Time Stamp Counter
    {   cpuid
        rdtsc
        mov     ecx,rtn
        mov     [ecx],eax
        mov     [ecx+4],edx
        cpuid
    }
#else
#include <time.h>
  time_t tt;
  tt     = time(NULL);
  rtn[0] = tt;
  rtn[1] = tt & -36969l;
  return;
#endif
}

#define RAND(a,b) (((a = 36969 * (a & 65535) + (a >> 16)) << 16) + \
                   (b = 18000 * (b & 65535) + (b >> 16))  )

void fillrand(char *buf, const int len)
{   static unsigned long a[2], mt = 1, count = 4;
    static char         r[4];
    int                 i;

    if(mt) { mt = 0; cycles((unsigned __int64 *)a); }

    for(i = 0; i < len; ++i)
    {
        if(count == 4)
        {
            *(unsigned long*)r = RAND(a[0], a[1]);
            count = 0;
        }
```

```
        buf[i] = r[count++];
    }
}

int encfile(FILE *fin, FILE *fout, aes_ctx *ctx, const char* ifn, const char* ofn)
{   char            buf[BLOCK_LEN], dbuf[2 * BLOCK_LEN];
    fpos_t          flen;
    unsigned long   i, len, rlen;

    // set a random IV

    fillrand(dbuf, BLOCK_LEN);

    // find the file length

    fseek(fin, 0, SEEK_END);
    fgetpos(fin, &flen);
    rlen = file_len(flen);
    // reset to start
    fseek(fin, 0, SEEK_SET);

    if(rlen <= BLOCK_LEN)
    {   // if the file length is less than or equal to 16 bytes

        // read the bytes of the file into the buffer and verify length
        len = (unsigned long) fread(dbuf + BLOCK_LEN, 1, BLOCK_LEN, fin);
        rlen -= len;
        if(rlen > 0)
            return READ_ERROR;

        // pad the file bytes with zeroes
        for(i = len; i < BLOCK_LEN; ++i)
            dbuf[i + BLOCK_LEN] = 0;

        // xor the file bytes with the IV bytes
        for(i = 0; i < BLOCK_LEN; ++i)
            dbuf[i + BLOCK_LEN] ^= dbuf[i];

        // encrypt the top 16 bytes of the buffer
        aes_enc_blk(dbuf + BLOCK_LEN, dbuf + len, ctx);

        len += BLOCK_LEN;
        // write the IV and the encrypted file bytes
        if(fwrite(dbuf, 1, len, fout) != len)
```

```
        return WRITE_ERROR;
}
else
{   // if the file length is more 16 bytes

    // write the IV
    if(fwrite(dbuf, 1, BLOCK_LEN, fout) != BLOCK_LEN)
        return WRITE_ERROR;

    // read the file a block at a time
    while(rlen > 0 && !feof(fin))
    {
        // read a block and reduce the remaining byte count
        len = (unsigned long)fread(buf, 1, BLOCK_LEN, fin);
        rlen -= len;

        // verify length of block
        if(len != BLOCK_LEN)
            return READ_ERROR;

        // do CBC chaining prior to encryption
        for(i = 0; i < BLOCK_LEN; ++i)
            buf[i] ^= dbuf[i];

        // encrypt the block
        aes_enc_blk(buf, dbuf, ctx);

        // if there is only one more block do ciphertext stealing
        if(rlen > 0 && rlen < BLOCK_LEN)
        {
            // move the previous ciphertext to top half of double buffer
            // since rlen bytes of this are output last
            for(i = 0; i < BLOCK_LEN; ++i)
                dbuf[i + BLOCK_LEN] = dbuf[i];

            // read last part of plaintext into bottom half of buffer
            if(fread(dbuf, 1, rlen, fin) != rlen)
                return READ_ERROR;

            // clear the remainder of the bottom half of buffer
            for(i = 0; i < BLOCK_LEN - rlen; ++i)
                dbuf[rlen + i] = 0;

            // do CBC chaining from previous ciphertext
```

```
            for(i = 0; i < BLOCK_LEN; ++i)
               dbuf[i] ^= dbuf[i + BLOCK_LEN];

            // encrypt the final block
            aes_enc_blk(dbuf, dbuf, ctx);

            // set the length of the final write
            len = rlen + BLOCK_LEN; rlen = 0;
         }

         // write the encrypted block
         if(fwrite(dbuf, 1, len, fout) != len)
            return WRITE_ERROR;
      }
   }

   return 0;
}

int decfile(FILE *fin, FILE *fout, aes_ctx *ctx, const char* ifn, const char* ofn)
{  char          buf1[BLOCK_LEN], buf2[BLOCK_LEN], dbuf[2 * BLOCK_LEN];
   char          *b1, *b2, *bt;
   fpos_t        flen;
   unsigned long  i, len, rlen;

   // find the file length

   fseek(fin, 0, SEEK_END);
   fgetpos(fin, &flen);
   rlen = file_len(flen);
   // reset to start
   fseek(fin, 0, SEEK_SET);

   if(rlen <= 2 * BLOCK_LEN)
   {   // if the original file length is less than or equal to 16 bytes

      // read the bytes of the file and verify length
      len = (unsigned long)fread(dbuf, 1, 2 * BLOCK_LEN, fin);
      rlen -= len;
      if(rlen > 0)
         return READ_ERROR;

      // set the original file length
      len -= BLOCK_LEN;
```

```
    // decrypt from position len to position len + BLOCK_LEN
    aes_dec_blk(dbuf + len, dbuf + BLOCK_LEN, ctx);

    // undo CBC chaining
    for(i = 0; i < len; ++i)
        dbuf[i] ^= dbuf[i + BLOCK_LEN];

    // output decrypted bytes
    if(fwrite(dbuf, 1, len, fout) != len)
        return WRITE_ERROR;
}
else
{   // we need two input buffers because we have to keep the previous
    // ciphertext block - the pointers b1 and b2 are swapped once per
    // loop so that b2 points to new ciphertext block and b1 to the
    // last ciphertext block

    rlen -= BLOCK_LEN; b1 = buf1; b2 = buf2;

    // input the IV
    if(fread(b1, 1, BLOCK_LEN, fin) != BLOCK_LEN)
        return READ_ERROR;

    // read the encrypted file a block at a time
    while(rlen > 0 && !feof(fin))
    {
        // input a block and reduce the remaining byte count
        len = (unsigned long)fread(b2, 1, BLOCK_LEN, fin);
        rlen -= len;

        // verify the length of the read operation
        if(len != BLOCK_LEN)
            return READ_ERROR;

        // decrypt input buffer
        aes_dec_blk(b2, dbuf, ctx);

        // if there is only one more block do ciphertext stealing
        if(rlen > 0 && rlen < BLOCK_LEN)
        {
            // read last ciphertext block
            if(fread(b2, 1, rlen, fin) != rlen)
                return READ_ERROR;
```

```
            // append high part of last decrypted block
            for(i = rlen; i < BLOCK_LEN; ++i)
                b2[i] = dbuf[i];

            // decrypt last block of plaintext
            for(i = 0; i < rlen; ++i)
                dbuf[i + BLOCK_LEN] = dbuf[i] ^ b2[i];

            // decrypt last but one block of plaintext
            aes_dec_blk(b2, dbuf, ctx);

            // adjust length of last output block
            len = rlen + BLOCK_LEN; rlen = 0;
        }

        // unchain CBC using the last ciphertext block
        for(i = 0; i < BLOCK_LEN; ++i)
            dbuf[i] ^= b1[i];

        // write decrypted block
        if(fwrite(dbuf, 1, len, fout) != len)
            return WRITE_ERROR;

        // swap the buffer pointers
        bt = b1, b1 = b2, b2 = bt;
    }
  }

  return 0;
}

int main(int argc, char *argv[])
{  FILE        *fin = 0, *fout = 0;
  char        *cp, ch, key[32];
  int         i, by, key_len, err = 0;
  aes_ctx     ctx[1];

  if(argc != 5 || toupper(*argv[3]) != 'D' && toupper(*argv[3]) != 'E')
  {
    printf("usage: xam in_filename out_filename [d/e] key_in_hex\n");
    err = -1; goto exit;
  }
```

```
ctx->n_rnd = 0; // ensure all flags are initially set to zero
ctx->n_blk = 0;
cp = argv[4];   // this is a pointer to the hexadecimal key digits
i = 0;          // this is a count for the input digits processed

while(i < 64 && *cp)        // the maximum key length is 32 bytes and
{                           // hence at most 64 hexadecimal digits
   ch = toupper(*cp++);    // process a hexadecimal digit
   if(ch >= '0' && ch <= '9')
      by = (by << 4) + ch - '0';
   else if(ch >= 'A' && ch <= 'F')
      by = (by << 4) + ch - 'A' + 10;
   else                     // error if not hexadecimal
   {
      printf("key must be in hexadecimal notation\n");
      err = -2; goto exit;
   }

   // store a key byte for each pair of hexadecimal digits
   if(i++ & 1)
      key[i / 2 - 1] = by & 0xff;
}

if(*cp)
{
   printf("The key value is too long\n");
   err = -3; goto exit;
}
else if(i < 32 || (i & 15))
{
   printf("The key length must be 32, 48 or 64 hexadecimal digits\n");
   err = -4; goto exit;
}

key_len = i / 2;

if(!(fin = fopen(argv[1], "rb")))   // try to open the input file
{
   printf("The input file: %s could not be opened\n", argv[1]);
   err = -5; goto exit;
}

if(!(fout = fopen(argv[2], "wb")))  // try to open the output file
{
```

```
        printf("The input file: %s could not be opened\n", argv[1]);
        err = -6; goto exit;
    }

    if(toupper(*argv[3]) == 'E') // encryption in Cipher Block Chaining mode
    {
        aes_enc_key(key, key_len, ctx);

        err = encfile(fin, fout, ctx, argv[1], argv[2]);
    }
    else                        // decryption in Cipher Block Chaining mode
    {
        aes_dec_key(key, key_len, ctx);

        err = decfile(fin, fout, ctx, argv[1], argv[2]);
    }
exit:
    if(err == READ_ERROR)
        printf("Error reading from input file: %s\n", argv[1]);

    if(err == WRITE_ERROR)
        printf("Error writing to output file: %s\n", argv[2]);

    if(fout)
        fclose(fout);

    if(fin)
        fclose(fin);

    return err;
}
```

The various header files that were used (all open-source code of AES):

```
/*
 ---------------------------------------------------------------------------
 Copyright (c) 2001, Dr Brian Gladman <brg@gladman.me.uk>, Worcester, UK.
 All rights reserved.

 LICENSE TERMS

 The free distribution and use of this software in both source and binary
 form is allowed (with or without changes) provided that:
```

1. distributions of this source code include the above copyright
   notice, this list of conditions and the following disclaimer;

2. distributions in binary form include the above copyright
   notice, this list of conditions and the following disclaimer
   in the documentation and/or other associated materials;

3. the copyright holder's name is not used to endorse products
   built using this software without specific written permission.

DISCLAIMER

This software is provided 'as is' with no explicit or implied warranties
in respect of its properties, including, but not limited to, correctness
and fitness for purpose.

Issue Date: 29/07/2002

This file contains the definitions required to use AES (Rijndael) in C.
*/

```
#ifndef _AES_H
#define _AES_H

/*  This include is used only to find 8 and 32 bit unsigned integer types   */

#include "limits.h"

#if UCHAR_MAX == 0xff       /* an unsigned 8 bit type for internal AES use  */
  typedef unsigned char      aes_08t;
#else
#error Please define an unsigned 8 bit type in aes.h
#endif

#if UINT_MAX == 0xffffffff  /* an unsigned 32 bit type for internal AES use */
  typedef   unsigned int     aes_32t;
#elif ULONG_MAX == 0xffffffff
  typedef   unsigned long    aes_32t;
#else
#error Please define an unsigned 32 bit type in aes.h
#endif

/*  BLOCK_SIZE is in BYTES: 16, 24, 32 or undefined for aes.c and 16, 20,
    24, 28, 32 or undefined for aespp.c.  When left undefined a slower
```

```
   version that provides variable block length is compiled.
*/

#define BLOCK_SIZE  16

/* key schedule length (in 32-bit words)    */

#if !defined(BLOCK_SIZE)
#define KS_LENGTH   128
#else
#define KS_LENGTH   4 * BLOCK_SIZE
#endif

#if defined(__cplusplus)
extern "C"
{
#endif

typedef unsigned int aes_fret;   /* type for function return value       */
#define aes_bad      0          /* bad function return value           */
#define aes_good     1          /* good function return value          */
#ifndef AES_DLL                 /* implement normal or DLL functions    */
#define aes_rval     aes_fret
#else
#define aes_rval     aes_fret __declspec(dllexport) _stdcall
#endif


typedef struct                  /* the AES context for encryption   */
{   aes_32t    k_sch[KS_LENGTH];   /* the encryption key schedule      */
    aes_32t    n_rnd;           /* the number of cipher rounds      */
    aes_32t    n_blk;           /* the number of bytes in the state */
} aes_ctx;

#if !defined(BLOCK_SIZE)
aes_rval aes_blk_len(unsigned int blen, aes_ctx cx[1]);
#endif

aes_rval aes_enc_key(const unsigned char in_key[], unsigned int klen, aes_ctx cx[1]);
aes_rval aes_enc_blk(const unsigned char in_blk[], unsigned char out_blk[], const aes_ctx
cx[1]);

aes_rval aes_dec_key(const unsigned char in_key[], unsigned int klen, aes_ctx cx[1]);
aes_rval aes_dec_blk(const unsigned char in_blk[], unsigned char out_blk[], const aes_ctx
```

```
cx[1]);

#if defined(__cplusplus)
}
#endif

#endif


/*
 ---------------------------------------------------------------------------
 Copyright (c) 2001, Dr Brian Gladman <brg@gladman.me.uk>, Worcester, UK.
 All rights reserved.

 LICENSE TERMS

 The free distribution and use of this software in both source and binary
 form is allowed (with or without changes) provided that:

   1. distributions of this source code include the above copyright
      notice, this list of conditions and the following disclaimer;

   2. distributions in binary form include the above copyright
      notice, this list of conditions and the following disclaimer
      in the documentation and/or other associated materials;

   3. the copyright holder's name is not used to endorse products
      built using this software without specific written permission.

 DISCLAIMER

 This software is provided 'as is' with no explicit or implied warranties
 in respect of its properties, including, but not limited to, correctness
 and fitness for purpose.
 ---------------------------------------------------------------------------
 Issue Date: 15/01/2002
*/

#include <iostream>
#include <fstream>
#include <strstream>
#include <iomanip>
#include <cctype>
```

```
typedef unsigned char   byte;
typedef unsigned long   word;

enum line_type { bad_line = 0, block_len, key_len, test_no, iv_val, key_val, pt_val, ct_val };
#define NO_LTYPES   8
#define BADL_STR    "BADLINE="
#define BLEN_STR    "BLOCKSIZE="
#define KLEN_STR    "KEYSIZE=  "
#define TEST_STR    "TEST= "
#define IV_STR      "IV=   "
#define KEY_STR     "KEY=  "
#define PT_STR      "PT=   "
#define CT_STR      "CT=   "

char      *file_name(char* buf, const word type, const word blen, const word klen);
char      *copy_str(char *s, const char *fstr);
bool      get_line(std::ifstream& inf, char s[]);
void      block_out(const line_type ty, const byte b[], std::ofstream& outf, const word len);

int       find_string(const char *s1, const char s2[]);
line_type find_line(std::ifstream& inf, char str[]);

word    rand32(void);
byte    rand8(void);
int     block_in(byte l[], const char *p);
void    block_clear(byte l[], const word len);
void    block_reverse(byte l[], const word len);
void    block_copy(byte l[], const byte r[], const word len);
void    block_xor(byte l[], const byte r[], const word len);
bool    block_cmp(const byte l[], const byte r[], const word len);
void    block_rndfill(byte l[], word len);

void    put_dec(char *s, word val);
word    get_dec(const char *s);
int     cmp_nocase(const char *s1, const char *s2);
bool    test_args(int argc, char *argv[], char des_chr, char tst_chr);
```