Rhode Island College
Digital Commons @ RIC

Honors Projects Overview Honors Projects

2007

PROE: Pseudo Random Optimized Encryption

Louis J. Ricci
Rhode Island College, R22lou@cox.net

Follow this and additional works at: https://digitalcommons.ric.edu/honors_projects

Part of the Other Computer Sciences Commons, and the Theory and Algorithms Commons

Recommended Citation
Ricci, Louis J., "PROE: Pseudo Random Optimized Encryption" (2007). *Honors Projects Overview*. 7.
https://digitalcommons.ric.edu/honors_projects/7

This Honors is brought to you for free and open access by the Honors Projects at Digital Commons @ RIC. It has been accepted for inclusion in Honors Projects Overview by an authorized administrator of Digital Commons @ RIC. For more information, please contact digitalcommons@ric.edu.

PROE: PSEUDO RANDOM OPTIMIZED ENCRYPTION

By

Louis J. Ricci

An Honors Project Submitted in Partial Fulfillment

of the Requirements for Honors

in

The Department of Computer Science

School of Mathematics and Computer Science

Rhode Island College

2007

PROE: PSEUDO RANDOM OPTIMIZED

ENCRYPTION

FORM

An Undergraduate Honors Project Presented

By

Louis J. Ricci

To

: Computer Science Department

Approved:

Project Advisor	Date
Chair, Department Honors Committee	Date
Department Chair	Date

Copyright © by Louis J. Ricci

2007

Abstract

PROE is an encryption algorithm designed for the most modern systems and performance requirements. PROE is implemented in x86-64 assembler to ensure a high level of performance. This thesis examines the development and testing of the PROE algorithm, including the important design decisions ensuring PROE's security and speed.

The PROE algorithm uses large keys to seed its internal random number generator. By using a fast and secure random number generator PROE is able to quickly encrypt and decrypt large blocks of data. When compared with the current leading encryption algorithm (AES) it is shown that PROE is at least 30% faster and achieves an equal level of security.

Table of Contents

Terms and Acronyms 1

Section 1: Introduction 3

1.1 What is encryption? 3

1.2 The Advanced Encryption Standard (AES) 3

1.3 Optimization 5

1.4 64-bit Computing 8

1.5 What is the PROE algorithm? 9

Section 2: The PROE Specification 11

2.1 Pseudo Random Number Generator 11

2.2 Pseudo Random Bit Rolling 13

2.3 Checksum 14

2.4 Encryption and Decryption 16

Section 3: Security 20

Section 4: Testing 22

4.1 Overview 22

4.2 Test Plans 23

4.3 Test Results 24

4.3.1 Security 25

4.3.2 Performance 26

4.3.3 Proof of Concept – File Encryption 27

Section 5: Future Considerations and Conclusions 30

Bibliography 32

Appendix A: PROE_Lib_Win64 source code 34

Appendix B: Lib_Test source code...61

Appendix C: PROE File Encryption source code..70

Appendix D: PROE File Decryption source code 103

Terms and Acronyms

National Institute of Standards and Technology (NIST) – A federal technology agency that develops and promotes measurement, standards, and technology.

Advanced Encryption Standard (AES) – The current recognized national standard for encryption in the United States. AES represents an algorithm that has been verified by NIST.

Federal Information Processing Standards Publications (FIPS) or (FIPS PUBS) – The articles published by NIST representing the guidelines for a variety of information processing standards.

x86-64 – A microprocessor architecture which uses the standard x86 instruction set as well as the 64 bit extension of that instruction set.

Extended Multimedia Extensions (XMMX) – A mnemonic used for the CPU registers (special memory inside of the microprocessor) used by instructions especially designed to work on vectorized (parallel) data.

Substitution Box (S-Box) – A term used in cryptography to represent a two-dimensional array of values. This is very similar to the general programming term known as a lookup table. This two-dimensional array is usually indexed into by portions of the input value and the output is the value found at the indexed location in the array.

Pseudo Random Number Generator (PRNG) - A computer algorithm used to create a sequence of numbers which statistically seem to be random. If the algorithm's state is the same as another instance of itself than both PRNG's will produce the same sequence of numbers.

Flat Assembler (FASM) – The compiler used for all of the PROE source code, including the testing program and proof of concept file encryption/decryption programs. <http://www.flatassembler.net>.

NIST Statistical Test Suite – A RNG testing program.

<http://csrc.nist.gov/rng/rng2.html>.

OpenSSL – The method used to benchmark AES

<http://www.openssl.org/>.

ENT – A pseudo random number sequence test program.

<http://www.fourmilab.ch/random/>.

DIEHARD – A RNG testing program.

<http://www.csis.hku.hk/~diehard/cdrom/>.

Section 1: Introduction

1.1 What is encryption?

Encryption is a process by which data is transformed (encrypted), so that it can only be used by its intended audience. There are two major categories of encryption: symmetric key algorithms and asymmetric key algorithms. Symmetric key algorithms involve using the same secret key for encrypting and decrypting data, while asymmetric key algorithms use a public key which everyone has access to for encrypting data and a private key which is secret for decrypting the data. This thesis covers the former.

Symmetric key algorithms can be further broken down into stream ciphers and block ciphers. Stream ciphers encrypt data at the byte level, taking in one byte at a time and manipulating it based on the cipher's state. Block ciphers manipulate blocks of data which are fixed in length in which the same algorithm is applied to each block.

1.2 The Advanced Encryption Standard (AES)

AES is the Advanced Encryption Standard currently advocated by the US Government. The encryption algorithm's original name is Rijndael. AES is a block cipher that supports block sizes of 128 bits and key sizes of 128, 192 or 256 bits.

The number of rounds (iterations) performed by AES is based on the key size. For a key size of 128 bits 9 rounds are used, and for a key size of 256 13 rounds are used. Every round in AES consists of four operations, which are Add Round Key, Substitute Bytes, Shift Rows and Mix Columns. After the standard rounds a special round is added to the end which doesn't use the Mix Columns operation.

The AES algorithm refers to its input block as a state. Since the block is made up of 16 bytes the state can be considered a two-dimensional array with four columns and four rows. This state is used by the Mix Columns and Shift Rows operations.

The key schedule algorithm used by AES enlarges the initial key so that every round will have a unique sub key the same number of bytes as the input block. Expanding the initial key gives the algorithm more secret data to mix with the data being encrypted. This helps to ensure the security of the encrypted data. In the Add Round Key operation the bytes from the round key are XORed with the corresponding bytes of the input block. Because the Add Round Key is an XOR operation, the same operation is done for decryption.

The Mix Columns operation in AES combines the four bytes that make up a column of the state using a matrix multiply. Each column is multiplied by a matrix, but since the AES algorithm operates on a finite field the multiplication and addition that make up the matrix multiply are replaced with a set of more complex shifting and XORing operations. For decryption an inverse of the encryption matrix is used.

The finite field used by AES is 2^8. This is because the values that make up a state are limited to the set of numbers in one byte which is 256 (2^8). The following will demonstrate the Mix Columns step for encryption on one column of the state.

Column		Matrix		Calculations (note that *2 is equivalent to a 1 bit shift to the left)		Result
[219]		[2, 3, 1, 1]		[(219*2 XOR 27) XOR (69) XOR (83) XOR (19*2) XOR (19)]		[142]
[19]	X	[1, 2, 3, 1]	=	[(19*2) XOR (219) XOR (69) XOR (83*2) XOR (83)]	=	[77]
[83]		[1, 1, 2, 3]		[(83*2) XOR (19) XOR (219) XOR (69*2) XOR (69)]		[161]
[69]		[3, 1, 1, 2]		[(69*2) XOR (83) XOR (19) XOR (219*2 XOR 27) XOR (219)]		[188]

The Shift Rows operation in AES make sure the resulting state has columns that contain values from all of the initial state's columns. The first row of the state is not changed, but the second, third and fourth rows are shifted to the left by 1, 2 and 3 respectively. The following is a visual representation of the operation.

[i00 i01 i02 i03]		[i00 i01 i02 i03]
[i10 i11 i12 i13]	(Shift Row) =	[i11 i12 i13 i10]
[i20 i21 i22 i23]		[i22 i23 i20 i21]
[i30 i31 i32 i33]		[i33 i30 i31 i32]

For decryption the reverse operation is performed shifting the rows right instead of left.

The Substitute Byte operation replaces the bytes of the state with their corresponding value found in the AES algorithm's main substitution box (S-Box). The S-Box is a 16x16 two-dimensional array. The lookup is performed by taking the input byte and using the upper (most significant) four bits

as the index into the row and the lower (least significant) four bits as the index into the column of the S-Box. For decryption the inverse of the encryption S-Box is used.

AES was chosen as a point of comparison for the speed design of PROE because it is internationally recognized as a standard in cryptography. For a detailed AES specification read FIPS PUB 197.

1.3 Optimization

When the goal is efficiency, optimization can be an important part of algorithm design. Many of the optimization techniques introduced here are used in PROE. Most processor level optimization is done after a proper algorithm is chosen. No matter how optimized an algorithm is, a bubble sort will still run slower than a radix sort. Choosing the best algorithm involves picking the most efficient solution to the problem. In our case the problem is quickly and securely encrypting and decrypting large blocks of data.

When working with an architecture that has the ability to process data simultaneously, through SIMD instructions and multiple cores we should try to make our algorithm as parallel as possible.

SIMD means single instruction multiple data. It is the basis for the x86 architectures SSE (streaming SIMD extensions). A SIMD instruction is able to process a vector of data that is 128 bits in size. The PROE algorithm uses unsigned integer vectors in conjunction with the SSE instruction set. SIMD instructions are present on all of todays x86-64 processors.

Making an algorithm parallel involves processing the greatest amount of data per step. For example, when summing a large array you can iterate through the array one element at a time and add it to the total, or you could sum both halves of the array at the same time and then add the two totals' together. The latter would be a parallel solution. By using SIMD instructions the PROE algorithm can process more data, faster.

Another method of parallel execution is using multiple threads. The PROE algorithm itself doesn't make use of multiple threads of execution, but the proof of conception implementation does. The program shows one method of using multiple threads to encrypt and decrypt data faster than a single thread would be able to. Some algorithms like CRC32 (cyclic redundancy check 32 bit) are impossible to implement with multiple threads. This is because the algorithm's state is changed based on previous byte values. Since only one byte can be processed at a time there's no room for parallel execution.

The way a computer iterates through an array can also be optimized. By iterating in reverse we can avoid having to compare the loop index with the size of the array. This is because when a subtraction operation is done on x86 processors a flag bit in the processor is automatically set telling us if the result is less than zero (this flag is called the sign bit flag). The following two examples will demonstrate this concept.

```
Slow:
Index = 0
Label1:                                  // Here are the assembler instructions for
        Sum = Sum + Array[Index]         // the important section of the code
        Index = Index + 1                // INC   [Index]              (Increment)
        IF Index < Array.length THEN     // CMP Index, Array.length    (Compare)
                GOTO Label1:             // JL    .Label1              (Branch If Less Than)
        END IF

Fast:
Index = Array.length - 1
Label1:                                  // Here are the assembler instructions for
        Sum = Sum + Array[Index]         // the important section of the code
        Index = Index - 1                // DEC   [Index]              (Decrement)
        IF NOT SIGN FLAG THEN
                GOTO Label1:             // JNS   .Label1              (Branch If Not Sign Flag)
        END IF
```

As you can see the fast implementation does not have to use the CMP instruction when deciding whether to loop or not.

Lookup tables have been widely used to speed up operations in algorithms. A lookup table is a portion of memory that has previously calculated results in it. For example, if we had a 2 bit number (0,

1, 2, or 3) and we wanted to multiply it by 5, then add 3 and finally divide by 2 we could directly perform these operations in our algorithm or we could use a lookup table to save a good deal of execution time. The following two examples will outline this concept.

Slow:
```
Result = Variable
Result = Result * 5
Result = Result + 3
Result = Result / 2
```

Fast:
```
Lookup_Table = {1, 4, 6, 9}
Result = Lookup_Table [Variable]
```

Hardware optimization is done on the instructions that make up an algorithm. An example would be using a bit shift instruction as opposed to a slower multiply or divide instruction. Because all of the current x86-64 architectures are similar they share many of the same hardware optimizations. Hardware optimizations aren't just based on instruction choice but also address alignment, data dependencies, and processor caching.

Address alignment plays an important role when optimizing an algorithm for a specific architecture. An aligned address is an address whose value is a multiple of some power of two. By aligning addresses used for branching, loading data, or storing data to a multiple of 16 bytes we can improve the performance of those instructions on x86-64 architectures.

Avoiding data dependencies is another important hardware optimization. A stale is when the processor has to wait for one operation to complete before another can complete, because of a data dependency. Interleaving instructions so that the same piece of memory isn't consecutively written to then read from will improve performance.

Processor cache can be accessed very quickly, but processor cache is finite and only a fraction of the size of main memory. Accessing data sequentially and using prefetching instructions when accessing exceptionally large amounts of a data are good ways of optimizing an algorithm. Also, keeping the size of the code being executed to a minimum can help avoid cache misses. Recursive functions are suboptimal in terms of cache performance. This is because every recursive call creates

entries on the calling process's stack and if a large enough volume of calls are made then those entries can evict other important data from the cache memory.

Loop unrolling is a very useful hardware optimization because it helps to avoid unnecessary branching. Loop unrolling means repeating the inner instructions of a loop so that less comparisons with the loop counter need to be made and/or less branches are executed. The following is an example of unrolling a loop to improve the performance of a sum-of-all-integers-in-an-array algorithm.

```
FUNCTION SUM_ARRAY (Array as Integer[])
    DIM X AS INTEGER = 0
    DIM TOTAL AS INTEGER = 0
    DIM TEMP AS INTEGER
    LABEL1:
    IF X <= Array.length – 2 THEN
        TEMP = Array[X] + Array[X+1]
        X = X + 2
        TOTAL = TOTAL + TEMP
        GOTO LABEL1
    END IF
    IF X = Array.length – 1 THEN
        TOTAL = TOTAL + Array[X]
    END IF
    RETURN TOTAL
END FUNCTION
```

In an array of size N the number of comparisons being made by the algorithm is N/2 + 1, which is still O (N) time but is effectively smaller which improves the performance of the algorithm. Loop unrolling works best when the loop is fairly compact and the number of iterations is large. In the larger loop bodies and small iterations of the PROE algorithm unrolling is less effective.

1.4 64-bit Computing

64-bit computing became mainstream when AMD released their line of K8 processors in mid 2003. These new processors extended the 32-bit instruction set to support 64-bit computing. Improvements included 64-bit wide general purpose registers (GPR), addressability of 1 terabyte of RAM, a 256 terabyte virtual addresses space, double the previous amount of GPR's and XMM registers (from 8 to 16), and the addition of the SSE/2/3 streaming SIMD extensions.

Software that fully takes advantage of 64-bit computing is still rare. This is partially due to the industry's slow adoption of new technologies and the unwillingness of software engineers to create two versions of their software (one native 32-bit and the other native 64-bit). Because the 64-bit architecture is an extension of the x86 (PC) architecture it is often referred to as x86-64.

1.5 What is the PROE algorithm?

The PROE algorithm is a block and stream cipher hybrid which takes full advantage of the x86-64 architecture. It is unique because the encryption and decryption algorithms are coded almost entirely using the streaming SIMD instruction set extensions (SSE/2) and the block and key sizes are much larger than those found in current block ciphers. For example the AES uses a block size of 128 bits while the block size used by PROE is 512 bits, also the largest key size used by AES is 256 bits while PROE uses a key size of 1024 bits.

Both the encryption and decryption algorithms have three major parts, the most important being the pseudo random number generator (PRNG) which uses the encryption key as a seed. The key is then manipulated by the PRNG and from that result random data is created and XORed with the input block. The next part is the bit rolling portion; this further obfuscates the input block by rotating the bits of the input by a pseudo randomly selected value. The third part is the checksum function; this was implemented because data integrity is an important part of encryption. The same checksum values are produced with the encryption and decryption algorithms, so if the same original data (plain text for encryption and the cipher text produced from the plain text for decryption) and the original key are used as input for both algorithms the checksums will be equal. These values can be used to make sure that the encrypted data was not modified. It should be noted that the checksum function was not designed to be secure in and of itself, this is because the checksum for an encrypted piece of data should be considered as secret as the encryption key used.

The PROE algorithm uses a fairly large 128 byte key for encryption and decryption. The algorithm will not work properly if a key of all null bytes is chosen. Keys should be chosen in as random a manner as possible. Key selection is left up to the user, a hardware true random number generator could be used or simply another secure PRNG algorithm.

Section 2: The PROE Specification

2.1 Pseudo Random Number Generator

The PRNG algorithm makes up the largest portion of PROE. It's responsible for manipulating PROE's 128 byte key (Seed) and the 64 byte input data. The PRNG changes the Seed in such a way that when it's compressed from 128 to 64 bytes those bytes are pseudo random. The random bytes are XORed with the 64 bytes of input data to create the new partially encrypted or decrypted data.

PRNG is called multiple times (eight is recommended) for each block of input data. This is done to ensure that the process can not be reversed and that no Seed data can be derived from the output. Using the PRNG algorithm in this manner makes it cryptographically secure (a level of security suitable for cryptography). PROE's security will be discussed in greater detail in the following section.

The following is pseudo-code for PROE's PRNG algorithm with a thorough description.

```
CONST DQWORD = 16 byte word

CONST QWORD = 8 byte word
CONST DWORD = 4 byte word
CONST WORD = 2 byte word
PROCEDURE PRNG (  Input as Unsigned Byte[64] IN/OUT,
                  Seed as Unsigned Byte[128] IN/OUT )
        LOCAL A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P as DQWORD
1:      A,B,C,D,E,F,G,H       <-     consecutive DQWORD chunks of Seed
2:      I,J,K,L,M,N,O,P       <-     A,B,C,D,E,F,G,H
3:      I,J,K,L,M,N,O,P       <-     I,J,K,L,M,N,O,P 's QWORD chunks >> 63
4:      E,F,G,H               <-     E,F,G,H 's QWORD chunks << 1
5:      A,B,C,D               <-     A,B,C,D + M,N,O,P summing respective QWORDs individually
6:      E,F,G,H               <-     E,F,G,H OR I,J,K,L
7:      A,B,C,D               <-     A,B,C,D's QWORD chunks Roll_Left 7, 7, 5, 5, 3, 3, 11, 11
8:      A,B,C,D               <-     A,B,C,D Roll_Right 32
9:      B,C,D,E               <-     B,C,D,E + consecutive bytes of Seed[0...63]
                                            summing respective bytes individually
10:     F,G,H,A               <-     F,G,H,A XOR Seed(64...128]
11:     Seed                  <-     B,C,D,E,F,G,H,A
12:     I,J,K,L,M,N,O,P       <-     A,B,C,D,E,F,G,H
13:     A,G                   <-     A,G 's WORD chunks << 3
14:     B,H                   <-     B,H 's WORD chunks << 5
15:     C,E                   <-     C,E 's WORD chunks << 7
16:     D,F                   <-     D,F 's WORD chunks << 11
17:     L,N                   <-     L,N 's WORD chunks >> 3
18:     I,O                   <-     I,O 's WORD chunks >> 5
19:     J,P                   <-     J,P 's WORD chunks >> 7
```

20: K,M <- K,M 's WORD chunks >> 11
21: A,B,C,D <- A,B,C,D XOR I,J,K,L
22: E,F,G,H <- E,F,G,H + M,N,O,P summing respective bytes individually
23: A,B,C <- A,B,C XOR E,F,G
24: D <- D + H summing respective bytes individually
25: A,B,C,D <- A,B,C,D XOR consecutive DQWORD chunks of Input
26: Input <- A,B,C,D
END PROCEDURE

The PRNG takes in two parameters, the 64 byte input that represents the data being encrypted or decrypted and the 128 byte seed. Both of the parameters can be read from and modified by the algorithm. The first thing the PRNG does is make two copies of the seed in local memory. Think of the copies as being split into 16 eight byte pieces known as QWORDs (steps 1-2).

The QWORDs from the second copy are shifted 63 bits to the right so the most significant bit becomes the least significant bit and all of the other bits are cleared to 0 (step 3). Next the right most 8 QWORDs from the first copy are shifted 1 bit to the left, which clears the least significant bit (step 4). Now the left most 8 QWORDS from the first copy are summed with the right most 8 QWORDs of the the second copy, also the right most 8 QWORDs from the first copy are logically ORed with the left most 8 QWORDs from the second copy (steps 5-6). The second copy is now discarded. These steps help to keep the PRNGs period (greatest number of iterations before repeating) high.

The left most 8 QWORDs from the first copy are bit rotated to the left by 7,7,5,5,3,3,11 and 11 bits respectively (step 7). For the next step consider that the consecutive pairs of QWORDs from the left most 8 from the first copy are connected to become 4 sixteen byte pieces known as DQWORDS. Now we'll rotate those DQWORDS to the right by 32 bits (step 8). These steps make sure that different bits are combined with every call to the PRNG.

These steps will take the values we've just created, combine them with the old seed and save them as the new seed. The bytes from the second, third, fourth and fifth DQWORDs from the first copy are now summed with the bytes from the left most 4 DQWORDs of the seed (step 9). Next, the sixth, seventh, eighth and first DQWORDs from the first copy are XORed with the right most 4 DQWORDs

from the seed (step 10). The seed is now replaced with the second through eighth and then first DQWORDs from the first copy (step 11). The seed rotation is a key portion of the algorithm. It allows a higher level of security by distributing the seed data fully over eight calls to the PRNG and it helps keep the encrypted output as random looking as possible.

These final steps will combine our 8 DQWORDS into 4, XOR them with the input and save the new values as the updated input. We will start by copying the 8 DQWORDs from the first copy and for simplicity the original 8 DQWORDS will be referenced as 1 through 8 (from left to right) and the 8 copied DQWORDs will be referred to as 9 through 16 (from left to right) (step 12). For the following step the consecutive two byte pieces of the DQWORDs known as WORDs will be shifted by prime numbers. DQWORDs 1 through 8 will be shifted to the left by 3,5,7,11,7,11,3 and 5 bits respectively and DQWORDs 9 though 16 will be shifted to the right by 5,7,11,3,11,3,5 and 7 bits respectively (steps 13-20). Next, DQWORDs 1 through 4 will be XORed with DQWORDs 9 through 12 and the bytes of DQWORDs 5 through 8 will be summed with the bytes from DQWORDs 13 through 16 (steps 21-22). Now DQWORDs 1 through 3 will be XORed with DQWORDS 5 through 7 and the bytes from DQWORD 4 will be summed with the bytes from DQWORD 8 (steps 23-24). Finally, DQWORDs 1 through 4 will be XORed with the four DQWORDs that make up the input and DQWORDs 1 through 4 will be saved as the updated input (steps 25-26). The shifting steps are very important because they remove bits from the data before its combined, which makes it more difficult to extrapolate the original data or the seed that created it.

2.2 Pseudo Random Bit Rolling

The bit rolling algorithm is called after a user defined number of calls to the PRNG algorithm are completed on a block. The number of calls to PRNG are synonymous with the number of passes (eight is recommended). This algorithm takes two parameters the 64 byte input which represents the

data being encrypted and the index value which can be an integer from 0 to 255. The bit rolling algorithm essentially rotates the QWORDS (8 byte words) that make up the input by a parameter defined number of bits. For encryption the bit rolling algorithm rolls the bits to the right and for decryption the opposite function is done, so the bits are rolled to the left.

The following is the pseudo-code for PROE's bit rolling algorithm with a thorough description.

```
CONST QWORD = 8 Byte Word
PROCEDURE Roll_[Right/Left] (        Input as Unsigned QWORD[8] IN/OUT,
                                     Index as Unsigned Byte IN )
        LOCAL Value_A as Unsigned Byte
        LOCAL A,B,C,D,E,F,G,H as Unsigned QWORD
1:      A,B,C,D,E,F,G,H        <-    consecutive QWORD chunks of Input
2:      Value_A                <-    Index AND 63
3:      A,B,C,D,E,F,G,H        <-    A,B,C,D,E,F,G,H Roll_[Right/Left] Value_A
4:      Input                  <-    A,B,C,D,E,F,G,H
END PROCEDURE
```

First, roll_[left/right] makes a copy of the input in local memory (step 1). Next the index parameter is logically ANDed with 63 and the result is stored in a local variable (step 2). Now the QWORDs that make up the local copy of the input are rotated by the local variable's value in bits (step 3). The rotation is a right rotation if the algorithm is used for encryption and a left rotation if used for decryption. Finally, the modified local memory is saved as the updated input (step 4).

The bit rolling algorithm is effective because the index parameter is a randomly chosen byte from the seed, which is manipulated by the PRNG. Since QWORDs are 64 bits in size the index parameter is truncated to 6 bits. The random bit rolling adds an extra level of obfuscation to the cipher text. Because of the excellent security offered just by the PRNG this algorithm may not even be necessary.

2.3 Checksum

A checksum is an algorithm-based method of determining the integrity and authenticity of a digital data object, which is used to check whether errors or alterations have occurred during the transmission or storage of a data object. Checksums usually try to combine every byte from the input

into the checksum value sequentially, this is done to get the best distribution of bits. PROE's checksum algorithm works with 64 byte chunks of input and as a result doesn't distribute every byte across the whole checksum value. However, because the seed values are combined with the checksum value, a sufficiently unique checksum is still produced and collisions are avoided. Collisions are caused by two different pieces of data producing the same checksum value.

The following is the pseudo-code for PROE's checksum algorithm, as well as, a thorough description.

```
CONST DQWORD = 16 byte word
CONST DWORD = 4 byte word
PROCEDURE Checksum(          Input as Unsigned Byte[64] IN/OUT,
                             Checksum as Unsigned Byte[16] IN/OUT,
                             Seed as Unsigned Byte[16] IN)
        LOCAL A,B,C,D,E,F,G,H,I,J as Unsigned Byte[16]
1:      A,B,C,D       <-     consecutive DQWORD chunks of Input
2:      I             <-     Checksum
3:      J             <-     I
4:      E,F,G,H       <-     A,B,C,D Roll_Right 32
5:      H,G,F,E       <-     H,G,F,E XOR A,B,C,D
6:      I             <-     I's QWORD chunks >> 3
7:      J             <-     J + Seed summing respective bytes individually
8:      F             <-     F XOR E
9:      H             <-     H XOR G
10:     I             <-     I + H summing respective bytes individually
11:     J             <-     J + F summing respective bytes individually
12:     I             <-     I XOR J
13:     Checksum      <-     I
END PROCEDURE
```

The checksum algorithm is used solely as a data integrity check. Because PROE's checksum algorithm uses seed (key) data the resulting checksum of an encryption should be considered as secret as the key itself. The algorithm takes in three parameters the, the 64 byte input which is the data block being encrypted or decrypted, the previous 16 byte value of the checksum and the first 16 bytes of the seed. The seed parameter is actually the preserved seed before any manipulations for the current block were done.

First, the checksum algorithm makes a copy of the input in local memory, we refer to this data as 1 through 4 representing the four DQWORDs (consecutive 16 byte chunks) that make up the copy of

the input (step 1). Next, two copies of the previous checksum value are made these copies will be called A and B (steps 2-3). DQWORDs 1 through 4 will be copied, the results will be referenced as 5 through 8. DQWORDs 5 through 8 will be rotated to the right by 32 bits (step 4). Now DQWORDs 8, 7, 6 and 5 will be logically XORed with DQWORDs 1 through 4 respectively (step 5). Next, DQWORD A's QWORDs (8 byte pieces) will be shifted to the right by 3 bits and DQWORD B's bytes will be summed with the bytes making up the seed parameter (steps 6-7).

Now DQWORD 6 is XORed with DQWORD 5 and DQWORD 8 is XORed with DQWORD 7 (steps 8-9). DQWORD A's bytes are summed with DQWORD 8's bytes and DQWORD B's bytes are summed with DQWORD 6's bytes (steps 10-11). Finally DQWORD A is XORed with DQWORD B and the result is saved as the updated checksum (steps 12-13).

2.4 Encryption and Decryption

The Encryption and Decryption algorithms were designed to take in an array of data with a length that is a multiple of 64 bytes, a 128 byte seed which is the initial encryption key, a 16 byte checksum value which starts as zero and the number of passes or iterations of PRNG to perform on each 64 byte block. The two algorithms are almost identical. The three variances are the swapping of the Checksum and Rolling algorithm calls, and the use of the SeedSave variable. SeedSave preserves the original state of the Seed before the PRNG algorithm manipulates it this makes sure the Encryption and Decryption checksum values are equal when the data provided and the encryption key are the same.

One of the odd things you may notice about the Encrypt and Decrypt algorithms is that they iterate through the Data array in reverse. This was simply an optimization decision because traversing arrays in reverse allows you to avoid unneeded comparisons in the loop logic. Also, notice how the Index and IndexSave variables are updated. This updating is pseudo random because it's based on

randomly selected values from the Seed.

The following is the pseudo-code for the PROE Encryption algorithm and a detailed description. Because references to earlier sections in this chapter (PROE's Specifications) are made it is recommended that you read the previous sections first. For an x86-64 assembler implementation of this algorithm please refer to Appendix A.

```
CONST QWORD = 8 byte word
PROCEDURE Encrypt (          Data as Unsigned Byte[] IN/OUT
                                     ASSUME Data.length Mod 64 = 0,
                             Seed as Unsigned Byte[128] IN/OUT,
                             Checksum as Unsigned Byte[16] IN/OUT,
                             Passes   as Unsigned Byte IN )
        LOCAL Offset as Unsigned QWORD
        LOCAL Index,IndexSave as Unsigned Byte
1:      Offset                  <-      Data.length - 64
2:      IndexSave               <-      0
3:      FOR(X = Offset; X>=0; X=X-64)
4:              Index           <-      Seed[IndexSave]
5:              IndexSave       <-      (IndexSave + Index)
6:              IndexSave       <-      IndexSave AND 127
7:              CALL Checksum (Data[Offset ... Offset + 63], Checksum, Seed[0 ... 15])
8:              FOR(Y = 0; Y<Passes; Y++)
9:                      CALL PRNG (Data[Offset ... Offset + 63], Seed)
10:             NEXT Y
11:             CALL Roll_Right(Data[Offset ... Offset + 63], Index)
12:     NEXT X
END PROCEDURE
```

The algorithm starts by setting the values of two of its local variables, Offset is set to the length of Data minus 64 this is essentially the last 64 bytes of Data and IndexSave is set to zero (steps 1-2). Next, Index is set to the byte value of the Seed referenced by IndexSave (step 4). IndexSave is now summed with Index and then logically ANDed with 127 (steps 5-6). Checksum is called using the current block of Data from Offset to Offset + 63, the checksum parameter and the first 16 bytes from the Seed (step 7). Next, the PRNG is called using the current Data block from Offset to Offset + 63 and the Seed (step 9). The PRNG calling step is repeated based on the Passes parameter (steps 8-10). Finally, the Roll_Right algorithm is called using the current Data block from Offset to Offset + 63 and the Index variable, also Offset is decremented by 64 (steps 11-12). The next block of Data is now processed in a similar fashion meaning steps 3 through 12 are repeated as long as Offset is not less

than zero (steps 3-12).

The following is the pseudo-code for the PROE Decryption algorithm and a detailed description. Because references to earlier sections in this chapter (PROE's Specifications) are made it is recommended that you read the previous sections first. For an x86-64 bit assembler implementation of this algorithm please refer to Appendix A.

```
CONST QWORD = 8 byte word
CONST DQWORD = 16 byte word
PROCEDURE Decrypt (          Data as Unsigned Byte[] IN/OUT
                                     ASSUME Data.length Mod 64 = 0,
                             Seed as Unsigned Byte[128] IN/OUT,
                             Checksum as Unsigned Byte[16] IN/OUT,
                             Passes   as Unsigned Byte IN)
        LOCAL SeedSave as DQWORD
        LOCAL Offset as Unsigned QWORD
        LOCAL Index,IndexSave as Unsigned Byte
1:      Offset               <-      Data.length - 64
2:      IndexSave            <-      0
3:      FOR(X = Offset; X>=0; X=X-64)
4:              SeedSave     <-      Seed[0 ... 15] the first DQWORD chunk of Seed
5:              Index        <-      Seed[IndexSave]
6:              IndexSave    <-      IndexSave + Index
7:              IndexSave    <-      IndexSave AND 127
8:              CALL Roll_Left (Data[Offset ... Offset + 63], Index)
9:              FOR(Y = 0; Y<Passes; Y++)
10:                     CALL PRNG (Data[Offset ... Offset + 63], Seed)
11:             NEXT Y
12:             CALL Checksum (Data[Offset ... Offset + 63], Checksum, SeedSave)
13:     NEXT X
END PROCEDURE
```

The Decryption algorithm is designed to take in an array of data with a length that is a multiple of 64 bytes, a 128 byte seed which is the initial encryption key, a 16 byte checksum value which starts as zero and the number of passes or iterations of PRNG to perform on each 64 byte block. The algorithm starts by setting the values of two of its local variables, Offset is set to the length of Data minus 64 this is essentially the last 64 bytes of Data and IndexSave is set to zero (steps 1-2). Now the SeedSave variable is set to the value of the first DQWORD (16 bytes) in the Seed (step 4). Next, Index is set to the byte value of the Seed referenced by IndexSave (step 5). IndexSave is now summed with Index and then logically ANDed with 127 (steps 6-7). The Roll_Left algorithm is called using the current Data block from Offset to Offset + 63 and the Index variable (step 8). Next, the PRNG is called

using the current Data block from Offset to Offset + 63 and the Seed (step 10). The PRNG calling step is repeated based on the Passes parameter (steps 9-11). Finally, Checksum is called using the current block of Data from Offset to Offset + 63, the checksum parameter and the SeedSave variable, also Offset is decremented by 64 (steps 12-13). The next block of Data is now processed in a similar fashion meaning steps 3 through 13 are repeated as long as Offset is not less than zero (steps 3-13).

Section 3: Security

PROE was built with the intention of being a very secure as well as a very fast encryption algorithm. The PRNG that is the backbone of the PROE algorithm was tested using a variety of programs including the NIST Statistical Test Suite. These tests showed that the randomness that the RNG was able to achieve is suitable for cryptographic use.

The NIST Statistical Test Suite reads large data files and determines if the randomness of the bytes in those data files is suitable for cryptographic use. A data file is created by running a PRNG and storing the resulting data into the file. The NIST Suite uses a variety of statistical tests for instance, counting the largest runs of 0's and 1's and making sure they fall into a determined distribution.

The PRNG is designed to be cryptographically secure. Being cryptographically secure means its random output and algorithm are secure against attack. The German Certification Body and NIST have outlined requirements for security in respect to cryptography. Two tests important for security include the n-bits test and the state compromise test.

The n-bits test states that if the first n bits of the generator's output are known then the next bit (n+1) cannot be derived with a probability greater than 50%. The n-bits test is part of Andrew Chi-chih Yao's work on random numbers and cryptography, for which he won the Turing Award in 2000. Initial analysis shows that PROE is able to avoid bit patterns by using a combination of XOR and unsigned ADD repeated over each other with rotations by prime numbers. Before the output is derived from the current seed, bits are shifted out of the seed so only a partial state is used to derive the output.

A PRNG being used for cryptography should be very resistant to attack. The state compromise extension test states that if during execution the state of all the memory pertaining to the PRNG is known there should be no way of acquiring previous values generated by the PRNG. PROE's PRNG is secure in regards to this attack because the previous state of the seed is combined by XOR and ADD operations with the seed replacing it. The replacement is rotated so over eight calls to the PRNG any

new seed is fully mixed with the original.

While randomness is important it's not the only security concern. The PROE algorithm also needed to be built so that an attacker would be unable to derive the algorithm's state from a portion of cipher text in which the original plain text is already known. The features of the PRNG that aid in this are the key rotation portion as well as the bit shifting after the keys state has been saved. The bit shifting removes 2-8 bits from every WORD (16 bits) in the key before it is compressed into a random piece of data to XOR with the cipher text, while the key rotation completes a full cycle in eight passes (iterations). A recommended pass count of eight was chosen because of these factors.

Analyzing the strength of the PROE algorithm is important, especially in the case of an attacker knowing the original plain text of part of a message and trying to guess all or part of the seed that created the cipher text. Since the PRNG can't be implemented in reverse, because the current state is based on a combination of all previous states; the attacker would have to use probability and iterations of guessing and checking. Having knowledge of the plain test means that the attacker knows what was XORed with the 64 bytes of random output from the PRNG at least eight rounds ago. But knowing that doesn't tell the attacker anything about the state of the generator or its seed. This leaves the attacker with reversing at least eight passes of the PRNG. The PRNG uses a lot of XOR and ADD binary operations so running those in reverse require the other values or guessing them. Now reversing the full state of the PRNG at once wouldn't be smart, because the PRNG uses 128 bytes so it could take 10^308 number of guesses. Breaking the data down into smaller parts would be a much smarter approach, but because of the bit shifting used before the 64 random bytes were created even deriving sporadic bits from the seed would be tedious. Calculating the dependencies one bit has over one pass comes to 5, so over two passes all six bits each depend on five other bits and so on. Going to eight passes, the task becomes almost impossible for the attacker.

Section 4: Testing

4.1 Overview

PROE's performance and security were tested using a variety of programs and techniques. Randomness testing on PROE's PRNG was done using programs such as ENT, DIEHARD, and the NIST Statistical Test Suite. Performance testing was accomplished by building a benchmarking program for PROE, which acted in a congruent fashion to an AES benchmark built into OpenSSL and then comparing the results. Finally, real world implementation testing used a proof of concept file encryption and decryption program which used the PROE algorithm.

Building a fast and random PRNG isn't a difficult task, but making sure that the PRNG can produce cryptographically useful levels of randomness is a different challenge. The random output needs to be free of any patterns or signs that would make it not statistically random. The randomness testing programs return p-values for their various tests. These p-values can range from 0.0 to 1.0 and represent the probability that the test results came from random data. A successful p-value is usually in the range of 0.02 to 0.999, while anything less than 0.01 means the data has a strong probability of not being random. P-values of 1.0 can mean the data may be too random. This would be similar to rolling a six-sided die 60 times and getting exactly 10 rolls with each number. Truly random results tend to have a slight variance from the statistical ideal.

Testing PROE's PRNG and benchmarking the encryption and decryption speed were accomplished using a small testing program written in x86-64 assembler. The program performed two tasks, creating random output files using the PRNG and running an encryption benchmark calculating speed and other measurements.

The PROE encryption and decryption algorithm's were implemented in four x86-64 assembler routines EncryptBufferWithChecksum, DecryptBufferWithChecksum, EncryptBuffer, and

DecryptBuffer. To test PROE in a real world scenario two dialog programs were built to encrypt and decrypt files. The encrypt program also had the capability of creating the 128 byte keys that PROE uses.

4.2 Test Plans

The bulk of the security related testing was done using the NIST Suite, while ENT and DIEHARD were used during the initial design process. Testing the PRNG involved making large binary files filled with output from the PRNG (usually 100 MB in size) and allowing the NIST Suite to analyze the files. The seeds for the PRNG were generated using values from the processors nanosecond timer. The high precision of the timer ensured that the seed values would be different between consecutive runs of the test program. If the NIST Suite returned failing results, then the PRNG was modified. This process was repeated until the PRNG consistently passed the NIST Suite's tests. Version 1.8 of the NIST Suite was used for these tests. The following example shows p-values from the NIST Statistical Test Suite. All input files were 10 MB in size.

FILE:	**Output from PROE's PRNG**
P-VALUES:	**0.612438 0.701306 0.985837 0.080322 0.748275 0.968397 0.212097**
FILE:	**jakarta-tomcat-5.0.30.exe an executable file**
P-VALUES:	**0.000000 0.000000 0.000000 0.000000 0.000102 0.000000 0.000000**
FILE:	**A multiply with carry generator of the form x(n) = a*x(n-1) + carry mod M**
P-VALUES:	**0.024512 0.038853 0.029271 0.964925 0.000377 0.999486 0.000000**
FILE:	**A PRNG included in the NIST Suite that it calls XOR**
P-VALUES:	**0.266744 0.416268 0.219565 0.271232 0.942205 0.230305 0.228477**

A pen and paper analysis of the PRNG was done to ensure the security of the algorithm itself. The analysis included tracing the algorithm looking for weaknesses that an attacker could use to compromise PROE's security. If a weakness was found, corrections to the algorithm would be made and the NIST Suite's tests would be repeated to make sure the modifications didn't interfere with the PRNG's random output.

PROE's encryption speed was tested by comparing the byte per second speed obtained by the test program with that of OpenSSL's AES benchmark. Data (8192 bytes in size) would be encrypted

thousands of times. The total time from the repeated encrypting would be divided into the product of the data size and the number of iterations. These speed benchmarks were performed several times on two different x86-64 processors to ensure the accuracy of the results.

OpenSSL is an open source implementation of the secure sockets layer (SSL). SSL is a protocol used to safely transfer sensitive information over the Internet. OpenSSL uses efficient implementations of many encryption algorithms and it's benchmarking feature made it an excellent choice for testing PROE's speed versus its built in AES implementation. Version 0.9.8d of OpenSSL was used for these tests.

The file encrypting and decrypting programs were created to test PROE in a real world scenario. These proof of concept programs read file data into buffers, encrypt or decrypt the buffered data and then write the data to an output file. To encompass all aspects of file encryption a 128 byte key generator was built into the encryption program and both programs maintained key information using key files. These key files would contain the size of the file being encrypted, the encryption key used, the file path and the pass count used by PROE.

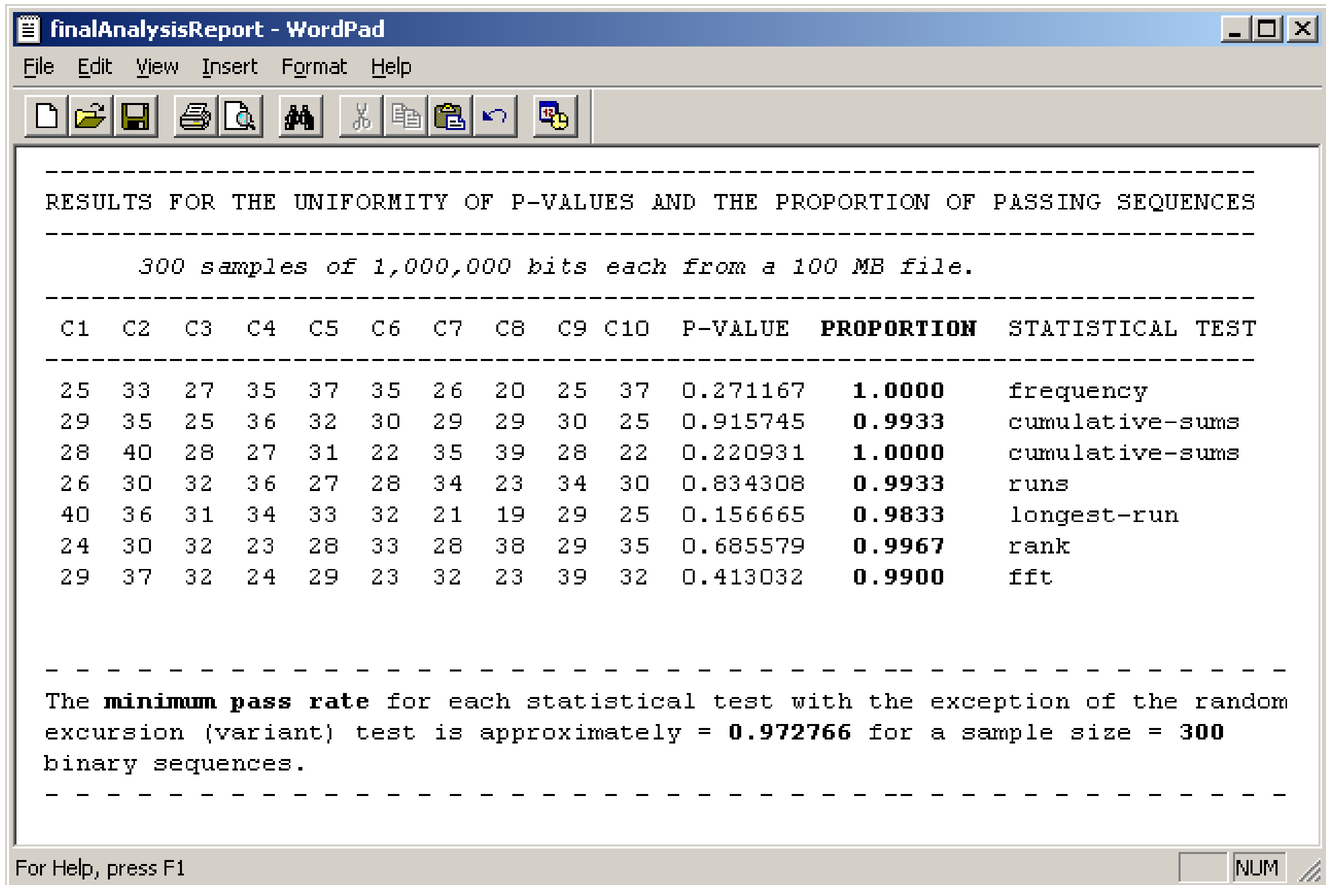
Testing the proof of concept programs was very simple. Files of varying size and type would be encrypted. The encrypted files would be examined to make sure their data was scrambled. The encrypted files would then be decrypted and compared with their originals to make sure they were exactly the same.

4.3 Test Results

The following sections represent a subset of all of the testing results for PROE and its components. Benchmark results were chosen to represent the average of all of the result values obtained. Because of this, there should be no problem reproducing the benchmark results on similar hardware.

4.3.1 Security

The following shows the results of a NIST Suite analysis of random output from PROE's PRNG. The NIST Suite generates a proportion that represents the percentage of samples that passed each statistical test. It also calculates a minimum value, which represents the cut off for being random or not. These critical values are represented in bold.



RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES

300 samples of 1,000,000 bits each from a 100 MB file.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	**PROPORTION**	STATISTICAL TEST
25	33	27	35	37	35	26	20	25	37	0.271167	**1.0000**	frequency
29	35	25	36	32	30	29	29	30	25	0.915745	**0.9933**	cumulative-sums
28	40	28	27	31	22	35	39	28	22	0.220931	**1.0000**	cumulative-sums
26	30	32	36	27	28	34	23	34	30	0.834308	**0.9933**	runs
40	36	31	34	33	32	21	19	29	25	0.156665	**0.9833**	longest-run
24	30	32	23	28	33	28	38	29	35	0.685579	**0.9967**	rank
29	37	32	24	29	23	32	23	39	32	0.413032	**0.9900**	fft

The **minimum pass rate** for each statistical test with the exception of the random excursion (variant) test is approximately = **0.972766** for a sample size = **300** binary sequences.

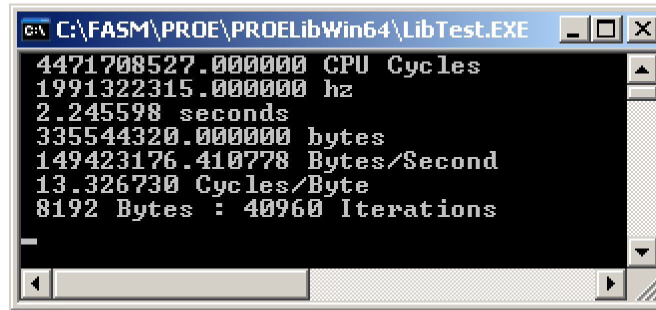
PROE's PRNG was able to pass the NIST Suite's tests repeatedly.

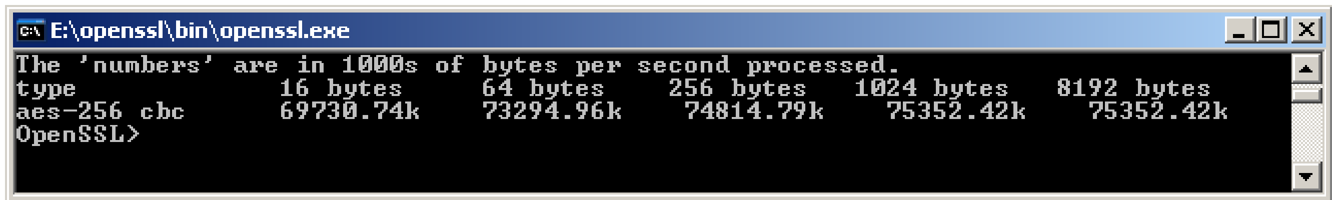
In an early implementation of the PRNG analysis showed that if the current state of the PRNG was revealed then an attacker would be able to run the PROE algorithm in reverse until the original key was obtained. This was corrected by combining all previous states of the PRNG with the current state. By doing this the algorithm became more secure because it was no longer reversible.

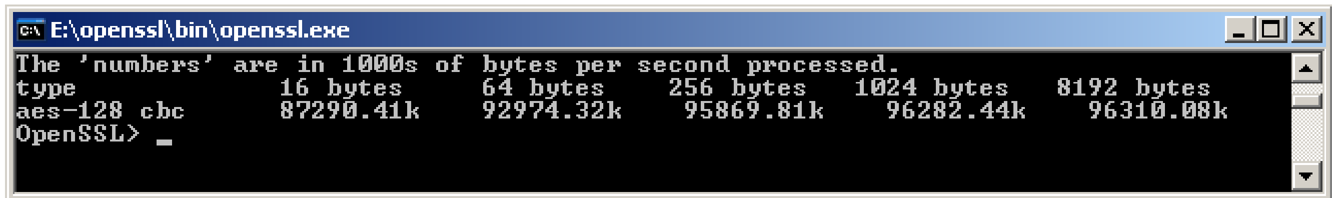
4.3.2 Performance

The benchmarks were performed on two x86-64 processors running in different test systems.

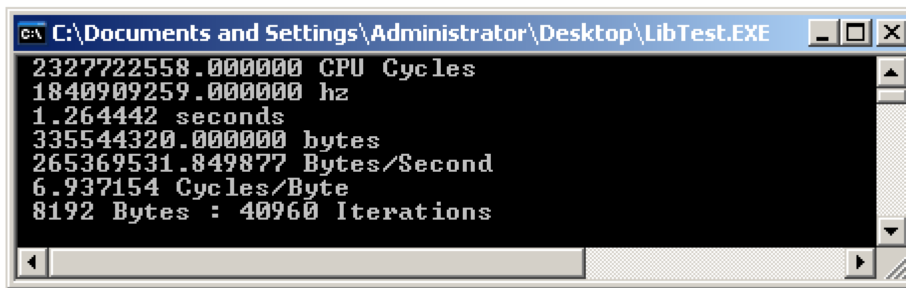
The following screen captures will show the test program's results PROE and OpenSSL's results for AES.







System: AMD64 x2 3800+ (2.0 Ghz) with 1GB of RAM. OS: Window XP 64 Bit.



```
C:\openSSL\bin\openssl.exe
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-256 cbc      65065.80k    71820.27k    72067.08k    73552.02k    73039.69k
OpenSSL>
```

```
C:\openSSL\bin\openssl.exe
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc      85206.79k    92974.32k    95433.54k    95460.69k    95869.81k
OpenSSL>
```

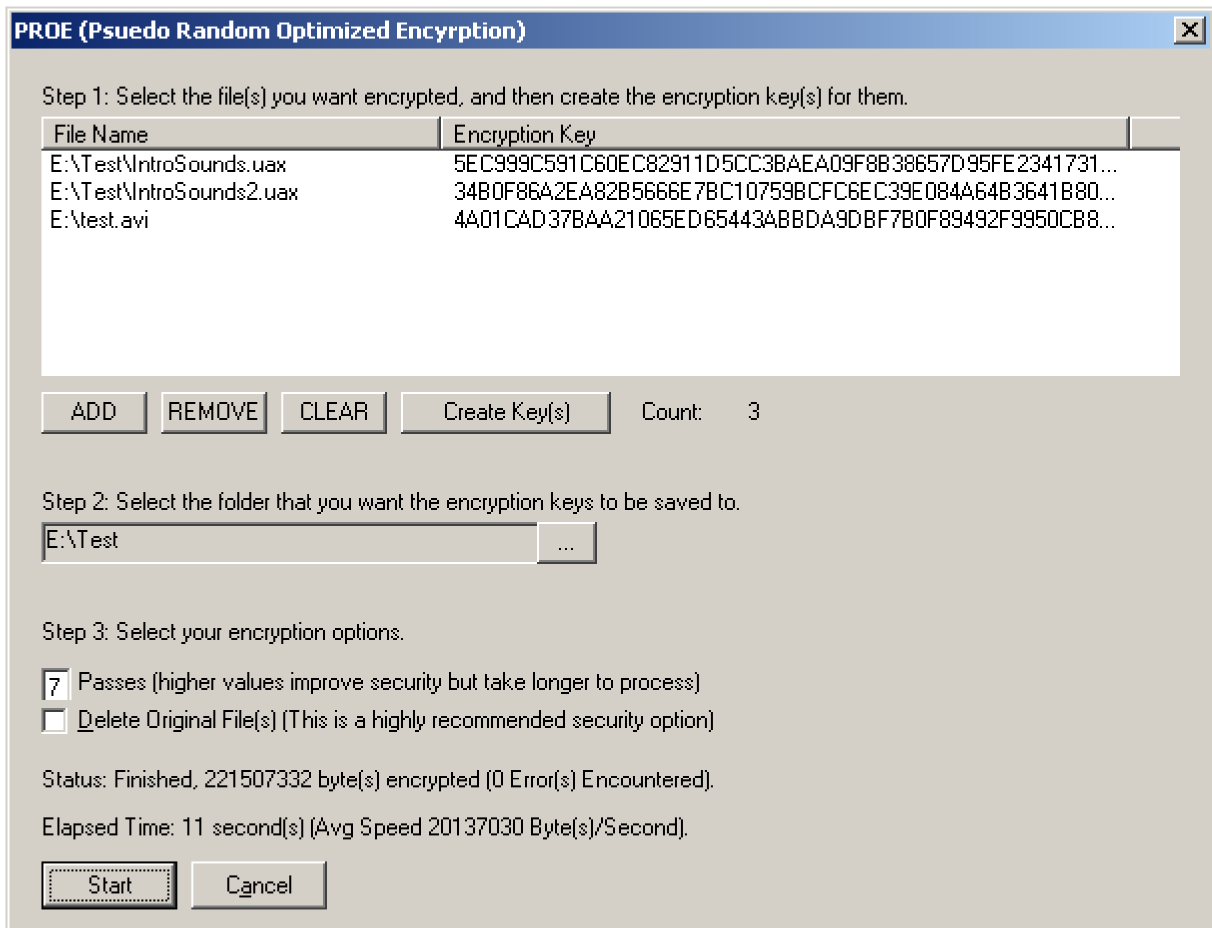
System: Intel Core2Duo e6300 (1.8 Ghz) with 2 GB of RAM. OS: Windows XP 64 Bit.

On the AMD64 system, the PROE algorithm was 1.5 times faster than the 128 bit key implementation of AES. The results from the Intel Core2Duo system show PROE being 2.7 times faster. Because PROE uses almost all SIMD instructions in its implementation, as x86 processors advance or get replaced with more vector oriented processors like IBM's Cell, then PROE's performance and scalability will only improve. This is evident with Intel's latest line of x86-64 processors known as Core2. Core2 enhanced the execution speed of its SIMD instructions by almost 50%. This enhancement can account for the variance in benchmark results between the AMD64 system and the Intel Core2 system.

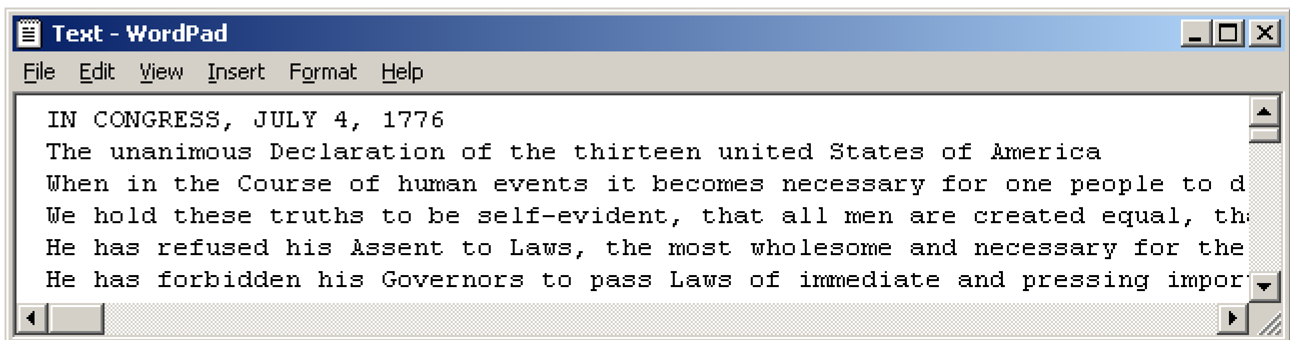
AES is implemented using the standard x86 instruction set, because of how the algorithm is designed. AES is not highly parallel or vectorized so it wouldn't benefit from a SIMD implementation. This seems to be the case for many modern block ciphers.

4.3.3 Proof of Concept – File Encryption

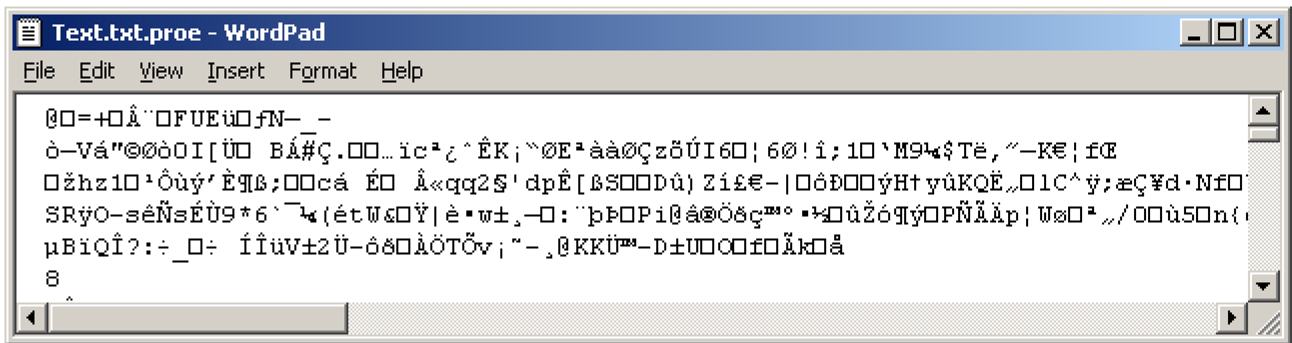
The following is a screen capture of the test program encrypting some files.



The speed of the hard drive is a limiting factor. Although the program uses multiple threads of execution to theoretically encrypt more than one file at a time, because reads and writes to the hard drive are performed sequentially contention for the disk resource occurs often.



These screen captures show the content of a text file before and after encryption.



Once the file was decrypted it returned to its original form with no errors.

Section 5: Future Considerations and Conclusions

The testing concluded that PROE is up to 160% (2.7x) faster than AES and this gap will only increase with newer hardware architectures. Also, the algorithm seems to be secure enough to protect sensitive data very effectively. PROE's development demonstrates how designing with regards to performance and modern architecture can yield formidable results.

The PROE algorithm seems to scale very well, because using multiple keys and threads to encrypt or decrypt should equate to a large performance increase. By using multiple keys and multiple threads of execution it should be easy to encrypt/decrypt different parts of a message in parallel. Key sizes may become exceptionally large if scaling it taken to a supercomputer level (10,000 threads would need a key of 1.28 MB).

The current proof of concept implementation of the PROE algorithm is a dialog application for Windows XP 64 bit ® . The application shows how PROE can be used to encrypt and decrypt files, but file access is such a bottleneck on modern computers that the application can't be used as a benchmark for performance.

Developing a communication protocol using PROE similar to SSL would help to demonstrate the flexibility of the algorithm. Using a public key encryption engine to transfer PROE's private key to the other party would allow peer-to-peer or client-server communication to be done with the better performing algorithm, while still maintaining a high level of security.

PROE was designed specifically for modern hardware and software implementations. It's larger block and key sizes, and its high level of performance are an example of this. Because of considerations in the way of parallelism and being SIMD instruction compatible PROE will be able to move ahead with future technology, which will be more parallel and contain more vector based execution.

Spending more time analyzing the PROE algorithm can lead to further optimizations or

enhancements. Also security concerns not previously thought of could be discovered. With some professional analysis PROE could become a viable means for security for today and future generations.

Bibliography

Anwendungshinweise und Interpretationen zum Schema (AIS) 20, Evaluation of Deterministic Random Number Generators, 1999 February 12. <http://www.bsi.bund.de/zertifiz/zert/interpr/ais20.pdf>.

Barker, Elaine and John Kelsey. NIST Special Publication 800-90, Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), 2007. <http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf>.

"Cryptographically secure pseudorandom number generator." Wikipedia, The Free Encyclopedia. 18 Mar 2007, 12:40 UTC. Wikimedia Foundation, Inc. 5 Nov 2006 <http://en.wikipedia.org/w/index.php?title=Cryptographically_secure_pseudorandom_number_generator>.

Dworkin, Morris. NIST Special Publication 800-38A, Recommendation for Block Cipher Modes of Operation: Methods and Techniques, 2001. <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.

Federal Information Processing Standards Publication (FIPS PUB) 1401, Security Requirements For Cryptographic Modules, 1994 January 11. <http://csrc.nist.gov/publications/fips/fips1401.pdf>.

Federal Information Processing Standards Publication (FIPS PUB) 197, Announcing the ADVANCED ENCRYPTION STANDARD (AES), 2001 November 26. <http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

Kelsey, John. Cryptanalytic Attacks on Pseudorandom Number Generators. <http://www.schneier.com/paper-prngs.pdf>.

"Pseudorandom number generator." Wikipedia, The Free Encyclopedia. 13 Mar 2007, 12:22 UTC. Wikimedia Foundation, Inc. 21 Nov 2007

<http://en.wikipedia.org/w/index.php?title=Pseudorandom_number_generator>.

Rukhin, Andrew, et al. NIST Special Publication 800-22, A Statistical Test Suite For Random And Pseudo Random Number Generators For Cryptographic Applications, 2001. <http://csrc.nist.gov/rng/SP800-22b.pdf>.

Savard, John J. G. “A Cryptographic Compendium.” John Savard's Home Page. 1999. 30 Oct 2006 <http://www.quadibloc.com/crypto/jscrypt.htm>.

Viega, John. Practical Random Number Generation in Software, 2003. <http://www.acsac.org/2003/papers/79.pdf>.

Appendix A: PROE_Lib_Win64 source code

```
;LOUIS J. RICCI
;PROE Library for Windows 64 Bit
;PROE_Lib_Win64.dll
format PE64 DLL
entry DllEntryPoint

macro AMDPad16
{
   virtual
      align 16
      a = $-$$
   end virtual
   if a=1
     db 90h
   end if
   if a=2
     db 66h,90h
   end if
   if a=3
     db 66h,66h,90h
   end if
   if a=4
     db 66h,66h,66h,90h
   end if
   if a=5
     db 66h,66h,90h,66h,90h
   end if
   if a=6
     db 66h,66h,90h,66h,66h,90h
   end if
   if a=7
     db 66h,66h,66h,90h,66h,66h,90h
   end if
   if a=8
     db 66h,66h,66h,90h,66h,66h,66h,90h
   end if
   if a=9
     db 66h,66h,90h,66h,66h,90h,66h,66h,90h
   end if
   if a=10
     db 66h,66h,66h,90h,66h,66h,90h,66h,66h,90h
   end if
   if a=11
     db 66h,66h,66h,90h,66h,66h,66h,90h,66h,66h,90h
   end if
   if a=12
     db 66h,66h,66h,90h,66h,66h,66h,90h,66h,66h,66h,90h
   end if
   if a=13
     db 66h,66h,66h,90h,66h,66h,90h,66h,66h,90h,66h,66h,90h
   end if
   if a=14
     db 66h,66h,66h,90h,66h,66h,66h,90h,66h,66h,90h,66h,66h,90h
   end if
```

```
   if a=15
      db 66h,66h,66h,90h,66h,66h,66h,90h,66h,66h,66h,90h,66h,66h,90h
   end if
}

include '%fasminc%\win64a.inc'

section '.data' data readable writeable
align 16
  StrTestBuffer db 'This is a PROE encryption test
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789',0
               dq 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
align 16
  TestKeyHash1  dq 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
align 16
  TestKeyHash2  dq 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  StrPassed     db 'Working Properly',0
  StrFailed     db 'Not Working Properly',0

align 16
ROLLLUT_SIZE equ 2048
RollLUT:
Repeat 256
      dq ((%-1) And 63)
end repeat
RollLUTI:
Repeat 256
      dq 64 - ((%-1) And 63)
end repeat

section '.code' code readable executable

proc DllEntryPoint hinstDLL,fdwReason,lpvReserved
       mov     eax,TRUE
       ret
endp


AMDPad16
;;;PARAMETERS   RCX: buffer address, 16byte aligned
;;;            RDX: buffer length in bytes, multiple of 64
;;;            R8 : key and hash address, 16 byte aligned, 128+16 bytes in size
;;;            R9 : number of passes, greater than or equal to 1, recommented 2
;;;RETURN       -1=Fail 0=Success
PROEValidateParameters:
       sub     rsp,8*7
       cmp     r9,1
       jl      .fail
       test    rdx,63
       jnz     .fail
       cmp     rcx,-1
       je      .fail
       test    rcx,rcx
       jz      .fail
       cmp     r8,-1
       je      .fail
```

```
        test    r8,r8
        jz      .fail
        xor     eax,eax
        jmp     .end
    .fail:
        mov     eax,-1
    .end:
        add     rsp,8*7
        ret     0



AMDPad16
;;;RETURN    -1=Failed 0=Success
PROESelfTest:
        sub     rsp,8*7
        mov     r8,TestKeyHash1
        mov     r15,0xabcdef89
        mov     rcx,124
    .keyfill1:
        mov     dword[r8+rcx],r15d
        add     r15,r15
        add     r15,r8
        sub     rcx,4
        jns     .keyfill1
        xor     ecx,ecx
        mov     [r8+128],rcx
        mov     [r8+136],rcx
        mov     rcx,StrTestBuffer
        mov     rdx,128
        ;lea    r8,[r8]
        mov     r9,9
        call    PROEEncryptBufferWithChecksum
        mov     r9,TestKeyHash1
        mov     r8,TestKeyHash2
        mov     r15,0xabcdef89
        mov     rcx,124
    .keyfill2:
        mov     dword[r8+rcx],r15d
        add     r15,r15
        add     r15,r9
        sub     rcx,4
        jns     .keyfill2
        xor     ecx,ecx
        mov     [r8+128],rcx
        mov     [r8+136],rcx
        mov     rcx,StrTestBuffer
        mov     rdx,128
        ;lea    r8,[r8]
        mov     r9,9
        call    PROEDecryptBufferWithChecksum

        mov     rcx,[TestKeyHash1+128]
        mov     rdx,[TestKeyHash1+136]
        cmp     rcx,[TestKeyHash2+128]
        jne     .error
```

```
        cmp     rdx,[TestKeyHash2+136]
        jne     .error
        xor     ecx,ecx
        mov     rdx,StrTestBuffer
        mov     r8,StrPassed
        mov     r9,1
        call    [MessageBox]
        xor     eax,eax
        add     rsp,8*7
        ret     0
  .error:
        xor     ecx,ecx
        mov     rdx,StrTestBuffer
        mov     r8,StrFailed
        mov     r9,1
        call    [MessageBox]
        mov     eax,-1
        add     rsp,8*7
        ret     0

DQ1     equ 0
DQ2     equ 16
DQ3     equ 32
DQ4     equ 48
DQ5     equ 64
DQ6     equ 80
DQ7     equ 96
DQ8     equ 112
;;;STRUCTURE    Key and Hash, 16 byte aligned
;;;            128 bytes + 16 bytes
AMDPad16
;;;PARAMETERS   RCX: buffer address, 16byte aligned
;;;            RDX: buffer length in bytes, multiple of 64
;;;            R8 : key and checksum address, 16 byte aligned, 128+16 bytes in size
;;;            R9 : number of passes, greater than or equal to 1, recommented 2
PROEEncryptBufferWithChecksum:
        sub     rsp,8*15
        mov     [rsp+8*10],rbx
        mov     [rsp+8*11],rsi
        mov     [rsp+8*12],rdi
        mov     [rsp+8*13],r12
        mov     [rsp+8*14],r13
        mov     rdi,r8
        mov     r13,r9
;;;hash and encrypt the block of data
        lea     rsi,[rcx+rdx]
        sub     rdx,64
        xor     eax,eax
        lea     rcx,[RollLUT]
AMDPad16
  .encryptBlock:
        movzx   r8,byte[rdi+rax]
        mov     rbx,r13
        add     rax,r8
        sub     rsi,64
        and     eax,127
```

```
;;update hash
      movdqa  xmm0,[rsi+DQ1]
      movdqa  xmm1,[rsi+DQ2]
      movdqa  xmm2,[rsi+DQ3]
      movdqa  xmm3,[rsi+DQ4]
      movdqa  xmm8,dqword[rdi+128]
      pshufd  xmm4,xmm0,00111001b
      pshufd  xmm5,xmm1,00111001b
      pshufd  xmm6,xmm2,00111001b
      pshufd  xmm7,xmm3,00111001b
      movdqa  xmm9,xmm8
      pxor    xmm7,xmm0
      pxor    xmm6,xmm1
      psllq   xmm8,3
      paddb   xmm9,dqword[rdi]
      pxor    xmm5,xmm2
      pxor    xmm4,xmm3
      pxor    xmm7,xmm6
      pxor    xmm5,xmm4
      paddb   xmm8,xmm7
      paddb   xmm9,xmm5
       pxor    xmm8,xmm9
       movdqa  [rdi+128],xmm8
AMDPad16
  .encryptBlock2:
;;setup
      movdqa  xmm0, [rdi+DQ1]
      movdqa  xmm1, [rdi+DQ2]
      movdqa  xmm2, [rdi+DQ3]
      movdqa  xmm3, [rdi+DQ4]
      movdqa  xmm4, [rdi+DQ5]
      movdqa  xmm5, [rdi+DQ6]
      movdqa  xmm6, [rdi+DQ7]
      movdqa  xmm7, [rdi+DQ8]
;;copy
      movdqa  xmm8, xmm0
      movdqa  xmm9, xmm1
      movdqa  xmm10, xmm2
      movdqa  xmm11, xmm3
      movdqa  xmm12, xmm4
      movdqa  xmm13, xmm5
      movdqa  xmm14, xmm6
      movdqa  xmm15, xmm7
;;shift right logical 63bits to have the mask of highest bit
      psrlq   xmm8, 63
      psrlq   xmm9, 63
      psrlq   xmm10, 63
      psrlq   xmm11, 63
      psrlq   xmm12, 63
      psrlq   xmm13, 63
      psrlq   xmm14, 63
      psrlq   xmm15, 63
;;shift left to remove the highest bit and empty the lowest
      psllq   xmm4, 1
      psllq   xmm5, 1
      psllq   xmm6, 1
```

```
        psllq   xmm7, 1
;;add masked bit
        paddq   xmm0, xmm12
        paddq   xmm1, xmm13
        paddq   xmm2, xmm14
        paddq   xmm3, xmm15
;;logical or lowest bit
        por     xmm4, xmm8
        por     xmm5, xmm9
        por     xmm6, xmm10
        por     xmm7, xmm11
;;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
;;Bit ROLL by prime numbers 7, 5, 3, 11
        psllq   xmm0, 7
        psllq   xmm1, 5
        psllq   xmm2, 3
        psllq   xmm3, 11
        psrlq   xmm8, 57;64-7
        psrlq   xmm9, 59;64-5
        psrlq   xmm10, 61;64-3
        psrlq   xmm11, 53;64-11
        por     xmm0, xmm8
        por     xmm1, xmm9
        por     xmm2, xmm10
        por     xmm3, xmm11
;;Dword order switching
        pshufd  xmm0, xmm0, 00111001b
        pshufd  xmm1, xmm1, 00111001b
        pshufd  xmm2, xmm2, 00111001b
        pshufd  xmm3, xmm3, 00111001b
;;xor/add old key with new key
        paddb  xmm1,[rdi+DQ1]
        paddb  xmm2,[rdi+DQ2]
        paddb  xmm3,[rdi+DQ3]
        paddb  xmm4,[rdi+DQ4]
        pxor   xmm5,[rdi+DQ5]
        pxor   xmm6,[rdi+DQ6]
        pxor   xmm7,[rdi+DQ7]
        pxor   xmm0,[rdi+DQ8]
;;Modify Key with rotation of dq words
        movdqa  [rdi+DQ1], xmm1
        movdqa  [rdi+DQ2], xmm2
        movdqa  [rdi+DQ3], xmm3
        movdqa  [rdi+DQ4], xmm4
        movdqa  [rdi+DQ5], xmm5
        movdqa  [rdi+DQ6], xmm6
        movdqa  [rdi+DQ7], xmm7
        movdqa  [rdi+DQ8], xmm0
;;prepare output
        ;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
```

```
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
        ;shift words by primes
        psllw   xmm0, 3
        psllw   xmm1, 5
        psllw   xmm2, 7
        psllw   xmm3, 11
        ;
        psrlw   xmm8, 5
        psrlw   xmm9, 7
        psrlw   xmm10, 11
        psrlw   xmm11, 3
        ;
        psllw   xmm4, 7
        psllw   xmm5, 11
        psllw   xmm6, 3
        psllw   xmm7, 5
        ;
        psrlw   xmm12, 11
        psrlw   xmm13, 3
        psrlw   xmm14, 5
        psrlw   xmm15, 7
        ; add / xor
        pxor    xmm0, xmm8
        pxor    xmm1, xmm9
        pxor    xmm2, xmm10
        pxor    xmm3, xmm11
        paddb   xmm4, xmm12
        paddb   xmm5, xmm13
        paddb   xmm6, xmm14
        paddb   xmm7, xmm15
        pxor    xmm0, xmm4
        pxor    xmm1, xmm5
        pxor    xmm2, xmm6
        paddb   xmm3, xmm7
;;;xor random 64bytes with block
        pxor    xmm0,dqword[rsi+DQ1]
        pxor    xmm1,dqword[rsi+DQ2]
        pxor    xmm2,dqword[rsi+DQ3]
        pxor    xmm3,dqword[rsi+DQ4]
;;;modify block
        movdqa  dqword[rsi+DQ1],xmm0
        movdqa  dqword[rsi+DQ2],xmm1
        movdqa  dqword[rsi+DQ3],xmm2
        movdqa  dqword[rsi+DQ4],xmm3
;;;
        dec     rbx
        jnz     .encryptBlock2
;;;buffer bit shuffling (ROLling) using values from the key
       ; movdqa  xmm0,[rsi+DQ1]
       ; movdqa  xmm1,[rsi+DQ2]
       ; movdqa  xmm2,[rsi+DQ3]
```

```
      ; movdqa  xmm3,[rsi+DQ4]
       movdqa  xmm4,xmm0
       movdqa  xmm5,xmm1
       movdqa  xmm6,xmm2
       movdqa  xmm7,xmm3
       movq    xmm8,[rcx+r8*8]
       movq    xmm9,[rcx+r8*8+ROLLLUT_SIZE] ;;;+SizeOfLUT
       psrlq   xmm0,xmm8
       psrlq   xmm1,xmm8
       psrlq   xmm2,xmm8
       psrlq   xmm3,xmm8
       psllq   xmm4,xmm9
       psllq   xmm5,xmm9
       psllq   xmm6,xmm9
       psllq   xmm7,xmm9
       por     xmm0,xmm4
       por     xmm1,xmm5
       por     xmm2,xmm6
       por     xmm3,xmm7
       movdqa  [rsi+DQ1],xmm0
       movdqa  [rsi+DQ2],xmm1
       movdqa  [rsi+DQ3],xmm2
       movdqa  [rsi+DQ4],xmm3
;;;
       sub     rdx,64
       js      .endEncryptBlock
;;;UNROLL ---------------------------
       movzx   r8,byte[rdi+rax]
       mov     rbx,r13
       add     rax,r8
       sub     rsi,64
       and     eax,127
;;update hash
       movdqa  xmm0,[rsi+DQ1]
       movdqa  xmm1,[rsi+DQ2]
       movdqa  xmm2,[rsi+DQ3]
       movdqa  xmm3,[rsi+DQ4]
       movdqa  xmm8,dqword[rdi+128]
       pshufd  xmm4,xmm0,00111001b
       pshufd  xmm5,xmm1,00111001b
       pshufd  xmm6,xmm2,00111001b
       pshufd  xmm7,xmm3,00111001b
       movdqa  xmm9,xmm8
       pxor    xmm7,xmm0
       pxor    xmm6,xmm1
       psllq   xmm8,3
       paddb   xmm9,dqword[rdi]
       pxor    xmm5,xmm2
       pxor    xmm4,xmm3
       pxor    xmm7,xmm6
       pxor    xmm5,xmm4
       paddb   xmm8,xmm7
       paddb   xmm9,xmm5
        pxor    xmm8,xmm9
        movdqa  [rdi+128],xmm8
AMDPad16
```

```
  .encryptBlock2_2:
;;setup
       movdqa  xmm0, [rdi+DQ1]
       movdqa  xmm1, [rdi+DQ2]
       movdqa  xmm2, [rdi+DQ3]
       movdqa  xmm3, [rdi+DQ4]
       movdqa  xmm4, [rdi+DQ5]
       movdqa  xmm5, [rdi+DQ6]
       movdqa  xmm6, [rdi+DQ7]
       movdqa  xmm7, [rdi+DQ8]
;;copy
       movdqa  xmm8, xmm0
       movdqa  xmm9, xmm1
       movdqa  xmm10, xmm2
       movdqa  xmm11, xmm3
       movdqa  xmm12, xmm4
       movdqa  xmm13, xmm5
       movdqa  xmm14, xmm6
       movdqa  xmm15, xmm7
;;shift right logical 63bits to have the mask of highest bit
       psrlq   xmm8, 63
       psrlq   xmm9, 63
       psrlq   xmm10, 63
       psrlq   xmm11, 63
       psrlq   xmm12, 63
       psrlq   xmm13, 63
       psrlq   xmm14, 63
       psrlq   xmm15, 63
;;shift left to remove the highest bit and empty the lowest
       psllq   xmm4, 1
       psllq   xmm5, 1
       psllq   xmm6, 1
       psllq   xmm7, 1
;;add masked bit
       paddq   xmm0, xmm12
       paddq   xmm1, xmm13
       paddq   xmm2, xmm14
       paddq   xmm3, xmm15
;;logical or lowest bit
       por     xmm4, xmm8
       por     xmm5, xmm9
       por     xmm6, xmm10
       por     xmm7, xmm11
;;copy
       movdqa  xmm8, xmm0
       movdqa  xmm9, xmm1
       movdqa  xmm10, xmm2
       movdqa  xmm11, xmm3
;;Bit ROLL by prime numbers 7, 5, 3, 11
       psllq   xmm0, 7
       psllq   xmm1, 5
       psllq   xmm2, 3
       psllq   xmm3, 11
       psrlq   xmm8, 57;64-7
       psrlq   xmm9, 59;64-5
       psrlq   xmm10, 61;64-3
```

```
        psrlq   xmm11, 53;64-11
        por     xmm0, xmm8
        por     xmm1, xmm9
        por     xmm2, xmm10
        por     xmm3, xmm11
;;Dword order switching
        pshufd  xmm0, xmm0, 00111001b
        pshufd  xmm1, xmm1, 00111001b
        pshufd  xmm2, xmm2, 00111001b
        pshufd  xmm3, xmm3, 00111001b
;;xor/add old key with new key
        paddb  xmm1,[rdi+DQ1]
        paddb  xmm2,[rdi+DQ2]
        paddb  xmm3,[rdi+DQ3]
        paddb  xmm4,[rdi+DQ4]
        pxor   xmm5,[rdi+DQ5]
        pxor   xmm6,[rdi+DQ6]
        pxor   xmm7,[rdi+DQ7]
        pxor   xmm0,[rdi+DQ8]
;;Modify Key with rotation of dq words
        movdqa  [rdi+DQ1], xmm1
        movdqa  [rdi+DQ2], xmm2
        movdqa  [rdi+DQ3], xmm3
        movdqa  [rdi+DQ4], xmm4
        movdqa  [rdi+DQ5], xmm5
        movdqa  [rdi+DQ6], xmm6
        movdqa  [rdi+DQ7], xmm7
        movdqa  [rdi+DQ8], xmm0
;;prepare output
        ;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
        ;shift words by primes
        psllw   xmm0, 3
        psllw   xmm1, 5
        psllw   xmm2, 7
        psllw   xmm3, 11
        ;
        psrlw   xmm8, 5
        psrlw   xmm9, 7
        psrlw   xmm10, 11
        psrlw   xmm11, 3
        ;
        psllw   xmm4, 7
        psllw   xmm5, 11
        psllw   xmm6, 3
        psllw   xmm7, 5
        ;
        psrlw   xmm12, 11
        psrlw   xmm13, 3
```

```
        psrlw   xmm14, 5
        psrlw   xmm15, 7
        ; add / xor
        pxor    xmm0, xmm8
        pxor    xmm1, xmm9
        pxor    xmm2, xmm10
        pxor    xmm3, xmm11
        paddb   xmm4, xmm12
        paddb   xmm5, xmm13
        paddb   xmm6, xmm14
        paddb   xmm7, xmm15
        pxor    xmm0, xmm4
        pxor    xmm1, xmm5
        pxor    xmm2, xmm6
        paddb   xmm3, xmm7
;;;xor random 64bytes with block
        pxor    xmm0,dqword[rsi+DQ1]
        pxor    xmm1,dqword[rsi+DQ2]
        pxor    xmm2,dqword[rsi+DQ3]
        pxor    xmm3,dqword[rsi+DQ4]
;;;modify block
        movdqa  dqword[rsi+DQ1],xmm0
        movdqa  dqword[rsi+DQ2],xmm1
        movdqa  dqword[rsi+DQ3],xmm2
        movdqa  dqword[rsi+DQ4],xmm3
;;;
        dec     rbx
        jnz     .encryptBlock2_2
;;;buffer bit shuffling (ROLling) using values from the key
       ; movdqa  xmm0,[rsi+DQ1]
       ; movdqa  xmm1,[rsi+DQ2]
       ; movdqa  xmm2,[rsi+DQ3]
       ; movdqa  xmm3,[rsi+DQ4]
        movdqa  xmm4,xmm0
        movdqa  xmm5,xmm1
        movdqa  xmm6,xmm2
        movdqa  xmm7,xmm3
        movq    xmm8,[rcx+r8*8]
        movq    xmm9,[rcx+r8*8+ROLLLUT_SIZE] ;;;+SizeOfLUT
        psrlq   xmm0,xmm8
        psrlq   xmm1,xmm8
        psrlq   xmm2,xmm8
        psrlq   xmm3,xmm8
        psllq   xmm4,xmm9
        psllq   xmm5,xmm9
        psllq   xmm6,xmm9
        psllq   xmm7,xmm9
        por     xmm0,xmm4
        por     xmm1,xmm5
        por     xmm2,xmm6
        por     xmm3,xmm7
        movdqa  [rsi+DQ1],xmm0
        movdqa  [rsi+DQ2],xmm1
        movdqa  [rsi+DQ3],xmm2
        movdqa  [rsi+DQ4],xmm3
;;;
```

```
      sub     rdx,64
      jns     .encryptBlock
   .endEncryptBlock:
      mov     rbx,[rsp+8*10]
      mov     rsi,[rsp+8*11]
      mov     rdi,[rsp+8*12]
      mov     r12,[rsp+8*13]
      mov     r13,[rsp+8*14]
      add     rsp,8*15
      ret     0

AMDPad16
;;;PARAMETERS   RCX: buffer address, 16byte aligned
;;;            RDX: buffer length in bytes, multiple of 64
;;;            R8 : key and checksum address, 16 byte aligned, 128+16 bytes in size
;;;            R9 : number of passes, greater than or equal to 1, recommented 2
PROEDecryptBufferWithChecksum:
      sub     rsp,8*15
      mov     [rsp+8*10],rbx
      mov     [rsp+8*11],rsi
      mov     [rsp+8*12],rdi
      mov     [rsp+8*13],r12
      mov     rdi,r8
      mov     r12,r9
;;;hash and encrypt the block of data
      lea     rsi,[rcx+rdx]
      sub     rdx,64
      xor     eax,eax
      lea     rcx,[RollLUT]
AMDPad16
   .decryptBlock:
      movdqa  xmm0,dqword[rdi]
      movzx   r8,byte[rdi+rax]
      mov     rbx,r12
      add     rax,r8
      movdqa  dqword[rsp-24],xmm0
      and     eax,127
      sub     rsi,64
;;;buffer bit shuffling (ROLling) using values from the key
      movdqa  xmm0,[rsi+DQ1]
      movdqa  xmm1,[rsi+DQ2]
      movdqa  xmm2,[rsi+DQ3]
      movdqa  xmm3,[rsi+DQ4]
      movdqa  xmm4,xmm0
      movdqa  xmm5,xmm1
      movdqa  xmm6,xmm2
      movdqa  xmm7,xmm3
      movq    xmm8,[rcx+r8*8]
      movq    xmm9,[rcx+r8*8+ROLLLUT_SIZE] ;;;+SizeOfLUT
      psllq   xmm0,xmm8
      psllq   xmm1,xmm8
      psllq   xmm2,xmm8
      psllq   xmm3,xmm8
      psrlq   xmm4,xmm9
      psrlq   xmm5,xmm9
      psrlq   xmm6,xmm9
```

```
        psrlq   xmm7,xmm9
        por     xmm0,xmm4
        por     xmm1,xmm5
        por     xmm2,xmm6
        por     xmm3,xmm7
        movdqa  [rsi+DQ1],xmm0
        movdqa  [rsi+DQ2],xmm1
        movdqa  [rsi+DQ3],xmm2
        movdqa  [rsi+DQ4],xmm3
AMDPad16
  .decryptBlock2:
;;setup
        movdqa  xmm0, [rdi+DQ1]
        movdqa  xmm1, [rdi+DQ2]
        movdqa  xmm2, [rdi+DQ3]
        movdqa  xmm3, [rdi+DQ4]
        movdqa  xmm4, [rdi+DQ5]
        movdqa  xmm5, [rdi+DQ6]
        movdqa  xmm6, [rdi+DQ7]
        movdqa  xmm7, [rdi+DQ8]
;;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
;;shift right logical 63bits to have the mask of highest bit
        psrlq   xmm8, 63
        psrlq   xmm9, 63
        psrlq   xmm10, 63
        psrlq   xmm11, 63
        psrlq   xmm12, 63
        psrlq   xmm13, 63
        psrlq   xmm14, 63
        psrlq   xmm15, 63
;;shift left to remove the highest bit and empty the lowest
        psllq   xmm4, 1
        psllq   xmm5, 1
        psllq   xmm6, 1
        psllq   xmm7, 1
;;add masked bit
        paddq   xmm0, xmm12
        paddq   xmm1, xmm13
        paddq   xmm2, xmm14
        paddq   xmm3, xmm15
;;logical or lowest bit
        por     xmm4, xmm8
        por     xmm5, xmm9
        por     xmm6, xmm10
        por     xmm7, xmm11
;;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
```

```
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
;;Bit ROLL by prime numbers 7, 5, 3, 11
        psllq   xmm0, 7
        psllq   xmm1, 5
        psllq   xmm2, 3
        psllq   xmm3, 11
        psrlq   xmm8, 57;64-7
        psrlq   xmm9, 59;64-5
        psrlq   xmm10, 61;64-3
        psrlq   xmm11, 53;64-11
        por     xmm0, xmm8
        por     xmm1, xmm9
        por     xmm2, xmm10
        por     xmm3, xmm11
;;Dword order switching
        pshufd  xmm0, xmm0, 00111001b
        pshufd  xmm1, xmm1, 00111001b
        pshufd  xmm2, xmm2, 00111001b
        pshufd  xmm3, xmm3, 00111001b
;;xor/add old key with new key
        paddb  xmm1,[rdi+DQ1]
        paddb  xmm2,[rdi+DQ2]
        paddb  xmm3,[rdi+DQ3]
        paddb  xmm4,[rdi+DQ4]
        pxor   xmm5,[rdi+DQ5]
        pxor   xmm6,[rdi+DQ6]
        pxor   xmm7,[rdi+DQ7]
        pxor   xmm0,[rdi+DQ8]
;;Modify Key with rotation of dq words
        movdqa  [rdi+DQ1], xmm1
        movdqa  [rdi+DQ2], xmm2
        movdqa  [rdi+DQ3], xmm3
        movdqa  [rdi+DQ4], xmm4
        movdqa  [rdi+DQ5], xmm5
        movdqa  [rdi+DQ6], xmm6
        movdqa  [rdi+DQ7], xmm7
        movdqa  [rdi+DQ8], xmm0
;;prepare output
        ;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
        ;shift words by primes
        psllw   xmm0, 3
        psllw   xmm1, 5
        psllw   xmm2, 7
        psllw   xmm3, 11
        ;
        psrlw   xmm8, 5
        psrlw   xmm9, 7
```

```
        psrlw   xmm10, 11
        psrlw   xmm11, 3
        ;
        psllw   xmm4, 7
        psllw   xmm5, 11
        psllw   xmm6, 3
        psllw   xmm7, 5
        ;
        psrlw   xmm12, 11
        psrlw   xmm13, 3
        psrlw   xmm14, 5
        psrlw   xmm15, 7
        ; add / xor
        pxor    xmm0, xmm8
        pxor    xmm1, xmm9
        pxor    xmm2, xmm10
        pxor    xmm3, xmm11
        paddb   xmm4, xmm12
        paddb   xmm5, xmm13
        paddb   xmm6, xmm14
        paddb   xmm7, xmm15
        pxor    xmm0, xmm4
        pxor    xmm1, xmm5
        pxor    xmm2, xmm6
        paddb   xmm3, xmm7
;;;xor random 64bytes with block
        pxor    xmm0,dqword[rsi+DQ1]
        pxor    xmm1,dqword[rsi+DQ2]
        pxor    xmm2,dqword[rsi+DQ3]
        pxor    xmm3,dqword[rsi+DQ4]
;;;modify block
        movdqa  dqword[rsi+DQ1],xmm0
        movdqa  dqword[rsi+DQ2],xmm1
        movdqa  dqword[rsi+DQ3],xmm2
        movdqa  dqword[rsi+DQ4],xmm3
;;;
        dec     rbx
        jnz     .decryptBlock2
;;update hash
;       movdqa  xmm0,[rsi+DQ1]
;       movdqa  xmm1,[rsi+DQ2]
;       movdqa  xmm2,[rsi+DQ3]
;       movdqa  xmm3,[rsi+DQ4]
        movdqa  xmm8,dqword[rdi+128]
        pshufd  xmm4,xmm0,00111001b
        pshufd  xmm5,xmm1,00111001b
        pshufd  xmm6,xmm2,00111001b
        pshufd  xmm7,xmm3,00111001b
        movdqa  xmm9,xmm8
        pxor    xmm7,xmm0
        pxor    xmm6,xmm1
        psllq   xmm8,3
        paddb   xmm9,dqword[rsp-24]
        pxor    xmm5,xmm2
        pxor    xmm4,xmm3
        pxor    xmm7,xmm6
```

```
        pxor    xmm5,xmm4
        paddb   xmm8,xmm7
        paddb   xmm9,xmm5
         pxor    xmm8,xmm9
         movdqa  [rdi+128],xmm8
;;;
        sub     rdx,64
        js      .endDecryptBlock
;;;UNROLL -------------------------------------
        movdqa  xmm0,dqword[rdi]
        movzx   r8,byte[rdi+rax]
        mov     rbx,r12
        add     rax,r8
        movdqa  dqword[rsp-24],xmm0
        and     eax,127
        sub     rsi,64
;;;buffer bit shuffling (ROLling) using values from the key
        movdqa  xmm0,[rsi+DQ1]
        movdqa  xmm1,[rsi+DQ2]
        movdqa  xmm2,[rsi+DQ3]
        movdqa  xmm3,[rsi+DQ4]
        movdqa  xmm4,xmm0
        movdqa  xmm5,xmm1
        movdqa  xmm6,xmm2
        movdqa  xmm7,xmm3
        movq    xmm8,[rcx+r8*8]
        movq    xmm9,[rcx+r8*8+ROLLLUT_SIZE] ;;;+SizeOfLUT
        psllq   xmm0,xmm8
        psllq   xmm1,xmm8
        psllq   xmm2,xmm8
        psllq   xmm3,xmm8
        psrlq   xmm4,xmm9
        psrlq   xmm5,xmm9
        psrlq   xmm6,xmm9
        psrlq   xmm7,xmm9
        por     xmm0,xmm4
        por     xmm1,xmm5
        por     xmm2,xmm6
        por     xmm3,xmm7
        movdqa  [rsi+DQ1],xmm0
        movdqa  [rsi+DQ2],xmm1
        movdqa  [rsi+DQ3],xmm2
        movdqa  [rsi+DQ4],xmm3
AMDPad16
  .decryptBlock2_2:
;;setup
        movdqa  xmm0, [rdi+DQ1]
        movdqa  xmm1, [rdi+DQ2]
        movdqa  xmm2, [rdi+DQ3]
        movdqa  xmm3, [rdi+DQ4]
        movdqa  xmm4, [rdi+DQ5]
        movdqa  xmm5, [rdi+DQ6]
        movdqa  xmm6, [rdi+DQ7]
        movdqa  xmm7, [rdi+DQ8]
;;copy
        movdqa  xmm8, xmm0
```

```
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
;;shift right logical 63bits to have the mask of highest bit
        psrlq   xmm8, 63
        psrlq   xmm9, 63
        psrlq   xmm10, 63
        psrlq   xmm11, 63
        psrlq   xmm12, 63
        psrlq   xmm13, 63
        psrlq   xmm14, 63
        psrlq   xmm15, 63
;;shift left to remove the highest bit and empty the lowest
        psllq   xmm4, 1
        psllq   xmm5, 1
        psllq   xmm6, 1
        psllq   xmm7, 1
;;add masked bit
        paddq   xmm0, xmm12
        paddq   xmm1, xmm13
        paddq   xmm2, xmm14
        paddq   xmm3, xmm15
;;logical or lowest bit
        por     xmm4, xmm8
        por     xmm5, xmm9
        por     xmm6, xmm10
        por     xmm7, xmm11
;;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
;;Bit ROLL by prime numbers 7, 5, 3, 11
        psllq   xmm0, 7
        psllq   xmm1, 5
        psllq   xmm2, 3
        psllq   xmm3, 11
        psrlq   xmm8, 57;64-7
        psrlq   xmm9, 59;64-5
        psrlq   xmm10, 61;64-3
        psrlq   xmm11, 53;64-11
        por     xmm0, xmm8
        por     xmm1, xmm9
        por     xmm2, xmm10
        por     xmm3, xmm11
;;Dword order switching
        pshufd  xmm0, xmm0, 00111001b
        pshufd  xmm1, xmm1, 00111001b
        pshufd  xmm2, xmm2, 00111001b
        pshufd  xmm3, xmm3, 00111001b
;;xor/add old key with new key
        paddb  xmm1,[rdi+DQ1]
```

```
        paddb  xmm2,[rdi+DQ2]
        paddb  xmm3,[rdi+DQ3]
        paddb  xmm4,[rdi+DQ4]
        pxor   xmm5,[rdi+DQ5]
        pxor   xmm6,[rdi+DQ6]
        pxor   xmm7,[rdi+DQ7]
        pxor   xmm0,[rdi+DQ8]
;;Modify Key with rotation of dq words
        movdqa  [rdi+DQ1], xmm1
        movdqa  [rdi+DQ2], xmm2
        movdqa  [rdi+DQ3], xmm3
        movdqa  [rdi+DQ4], xmm4
        movdqa  [rdi+DQ5], xmm5
        movdqa  [rdi+DQ6], xmm6
        movdqa  [rdi+DQ7], xmm7
        movdqa  [rdi+DQ8], xmm0
;;prepare output
        ;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
        ;shift words by primes
        psllw   xmm0, 3
        psllw   xmm1, 5
        psllw   xmm2, 7
        psllw   xmm3, 11
        ;
        psrlw   xmm8, 5
        psrlw   xmm9, 7
        psrlw   xmm10, 11
        psrlw   xmm11, 3
        ;
        psllw   xmm4, 7
        psllw   xmm5, 11
        psllw   xmm6, 3
        psllw   xmm7, 5
        ;
        psrlw   xmm12, 11
        psrlw   xmm13, 3
        psrlw   xmm14, 5
        psrlw   xmm15, 7
        ; add / xor
        pxor    xmm0, xmm8
        pxor    xmm1, xmm9
        pxor    xmm2, xmm10
        pxor    xmm3, xmm11
        paddb   xmm4, xmm12
        paddb   xmm5, xmm13
        paddb   xmm6, xmm14
        paddb   xmm7, xmm15
        pxor    xmm0, xmm4
```

```
       pxor    xmm1, xmm5
       pxor    xmm2, xmm6
       paddb   xmm3, xmm7
;;;xor random 64bytes with block
       pxor    xmm0,dqword[rsi+DQ1]
       pxor    xmm1,dqword[rsi+DQ2]
       pxor    xmm2,dqword[rsi+DQ3]
       pxor    xmm3,dqword[rsi+DQ4]
;;;modify block
       movdqa  dqword[rsi+DQ1],xmm0
       movdqa  dqword[rsi+DQ2],xmm1
       movdqa  dqword[rsi+DQ3],xmm2
       movdqa  dqword[rsi+DQ4],xmm3
;;;
       dec     rbx
       jnz     .decryptBlock2_2
;;update hash
;       movdqa  xmm0,[rsi+DQ1]
;       movdqa  xmm1,[rsi+DQ2]
;       movdqa  xmm2,[rsi+DQ3]
;       movdqa  xmm3,[rsi+DQ4]
       movdqa  xmm8,dqword[rdi+128]
       pshufd  xmm4,xmm0,00111001b
       pshufd  xmm5,xmm1,00111001b
       pshufd  xmm6,xmm2,00111001b
       pshufd  xmm7,xmm3,00111001b
       movdqa  xmm9,xmm8
       pxor    xmm7,xmm0
       pxor    xmm6,xmm1
       psllq   xmm8,3
       paddb   xmm9,dqword[rsp-24]
       pxor    xmm5,xmm2
       pxor    xmm4,xmm3
       pxor    xmm7,xmm6
       pxor    xmm5,xmm4
       paddb   xmm8,xmm7
       paddb   xmm9,xmm5
        pxor    xmm8,xmm9
        movdqa  [rdi+128],xmm8
;;;
       sub     rdx,64
       jns     .decryptBlock
  .endDecryptBlock:
       mov     rbx,[rsp+8*10]
       mov     rsi,[rsp+8*11]
       mov     rdi,[rsp+8*12]
       mov     r12,[rsp+8*13]
       add     rsp,8*15
       ret     0


AMDPad16
;;;PARAMETERS   RCX: buffer address, 16byte aligned
;;;            RDX: buffer length in bytes, multiple of 64
;;;            R8 : key address, 16 byte aligned, 128 bytes in size
;;;            R9 : number of passes, greater than or equal to 1, recommented 2
```

```
PROEEncryptBuffer:
        sub     rsp,8*15
        mov     [rsp+8*10],rbx
        mov     [rsp+8*11],rsi
        mov     [rsp+8*12],rdi
        mov     [rsp+8*13],r12
        mov     [rsp+8*14],r13
        mov     rdi,r8
        mov     r13,r9
;;;hash and encrypt the block of data
        lea     rsi,[rcx+rdx]
        sub     rdx,64
        xor     eax,eax
        lea     rcx,[RollLUT]
AMDPad16
   .encryptBlock:
        movzx   r8,byte[rdi+rax]
        mov     rbx,r13
        add     rax,r8
        sub     rsi,64
        and     eax,127
AMDPad16
   .encryptBlock2:
;;setup
        movdqa  xmm0, [rdi+DQ1]
        movdqa  xmm1, [rdi+DQ2]
        movdqa  xmm2, [rdi+DQ3]
        movdqa  xmm3, [rdi+DQ4]
        movdqa  xmm4, [rdi+DQ5]
        movdqa  xmm5, [rdi+DQ6]
        movdqa  xmm6, [rdi+DQ7]
        movdqa  xmm7, [rdi+DQ8]
;;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
;;shift right logical 63bits to have the mask of highest bit
        psrlq   xmm8, 63
        psrlq   xmm9, 63
        psrlq   xmm10, 63
        psrlq   xmm11, 63
        psrlq   xmm12, 63
        psrlq   xmm13, 63
        psrlq   xmm14, 63
        psrlq   xmm15, 63
;;shift left to remove the highest bit and empty the lowest
        psllq   xmm4, 1
        psllq   xmm5, 1
        psllq   xmm6, 1
        psllq   xmm7, 1
;;add masked bit
```

```
        paddq   xmm0, xmm12
        paddq   xmm1, xmm13
        paddq   xmm2, xmm14
        paddq   xmm3, xmm15
;;logical or lowest bit
        por     xmm4, xmm8
        por     xmm5, xmm9
        por     xmm6, xmm10
        por     xmm7, xmm11
;;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
;;Bit ROLL by prime numbers 7, 5, 3, 11
        psllq   xmm0, 7
        psllq   xmm1, 5
        psllq   xmm2, 3
        psllq   xmm3, 11
        psrlq   xmm8, 57;64-7
        psrlq   xmm9, 59;64-5
        psrlq   xmm10, 61;64-3
        psrlq   xmm11, 53;64-11
        por     xmm0, xmm8
        por     xmm1, xmm9
        por     xmm2, xmm10
        por     xmm3, xmm11
;;Dword order switching
        pshufd  xmm0, xmm0, 00111001b
        pshufd  xmm1, xmm1, 00111001b
        pshufd  xmm2, xmm2, 00111001b
        pshufd  xmm3, xmm3, 00111001b
;;xor/add old key with new key
        paddb  xmm1,[rdi+DQ1]
        paddb  xmm2,[rdi+DQ2]
        paddb  xmm3,[rdi+DQ3]
        paddb  xmm4,[rdi+DQ4]
        pxor   xmm5,[rdi+DQ5]
        pxor   xmm6,[rdi+DQ6]
        pxor   xmm7,[rdi+DQ7]
        pxor   xmm0,[rdi+DQ8]
;;Modify Key with rotation of dq words
        movdqa  [rdi+DQ1], xmm1
        movdqa  [rdi+DQ2], xmm2
        movdqa  [rdi+DQ3], xmm3
        movdqa  [rdi+DQ4], xmm4
        movdqa  [rdi+DQ5], xmm5
        movdqa  [rdi+DQ6], xmm6
        movdqa  [rdi+DQ7], xmm7
        movdqa  [rdi+DQ8], xmm0
;;prepare output
        ;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
```

```
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
        ;shift words by primes
        psllw   xmm0, 3
        psllw   xmm1, 5
        psllw   xmm2, 7
        psllw   xmm3, 11
        ;
        psrlw   xmm8, 5
        psrlw   xmm9, 7
        psrlw   xmm10, 11
        psrlw   xmm11, 3
        ;
        psllw   xmm4, 7
        psllw   xmm5, 11
        psllw   xmm6, 3
        psllw   xmm7, 5
        ;
        psrlw   xmm12, 11
        psrlw   xmm13, 3
        psrlw   xmm14, 5
        psrlw   xmm15, 7
        ; add / xor
        pxor    xmm0, xmm8
        pxor    xmm1, xmm9
        pxor    xmm2, xmm10
        pxor    xmm3, xmm11
        paddb   xmm4, xmm12
        paddb   xmm5, xmm13
        paddb   xmm6, xmm14
        paddb   xmm7, xmm15
        pxor    xmm0, xmm4
        pxor    xmm1, xmm5
        pxor    xmm2, xmm6
        paddb   xmm3, xmm7
;;;xor random 64bytes with block
        pxor    xmm0,dqword[rsi+DQ1]
        pxor    xmm1,dqword[rsi+DQ2]
        pxor    xmm2,dqword[rsi+DQ3]
        pxor    xmm3,dqword[rsi+DQ4]
;;;modify block
        movdqa  dqword[rsi+DQ1],xmm0
        movdqa  dqword[rsi+DQ2],xmm1
        movdqa  dqword[rsi+DQ3],xmm2
        movdqa  dqword[rsi+DQ4],xmm3
;;;
        dec     rbx
        jnz     .encryptBlock2
;;;buffer bit shuffling (ROLling) using values from the key
       ; movdqa  xmm0,[rsi+DQ1]
       ; movdqa  xmm1,[rsi+DQ2]
       ; movdqa  xmm2,[rsi+DQ3]
       ; movdqa  xmm3,[rsi+DQ4]
        movdqa  xmm4,xmm0
```

```
        movdqa  xmm5,xmm1
        movdqa  xmm6,xmm2
        movdqa  xmm7,xmm3
        movq    xmm8,[rcx+r8*8]
        movq    xmm9,[rcx+r8*8+ROLLLUT_SIZE] ;;;+SizeOfLUT
        psrlq   xmm0,xmm8
        psrlq   xmm1,xmm8
        psrlq   xmm2,xmm8
        psrlq   xmm3,xmm8
        psllq   xmm4,xmm9
        psllq   xmm5,xmm9
        psllq   xmm6,xmm9
        psllq   xmm7,xmm9
        por     xmm0,xmm4
        por     xmm1,xmm5
        por     xmm2,xmm6
        por     xmm3,xmm7
        movdqa  [rsi+DQ1],xmm0
        movdqa  [rsi+DQ2],xmm1
        movdqa  [rsi+DQ3],xmm2
        movdqa  [rsi+DQ4],xmm3
;;;
        sub     rdx,64
        jns     .encryptBlock
        mov     rbx,[rsp+8*10]
        mov     rsi,[rsp+8*11]
        mov     rdi,[rsp+8*12]
        mov     r12,[rsp+8*13]
        mov     r13,[rsp+8*14]
        add     rsp,8*15
        ret     0

AMDPad16
;;;PARAMETERS   RCX: buffer address, 16byte aligned
;;;            RDX: buffer length in bytes, multiple of 64
;;;            R8 : key address, 16 byte aligned, 128 bytes in size
;;;            R9 : number of passes, greater than or equal to 1, recommented 2
PROEDecryptBuffer:
        sub     rsp,8*15
        mov     [rsp+8*10],rbx
        mov     [rsp+8*11],rsi
        mov     [rsp+8*12],rdi
        mov     [rsp+8*13],r12
        mov     rdi,r8
        mov     r12,r9
;;;hash and encrypt the block of data
        lea     rsi,[rcx+rdx]
        sub     rdx,64
        xor     eax,eax
        lea     rcx,[RollLUT]
AMDPad16
  .decryptBlock:
        movzx   r8,byte[rdi+rax]
        mov     rbx,r12
        add     rax,r8
        sub     rsi,64
```

```
	and	eax,127
;;;buffer bit shuffling (ROLling) using values from the key
	movdqa	xmm0,[rsi+DQ1]
	movdqa	xmm1,[rsi+DQ2]
	movdqa	xmm2,[rsi+DQ3]
	movdqa	xmm3,[rsi+DQ4]
	movdqa	xmm4,xmm0
	movdqa	xmm5,xmm1
	movdqa	xmm6,xmm2
	movdqa	xmm7,xmm3
	movq	xmm8,[rcx+r8*8]
	movq	xmm9,[rcx+r8*8+ROLLLUT_SIZE] ;;;+SizeOfLUT
	psllq	xmm0,xmm8
	psllq	xmm1,xmm8
	psllq	xmm2,xmm8
	psllq	xmm3,xmm8
	psrlq	xmm4,xmm9
	psrlq	xmm5,xmm9
	psrlq	xmm6,xmm9
	psrlq	xmm7,xmm9
	por	xmm0,xmm4
	por	xmm1,xmm5
	por	xmm2,xmm6
	por	xmm3,xmm7
	movdqa	[rsi+DQ1],xmm0
	movdqa	[rsi+DQ2],xmm1
	movdqa	[rsi+DQ3],xmm2
	movdqa	[rsi+DQ4],xmm3
AMDPad16
  .decryptBlock2:
;;setup
	movdqa	xmm0, [rdi+DQ1]
	movdqa	xmm1, [rdi+DQ2]
	movdqa	xmm2, [rdi+DQ3]
	movdqa	xmm3, [rdi+DQ4]
	movdqa	xmm4, [rdi+DQ5]
	movdqa	xmm5, [rdi+DQ6]
	movdqa	xmm6, [rdi+DQ7]
	movdqa	xmm7, [rdi+DQ8]
;;copy
	movdqa	xmm8, xmm0
	movdqa	xmm9, xmm1
	movdqa	xmm10, xmm2
	movdqa	xmm11, xmm3
	movdqa	xmm12, xmm4
	movdqa	xmm13, xmm5
	movdqa	xmm14, xmm6
	movdqa	xmm15, xmm7
;;shift right logical 63bits to have the mask of highest bit
	psrlq	xmm8, 63
	psrlq	xmm9, 63
	psrlq	xmm10, 63
	psrlq	xmm11, 63
	psrlq	xmm12, 63
	psrlq	xmm13, 63
	psrlq	xmm14, 63
```

```
        psrlq   xmm15, 63
;;shift left to remove the highest bit and empty the lowest
        psllq   xmm4, 1
        psllq   xmm5, 1
        psllq   xmm6, 1
        psllq   xmm7, 1
;;add masked bit
        paddq   xmm0, xmm12
        paddq   xmm1, xmm13
        paddq   xmm2, xmm14
        paddq   xmm3, xmm15
;;logical or lowest bit
        por     xmm4, xmm8
        por     xmm5, xmm9
        por     xmm6, xmm10
        por     xmm7, xmm11
;;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
;;Bit ROLL by prime numbers 7, 5, 3, 11
        psllq   xmm0, 7
        psllq   xmm1, 5
        psllq   xmm2, 3
        psllq   xmm3, 11
        psrlq   xmm8, 57;64-7
        psrlq   xmm9, 59;64-5
        psrlq   xmm10, 61;64-3
        psrlq   xmm11, 53;64-11
        por     xmm0, xmm8
        por     xmm1, xmm9
        por     xmm2, xmm10
        por     xmm3, xmm11
;;Dword order switching
        pshufd  xmm0, xmm0, 00111001b
        pshufd  xmm1, xmm1, 00111001b
        pshufd  xmm2, xmm2, 00111001b
        pshufd  xmm3, xmm3, 00111001b
;;xor/add old key with new key
        paddb  xmm1,[rdi+DQ1]
        paddb  xmm2,[rdi+DQ2]
        paddb  xmm3,[rdi+DQ3]
        paddb  xmm4,[rdi+DQ4]
        pxor   xmm5,[rdi+DQ5]
        pxor   xmm6,[rdi+DQ6]
        pxor   xmm7,[rdi+DQ7]
        pxor   xmm0,[rdi+DQ8]
;;Modify Key with rotation of dq words
        movdqa  [rdi+DQ1], xmm1
        movdqa  [rdi+DQ2], xmm2
        movdqa  [rdi+DQ3], xmm3
        movdqa  [rdi+DQ4], xmm4
        movdqa  [rdi+DQ5], xmm5
        movdqa  [rdi+DQ6], xmm6
        movdqa  [rdi+DQ7], xmm7
```

```
        movdqa  [rdi+DQ8], xmm0
;;prepare output
        ;copy
        movdqa  xmm8, xmm0
        movdqa  xmm9, xmm1
        movdqa  xmm10, xmm2
        movdqa  xmm11, xmm3
        movdqa  xmm12, xmm4
        movdqa  xmm13, xmm5
        movdqa  xmm14, xmm6
        movdqa  xmm15, xmm7
        ;shift words by primes
        psllw   xmm0, 3
        psllw   xmm1, 5
        psllw   xmm2, 7
        psllw   xmm3, 11
        ;
        psrlw   xmm8, 5
        psrlw   xmm9, 7
        psrlw   xmm10, 11
        psrlw   xmm11, 3
        ;
        psllw   xmm4, 7
        psllw   xmm5, 11
        psllw   xmm6, 3
        psllw   xmm7, 5
        ;
        psrlw   xmm12, 11
        psrlw   xmm13, 3
        psrlw   xmm14, 5
        psrlw   xmm15, 7
        ; add / xor
        pxor    xmm0, xmm8
        pxor    xmm1, xmm9
        pxor    xmm2, xmm10
        pxor    xmm3, xmm11
        paddb   xmm4, xmm12
        paddb   xmm5, xmm13
        paddb   xmm6, xmm14
        paddb   xmm7, xmm15
        pxor    xmm0, xmm4
        pxor    xmm1, xmm5
        pxor    xmm2, xmm6
        paddb   xmm3, xmm7
;;;xor random 64bytes with block
        pxor    xmm0,dqword[rsi+DQ1]
        pxor    xmm1,dqword[rsi+DQ2]
        pxor    xmm2,dqword[rsi+DQ3]
        pxor    xmm3,dqword[rsi+DQ4]
;;;modify block
        movdqa  dqword[rsi+DQ1],xmm0
        movdqa  dqword[rsi+DQ2],xmm1
        movdqa  dqword[rsi+DQ3],xmm2
        movdqa  dqword[rsi+DQ4],xmm3
;;;
        dec     rbx
```

```
      jnz    .decryptBlock2
;;;
      sub    rdx,64
      jns    .decryptBlock
      mov    rbx,[rsp+8*10]
      mov    rsi,[rsp+8*11]
      mov    rdi,[rsp+8*12]
      mov    r12,[rsp+8*13]
      add    rsp,8*15
      ret    0


section '.idata' import data readable writeable
;;;API imports
  library user32,'USER32.DLL'

      import user32,\
            MessageBox,'MessageBoxA'

section '.edata' export data readable

  export 'PROE_Lib_Win64.DLL',\
       PROEEncryptBuffer,'PROEEncryptBuffer',\
       PROEEncryptBufferWithChecksum,'PROEEncryptBufferWithChecksum',\
       PROEDecryptBuffer,'PROEDecryptBuffer',\
       PROEDecryptBufferWithChecksum,'PROEDecryptBufferWithChecksum',\
       PROEValidateParameters,'PROEValidateParameters',\
       PROESelfTest,'PROESelfTest'

section '.reloc' fixups data discardable
```

Appendix B: Lib_Test source code

```
;;;LOUIS RICCI
;;;PROE TESTING PROGRAM
;;;LibTest.exe
format PE64 Console
entry start

include '%fasminc%\win64a.inc'

COUNTER         equ 171800 ;;;;FOR RANDOM OUTPUT FILE

section '.data' data readable writeable

align 16
PRNG_Seed       dq 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
align 16
PRNG_Buffer     dq 0,0,0,0,0,0,0,0
file_name       db 'proe'
SUFF            db 'A'
                db '.rng',0
hfile           dq 0
_fmtbi          db ' %d Bytes : %d Iterations ',13,10,0
_fmtc           db ' %f CPU Cycles ',13,10,0
_fmthz          db ' %f hz ',13,10,0
_fmtcb          db ' %f Cycles/Byte ',13,10,0
_fmtbs          db ' %f Bytes/Second ',13,10,0
_fmtb           db ' %f bytes ',13,10,0
_fmts           db ' %f seconds ',13,10,0
_fmtbyte        db '%d ',0
_fmtnl          db 13,10,0

align 16
CYCLES_SECOND   dq 2020000000.0 ;2.02Ghz
TEST_SIZE       dq 8192;640000000  ;# of bytes to encrypt / iteration
TEST_TMP        dq 0
CYCLES          dq 0
ITERATIONS      dq 40960  ;# iterations
;TESTTYPE       dq OUTPUT;TESTING;
TESTTYPE        dq TESTING;OUTPUT

section '.code' code readable executable

start:
        sub     rsp,8*8
        call    [PROESelfTest]
        mov     rax,TESTING
        jmp     QWORD[TESTTYPE]
TESTING:
        rdtsc
        mov     r8,rax
        mov     rcx,124
        mov     rdx,PRNG_Seed
  .fillk:
        mov     dword[rdx+rcx],r8d
        push    rdx
```

```
      rdtsc
      pop     rdx
      rol     eax,7
      xor     r8d,eax
      sub     rcx,4
      jns     .fillk
      mov     rcx,qword[TEST_SIZE]
      call    AllocateOrDie
      mov     [hfile],rax
;;;GET CPU HZ
      rdtsc
      mov     ebx,eax
      mov     ebp,edx
      mov     rdx,0
      mov     rcx,1000
      call    [SleepEx]
      rdtsc
      sub     eax,ebx
      sbb     edx,ebp
      shl     rdx,32
      or      rax,rdx
      cvtsi2sd xmm0,rax
      movq    qword[CYCLES_SECOND],xmm0
;;;;;;;CYCLES / BYTE of encryption
      mov     rcx,0
      mov     rdx,0
      call    [SleepEx]
      mov     r15,[ITERATIONS]
      mov     rcx,qword[TEST_SIZE]
      call    AllocateOrDie
      mov     [hfile],rax
      mov     r14,rax
      mov     r13,qword[TEST_SIZE]
      mov     r12,PRNG_Seed
      mov     r11, 8
      rdtsc
      mov     rbx,rax
      mov     rbp,rdx
  .iterate:
      mov     rcx,[hfile]   ;;;;;BUFFER ADDRESS
      mov     rdx,qword[TEST_SIZE] ;;;;;SIZE OF BUFFER IN BYTES
      mov     r8,PRNG_Seed  ;;;;;SEED ADDRESS
      mov     r9,8          ;;;;;NUMBER OF PASSES
      call    [PROEEncryptBufferWithChecksum]
    ;  call   [PROEEncryptBuffer]
      dec     r15
      jnz     .iterate
      rdtsc
      sub     eax,ebx
      sbb     edx,ebp
      mov     ebx,eax
      mov     ebp,edx
      shl     rdx,32
      or      rax,rdx
      cvtsi2sd xmm0,rax ;;;clock cycles total
      movq    qword[CYCLES],xmm0
```

```
    movsd    xmm1,[CYCLES_SECOND]
    movq     xmm2,xmm0
    divsd    xmm2,xmm1 ;;total seconds
    movq     qword[TEST_TMP],xmm2
    mov      r15,qword[TEST_TMP]
    mov      rax,qword[TEST_SIZE]
    cvtsi2sd xmm3,rax
    mov      rax,qword[ITERATIONS]
    cvtsi2sd xmm4,rax
    mulsd    xmm3,xmm4 ;;;TOTAL BYTES
    movq    qword[TEST_TMP],xmm3
    mov     r14,qword[TEST_TMP]
    divsd   xmm3,xmm2 ;;; BYTES / SECOND
    movq    qword[TEST_TMP],xmm3
    mov     r13,qword[TEST_TMP]
    divsd   xmm1,xmm3 ;;; CLOCKS / BYTE
    movq    qword[TEST_TMP],xmm1
    mov     r12,qword[TEST_TMP]

    mov     rdx,qword[CYCLES]
    mov     rcx,_fmtc
    call    [printf] ;;;TOTAL CYCLES
    mov     rdx,qword[CYCLES_SECOND]
    mov     rcx,_fmthz
    call    [printf] ;;;CPU hz
    mov     rdx,r15
    mov     rcx,_fmts
    call    [printf] ;;;TOTAL TIME (seconds)
    mov     rdx,r14
    mov     rcx,_fmtb
    call    [printf] ;;;TOTAL BYTES
    mov     rdx,r13
    mov     rcx,_fmtbs
    call    [printf] ;;; BYTES / SECOND
    mov     rdx,r12
    mov     rcx,_fmtcb
    call    [printf] ;;; CLOCKS / BYTE
    mov     r8,qword[ITERATIONS]
    mov     rdx,qword[TEST_SIZE]
    mov     rcx,_fmtbi
    call    [printf]

    mov     rcx,[hfile]
    call    Deallocate

    call    [PROESelfTest]
    xor     ecx,ecx
    call    [ExitProcess]


;;;;;;;OUTPUT RANDOM FILE FOR RANDOMNESS TESTING
OUTPUT:
    mov     rbp,5
 .main:
    rdtsc
    mov     r8,rax
```

```
    mov     rcx,124
    mov     rdx,PRNG_Seed
.fillkey:
    mov     dword[rdx+rcx],r8d
    push    rdx
    rdtsc
    pop     rdx
    add     r8d,eax
    rol     eax,7
    ror     r8d,3
    xor     r8d,eax
    sub     rcx,4
    jns     .fillkey
    mov     rcx,file_name
    mov     rdx,GENERIC_WRITE
    mov     r8,0
    xor     r9d,r9d
    mov     qword[rsp+4*8],CREATE_ALWAYS
    mov     qword[rsp+5*8],0
    mov     qword[rsp+6*8],0
    call    [CreateFile]
    mov     [hfile],rax
    mov     rbx,COUNTER
.loop:
    mov     rcx,PRNG_Buffer
    mov     rdx,PRNG_Seed
    call    PROE_PRNG
    mov     rcx,[hfile]
    mov     rdx,PRNG_Buffer
    mov     r8,64
    lea     r9,[rsp+8*5]
    mov     qword[rsp+8*4],0
    call    [WriteFile]
    cmp     rbx,1
    je      .print
    cmp     rbx,2
    je      .print
    jmp     .skipprint
 .print:
    call    PrintEveryEigth
 .skipprint:
    dec     rbx
    jnz     .loop
    mov     rcx,[hfile]
    call    [CloseHandle]
    mov     rcx,rbp
    add     cl,'A'
    mov     byte[SUFF],cl
    rdtsc
    movzx   rcx,al
    xor     edx,edx
    call    [SleepEx]
    dec     rbp
    jns     .main
    call    [PROESelfTest]
    mov     rcx,0
```

```
    call   [ExitProcess]

PrintEveryEigth: ;;;prints every eighth byte to the console
    sub rsp,8*8
    mov r14,rsp
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer]
    call [printf]
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer+8]
    call [printf]
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer+16]
    call [printf]
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer+24]
    call [printf]
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer+32]
    call [printf]
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer+40]
    call [printf]
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer+48]
    call [printf]
    mov rcx,_fmtbyte
    movzx rdx,byte[PRNG_Buffer+56]
    call [printf]
    mov rcx,_fmtnl
    call [printf]
    mov rsp,r14
    add rsp,8*8
    ret 0

DQ1    equ 0
DQ2    equ 16
DQ3    equ 32
DQ4    equ 48
DQ5    equ 64
DQ6    equ 80
DQ7    equ 96
DQ8    equ 112
;;;FOR TESTING AND DEVELOPEMENT
;;;STRIPPED DOWN VERSION OF THE PRNG FOR CREATING RANDOM OUTPUT FILES
align 16
PROE_PRNG:
    sub    rsp,8*7
;;setup
    movdqa  xmm0, [rdx+DQ1]
    movdqa  xmm1, [rdx+DQ2]
    movdqa  xmm2, [rdx+DQ3]
    movdqa  xmm3, [rdx+DQ4]
    movdqa  xmm4, [rdx+DQ5]
    movdqa  xmm5, [rdx+DQ6]
    movdqa  xmm6, [rdx+DQ7]
```

```
      movdqa  xmm7, [rdx+DQ8]
;;copy
      movdqa  xmm8, xmm0
      movdqa  xmm9, xmm1
      movdqa  xmm10, xmm2
      movdqa  xmm11, xmm3
      movdqa  xmm12, xmm4
      movdqa  xmm13, xmm5
      movdqa  xmm14, xmm6
      movdqa  xmm15, xmm7
;;shift right logical 63bits to have the highest bit in the lowest bit position
      psrlq   xmm8, 63
      psrlq   xmm9, 63
      psrlq   xmm10, 63
      psrlq   xmm11, 63
      psrlq   xmm12, 63
      psrlq   xmm13, 63
      psrlq   xmm14, 63
      psrlq   xmm15, 63
;;shift left to remove the highest bit and empty the lowest
      psllq   xmm4, 1
      psllq   xmm5, 1
      psllq   xmm6, 1
      psllq   xmm7, 1
;;add masked bit
      paddq   xmm0, xmm12
      paddq   xmm1, xmm13
      paddq   xmm2, xmm14
      paddq   xmm3, xmm15
;;logical or lowest bit
      por     xmm4, xmm8
      por     xmm5, xmm9
      por     xmm6, xmm10
      por     xmm7, xmm11
;;copy
      movdqa  xmm8, xmm0
      movdqa  xmm9, xmm1
      movdqa  xmm10, xmm2
      movdqa  xmm11, xmm3
;;Bit ROLL by prime numbers 7, 5, 3, 11
      psllq   xmm0, 7
      psllq   xmm1, 5
      psllq   xmm2, 3
      psllq   xmm3, 11
      psrlq   xmm8, 57;64-7
      psrlq   xmm9, 59;64-5
      psrlq   xmm10, 61;64-3
      psrlq   xmm11, 53;64-11
      por     xmm0, xmm8
      por     xmm1, xmm9
      por     xmm2, xmm10
      por     xmm3, xmm11
;;Dword rotate
      pshufd  xmm0, xmm0, 00111001b
      pshufd  xmm1, xmm1, 00111001b
      pshufd  xmm2, xmm2, 00111001b
```

```
      pshufd  xmm3, xmm3, 00111001b
;;xor/add old key with new key
      paddb  xmm1,[rdx+DQ1]
      paddb  xmm2,[rdx+DQ2]
      paddb  xmm3,[rdx+DQ3]
      paddb  xmm4,[rdx+DQ4]
      pxor   xmm5,[rdx+DQ5]
      pxor   xmm6,[rdx+DQ6]
      pxor   xmm7,[rdx+DQ7]
      pxor   xmm0,[rdx+DQ8]
;;Modify Key with rotation of dq words
      movdqa  [rdx+DQ1], xmm1
      movdqa  [rdx+DQ2], xmm2
      movdqa  [rdx+DQ3], xmm3
      movdqa  [rdx+DQ4], xmm4
      movdqa  [rdx+DQ5], xmm5
      movdqa  [rdx+DQ6], xmm6
      movdqa  [rdx+DQ7], xmm7
      movdqa  [rdx+DQ8], xmm0
;;prepare output
      ;copy
      movdqa  xmm8, xmm0
      movdqa  xmm9, xmm1
      movdqa  xmm10, xmm2
      movdqa  xmm11, xmm3
      movdqa  xmm12, xmm4
      movdqa  xmm13, xmm5
      movdqa  xmm14, xmm6
      movdqa  xmm15, xmm7
      ;shift words by primes
      psllw   xmm0, 3
      psllw   xmm1, 5
      psllw   xmm2, 7
      psllw   xmm3, 11
      ;
      psrlw   xmm8, 5
      psrlw   xmm9, 7
      psrlw   xmm10, 11
      psrlw   xmm11, 3
      ;
      psllw   xmm4, 7
      psllw   xmm5, 11
      psllw   xmm6, 3
      psllw   xmm7, 5
      ;
      psrlw   xmm12, 11
      psrlw   xmm13, 3
      psrlw   xmm14, 5
      psrlw   xmm15, 7
      ; add / xor
      pxor    xmm0, xmm8
      pxor    xmm1, xmm9
      pxor    xmm2, xmm10
      pxor    xmm3, xmm11
      paddb   xmm4, xmm12
      paddb   xmm5, xmm13
```

```
        paddb   xmm6, xmm14
        paddb   xmm7, xmm15
        pxor    xmm0, xmm4
        pxor    xmm1, xmm5
        pxor    xmm2, xmm6
        paddb   xmm3, xmm7
;;output random bytes
        movdqa  dqword[rcx+DQ1],xmm0
        movdqa  dqword[rcx+DQ2],xmm1
        movdqa  dqword[rcx+DQ3],xmm2
        movdqa  dqword[rcx+DQ4],xmm3
        add     rsp,8*7
        ret     0

;;;input: rcx = number of bytes to allocate
;;;return: address of memory if successful OR exit program with error message
AllocateOrDie:
        sub     rsp,8*5
        mov     rdx,rcx
        mov     r8d,MEM_COMMIT or MEM_RESERVE
        xor     rcx,rcx
        mov     r9d,PAGE_READWRITE
        call    [VirtualAlloc]
        test    rax,rax
        jz      .error
        add     rsp,8*5
        ret     0
  .error:
        xor     ecx,ecx
        xor     edx,edx
        mov     r8,rcx
        mov     r9,rcx
        call    [MessageBox]
        xor     ecx,ecx
        call    [ExitProcess]

;;;input: address of memory that needs to be freed
Deallocate:
        sub     rsp,8*5
        xor     edx,edx
        mov     r8d,MEM_RELEASE
        call    [VirtualFree]
        add     rsp,8*5
        ret     0

section '.idata' import data readable writeable
;;;API imports
  library kernel32,'KERNEL32.DLL',\
          msvcrt,'MSVCRT.DLL',\
          comdlg32,'COMDLG32.DLL',\
          shell32,'SHELL32.DLL',\
          ole32,'OLE32.DLL',\
          user32,'USER32.DLL',\
          PROELib,'PROE_Lib_Win64.DLL'

        import PROELib,\
```

```
        PROESelfTest,'PROESelfTest',\
        PROEValidateParameters,'PROEValidateParameters',\
        PROEEncryptBuffer,'PROEEncryptBuffer',\
        PROEDecryptBuffer,'PROEDecryptBuffer',\
        PROEEncryptBufferWithChecksum,'PROEEncryptBufferWithChecksum',\
        PROEDecryptBufferWithChecksum,'PROEDecryptBufferWithChecksum'

    import comdlg32,\
        GetOpenFileName,'GetOpenFileNameA',\
        GetSaveFileName,'GetSaveFileNameA'
    import shell32,\
        SHBrowseForFolder,'SHBrowseForFolderA',\
        SHGetPathFromIDList,'SHGetPathFromIDListA'
    import ole32,\
        OleInitialize,'OleInitialize',\
        CoTaskMemFree,'CoTaskMemFree'
;;;DEBUGGING function(s)
    import msvcrt,\
        printf,'printf'
  include '%fasminc%\apia\kernel32.inc'
  include '%fasminc%\apia\user32.inc'
```

Appendix C: PROE File Encryption source code

```
;;;LOUIS RICCI
;;;PROE Dialog GUI Main
;;;PROE Encrypt.exe
format PE64 GUI 4.0
entry start

include '%fasminc%\win64a.inc'
include 'DialogPROEEqu.asm'
include 'PROEEncEqu.asm'
include 'PROEGUIEqu.asm'


section '.data' data readable writeable
include 'UtilData.asm'
include 'PROEEncData.asm'
include 'PROEGUIData.asm'

section '.code' code readable executable

  start:
      sub     rsp,8*7
;;;
      xor     ecx,ecx
      call    [GetModuleHandle]
      mov     qword[handle],rax
;;;
      mov     qword [rsp+8*4],0
      lea     r9,[DialogProc]
      mov     r8d,HWND_DESKTOP
      mov     edx,ID_DIALOG
      mov     rcx,rax
      call    [DialogBoxParam]
      or      rax,rax
      jz      exit
  exit:
      xor     ecx,ecx
      call    [ExitProcess]
;;;input: rcx=handle rdx=msg r8=wParm r9=lParm
DialogProc:
      mov     rbx,rcx ;;; save for use later in the dialog proc and in "processed" functions
      push    rcx rdx r8 r9
      ;;;rsp+8* 8  7  6  5
      sub     rsp,8*5
      cmp     rdx,WM_INITDIALOG
      jz      wminitdialog
      cmp     rdx,WM_COMMAND
      jz      wmcommand
      cmp     rdx,WM_CLOSE
      jz      wmclose
      xor     eax,eax
      jmp     finish
  wminitdialog:
;;;general setup
      call    CountProcessors
```

```
        mov     rcx,FILE_BUFFER_SIZE
        call    AllocateOrDie
        mov     qword[path_buffer],rax
        mov     rcx,FILE_BUFFER_SIZE
        call    AllocateOrDie
        mov     qword[name_buffer],rax
        mov     rcx,FILE_BUFFER_SIZE
        call    AllocateOrDie
        mov     qword[lv_buffer],rax
        mov     rcx,FILE_BUFFER_SIZE
        call    AllocateOrDie
        mov     qword[browse_buffer],rax
;;;key saving setup
        xor     ecx,ecx
        call    [OleInitialize]
;;;dialog controls, handle grabbing
        mov     edx,ID_KEYOUT
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     [keyouthandle],rax
        mov     edx,ID_PASSES
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     [passeshandle],rax
        mov     edx,ID_DELETE
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     [deletehandle],rax
        mov     edx,ID_STATUS
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     [statushandle],rax
        mov     edx,ID_ETIME
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     [etimehandle],rax
;;;
        mov     r8,1
        mov     rdx,EM_SETLIMITTEXT
        mov     rcx,[passeshandle]
        call    [SendMessage]
        mov     word[rsp+8*4],0032h ;;;String '2',0
        xor     r8d,r8d
        lea     r9,[rsp+8*4]
        mov     rdx,WM_SETTEXT
        mov     rcx,[passeshandle]
        call    [SendMessage]
        mov     r8,BST_CHECKED
        mov     rdx,BM_SETCHECK
        mov     rcx,[deletehandle]
        call    [SendMessage]
;;;setup the list view control
        mov     edx,ID_INPUT
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[lvhandle],rax
```

```
        mov     rcx,rax
        mov     edx,LVM_INSERTCOLUMN
        mov     r8d,1
        mov     r9,LVCOLUMN_1
        mov     rax,LV_HEADER1
        mov     qword[LVCOLUMNpszText],rax
        call    [SendMessage]
        mov     rcx,qword[lvhandle]
        mov     edx,LVM_INSERTCOLUMN
        mov     r8d,2
        mov     r9,LVCOLUMN_1
        mov     rax,LV_HEADER2
        mov     qword[LVCOLUMNpszText],rax
        call    [SendMessage]
        mov     edx,ID_COUNT
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[counthandle],rax
        mov     edx,ID_ADD
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[addhandle],rax
        mov     edx,ID_REMOVE
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[removehandle],rax
        mov     edx,ID_CLEAR
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[clearhandle],rax
        mov     edx,ID_MAKEKEY
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[makekeyhandle],rax
        jmp     processed
  wmcommand:
;;;GUI INPUT PROCESSING
        cmp     r8d,BN_CLICKED shl 16 + ID_ADD
        jz      GetFiles
        cmp     r8d,BN_CLICKED shl 16 + ID_CLEAR
        jz      ClearListView
        cmp     r8d,BN_CLICKED shl 16 + ID_REMOVE
        jz      RemoveItem
        cmp     r8d,BN_CLICKED shl 16 + ID_MAKEKEY
        jz      CreateKeys
        cmp     r8d,BN_CLICKED shl 16 + ID_KEYDIR
        jz      KeySaveDir
        cmp     r8d,BN_CLICKED shl 16 + IDCANCEL
        jz      wmclose
        cmp     r8d,BN_CLICKED shl 16 + IDOK
        jz      StartEncryption
        jmp     processed
  wmclose:
        xor     edx,edx
        mov     rcx,[rsp + 8*8]
        call    [EndDialog]
```

```
  processed:
        mov     eax,1
  finish:
        add     rsp,8*5
        pop     r9 r8 rdx rcx
        ret

include 'PROEGUICode.asm'
include 'PROEEncCode.asm'
include 'UtilCode.asm'

section '.idata' import data readable writeable
;;;API imports
  library kernel32,'KERNEL32.DLL',\
          msvcrt,'MSVCRT.DLL',\
          comdlg32,'COMDLG32.DLL',\
          shell32,'SHELL32.DLL',\
          ole32,'OLE32.DLL',\
          user32,'USER32.DLL',\
          MouseHook,'MouseHook.DLL',\
          PROELib,'PROE_Lib_Win64.DLL'

        import PROELib,\
              PROESelfTest,'PROESelfTest',\
              PROEValidateParameters,'PROEValidateParameters',\
              PROEEncryptBuffer,'PROEEncryptBuffer',\
              PROEDecryptBuffer,'PROEDecryptBuffer',\
              PROEEncryptBufferWithChecksum,'PROEEncryptBufferWithChecksum',\
              PROEDecryptBufferWithChecksum,'PROEDecryptBufferWithChecksum'

        import MouseHook,\
              SharedAddr,'SharedAddr',\ ;;;Not a function dll shared data ptr
              CallbackMouseHook,'CallbackMouseHook'
        import comdlg32,\
              GetOpenFileName,'GetOpenFileNameA',\
              GetSaveFileName,'GetSaveFileNameA'
        import shell32,\
              SHBrowseForFolder,'SHBrowseForFolderA',\
              SHGetPathFromIDList,'SHGetPathFromIDListA'
        import ole32,\
              OleInitialize,'OleInitialize',\
              CoTaskMemFree,'CoTaskMemFree'
;;;DEBUGGING function(s)
        import msvcrt,\
              printf,'printf'

  include '%fasminc%\apia\kernel32.inc'
  include '%fasminc%\apia\user32.inc'

section '.rsrc' resource data readable
include 'DialogPROERes.asm'

;;;PROE Encyrption Code

;;;input: rcx = file size
;;;return: buffer size to use
```

```
FileSizeToBufferSize:
      add     rcx,64
      lea     rax,[FS2BS_LUT]
      bsr     rcx,rcx
      mov     rax,qword[rax+rcx*8]
      ret     0

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ThreadWorker:
      sub     rsp,8*9
      ;;;[rsp+8*7] ;;;hold for FileSizeToBufferSize value
  .loop:
;;;*********
      call    StackPop
      test    rax,rax
      jz      .finish
;;;store info in registers
      mov     rbp,rax
      mov     rdi,[rax+WORKER.inFile]
      mov     rbx,[rax+WORKER.inSize]
      mov     r12,[rax+WORKER.outFile]
      lea     r13,[rax+WORKER.encKey]
      lock add qword[bytecount],rbx
      movzx   r14,byte[rax+WORKER.encFlags]
      mov     rcx,rbx
      call    FileSizeToBufferSize
      mov     qword[rsp+8*7],rax
      mov     r15,rax
      mov     rcx,rax
      call    AllocateOrDie
      mov     rsi,rax
;;;calc size and ReadFile
  .readInFile:
      test    rbx,rbx
      jz      .writeKeyFile
      cmp     rbx,r15
      jae     .else
  .then:
      mov     r8,rbx
      lea     r15,[rbx+64]
      and     r15,0FFFFFFFFFFFFFFC0h ;;;Ceiling multiple of 64
;;;zero a portion of the buffer so old bytes don't corrupt the hashing
      lea     rcx,[r15-64]
      pxor    xmm0,xmm0
      movdqa  dqword[rsi+rcx+DQ1],xmm0
      movdqa  dqword[rsi+rcx+DQ2],xmm0
      movdqa  dqword[rsi+rcx+DQ3],xmm0
      movdqa  dqword[rsi+rcx+DQ4],xmm0
      xor     ebx,ebx
      jmp     .endIf
  .else:
      mov     r8,r15
      sub     rbx,r15
  .endIf:
      mov     rcx,rdi
      mov     rdx,rsi
```

```
        lea     r9,[rsp+8*5]
        mov     qword[rsp+8*4],0
        call    [ReadFile]
        test    rax,rax
        jz      .error
;;;hash and encrypt the block of data
        mov     rcx,rsi
        mov     rdx,r15
        mov     r8,r13
        mov     r9,r14
        call    [PROEEncryptBufferWithChecksum]
  .writeOutFile:
        mov     rcx,r12
        mov     rdx,rsi
        mov     r8,r15
        lea     r9,[rsp+8*5]
        mov     qword[rsp+8*4],0
        call    [WriteFile]
        test    rax,rax
        jz      .error
        jmp     .readInFile
  .writeKeyFile:
        mov     rcx,[rbp+WORKER.outKey]
        mov     rdx,rbp
        mov     r8,WORKER_SIZE
        lea     r9,[rsp+8*5]
        mov     qword[rsp+8*4],0
        call    [WriteFile]
        test    rax,rax
        jz      .error
;;;clean up memory and file alloations
        mov     rcx,r12  ;;;outFile handle
        call    [CloseHandle]
        mov     rcx,[rbp+WORKER.outKey]
        call    [CloseHandle]
        cmp     byte[rbp+WORKER.encFlags+1],1 ;;;delete?
        je      .deleteFile
  .afterDeleteFile:
        mov     rcx,rdi  ;;;inFile handle
        call    [CloseHandle]
        mov     rcx,rsi
        call    Deallocate
        mov     rcx,rbp
        call    Deallocate
        jmp     .loop
  .error:
        lock add dword[ErrorCount],1
        mov     rcx,ENCPATH_SIZE
        call    AllocateOrDie
        mov     rcx,rax
        mov     rbx,rax
        lea     rdx,[rbp+WORKER.encPath]
        mov     r8,8080808080808080h
  .strCopy:
        mov     rax,[rdx]
        add     rdx,8
```

```
    mov     [rcx],rax
    add     rcx,8
    lea     rax,[rax-0101010101010101h]
    and     rax,r8
    jz      .strCopy
    mov     rcx,rbx
    call    EncryptionError
    mov     rcx,[rbp+WORKER.inFile]
    call    [CloseHandle]
    mov     rcx,[rbp+WORKER.outFile]
    call    [CloseHandle]
    mov     rcx,[rbp+WORKER.outKey]
    call    [CloseHandle]
    mov     rcx,rsi
    call    Deallocate
    mov     rcx,rbp
    call    Deallocate
    jmp     .loop
.finish:
    call    [GetTickCount]
    mov     dword[endtime],eax
    lock add dword[CompletionCount],1
    add     rsp,8*9
    call    [ExitThread]
    ret     0
.deleteFile:
    mov     rcx,rdi
    call    [CloseHandle]
    mov     rbx,[rbp+WORKER.inSize]
    lea     rdi,[rbp+WORKER.encPath]
    xor     rax,rax
.findnull:
    inc     rax
    test    byte[rdi+rax],0FFh
    jnz     .findnull
    mov     dword[rdi+rax-5],0 ;;;null out the .proe extension to return the original file name
    mov     rdx,GENERIC_WRITE
    mov     r8,0 ;;; no sharing
    xor     r9d,r9d
    mov     rax,FILE_FLAG_WRITE_THROUGH Or FILE_FLAG_DELETE_ON_CLOSE
    mov     qword[rsp+4*8],OPEN_EXISTING
    mov     qword[rsp+5*8],rax
    mov     qword[rsp+6*8],0
    mov     rcx,rdi
    mov     r12,rdi
    call    [CreateFile]
    cmp     rax,INVALID_HANDLE_VALUE
    je      .error
    mov     rdi,rax
    mov     r15,qword[rsp+8*7]
.overWrite:
    mov     qword[rsp+8*4],0
    lea     r9,[rsp+8*5]
    mov     r8,r15
    mov     rdx,rsi
    mov     rcx,rdi
```

```
    call   [WriteFile]
    sub    rbx,r15
    jns    .overWrite
    mov    rcx,rdi
    call   [CloseHandle] ;;; make sure the buffer is flushed
    mov    rcx,r12
    call   [DeleteFile] ;;; make sure the file is deleted
    jmp    .afterDeleteFile

;;;INPUT rcx = size to make array stack
;;;returns NONZERO = SUCCESS  0 = FAIL
StackInit:
    shl    rcx,3
    mov    rdx,rcx ;;; size to qword
    xor    ecx,ecx
    mov    qword[rsp+8*4],rdx ;;; save
    mov    r8d,MEM_COMMIT or MEM_RESERVE
    mov    r9d,PAGE_READWRITE
    call   [VirtualAlloc]
    test   rax,rax
    jz     .fail
    mov    qword[stackPtr],rax
    mov    qword[stackHead],rax
    mov    rcx,qword[rsp+8*4]
    add    rax,rcx
    mov    qword[stackEnd],rax
    ret    0
  .fail:
    xor    eax,eax
    ret    0
;;;return NONZERO = SUCCESS  0 = FAIL
StackDelete:
    mov    rcx,qword[stackHead]
    xor    edx,edx
    mov    r8d,MEM_RELEASE
    call   [VirtualFree]
    test   rax,rax
    jz     .fail
    ret    0
  .fail:
    xor    eax,eax
    ret    0

;;;Single Thread PUSH (not thread safe)
;;;RCX = addr of data to put to the stack
;;;returns NONZERO = SUCCESS  0 = FAILURE
StackPush:
    mov    rax,8
    xadd   qword[stackPtr],rax  ;;; LOCKED exchange and add
    cmp    rax,qword[stackEnd]
    jae    .fail
    mov    qword[rax],rcx ;;;store addr of data
    ret    0
  .fail:
    xor    eax,eax
    ret    0
```

```
;;;Concurrent (Thread Safe) POP
;;;return data addr
;;;return if underflow 0 = failure
StackPop:
       mov     rax,-8
   lock xadd    qword[stackPtr],rax ;;; locked exchange and add(sub)
       cmp     rax,qword[stackHead]
       jbe     .fail
       mov     rax,qword[rax-8]  ;;;get addr of data
       ret     0
   .fail:
       xor     eax,eax
       ret     0

;;;input rcx = ptr to error string
EncryptionError:
       sub     rsp,8*7
       mov     r9,rcx
       xor     edx,edx
       lea     r8,[ThreadErrorMsg]
       mov     qword[rsp+8*4],rdx
       mov     qword[rsp+8*5],rdx
       mov     rcx,rdx
       call    [CreateThread]
       mov     rcx,rax
       call    [CloseHandle]
       add     rsp,8*7
       ret     0

;;;Creates an error message box in a seperate thread so encryption can continue
;;;input rcx = message string pointer
ThreadErrorMsg:
       sub     rsp,8*5
       mov     rbx,rcx ;;;save
       mov     r8,rcx
       lea     rdx,[StrEncError]
       xor     ecx,ecx
       mov     r9,rcx
       call    [MessageBox]
       mov     rcx,rbx
       call    Deallocate
       call    [ExitThread]
       add     rsp,8*5
       ret     0

;;;PROE Encryption data

;;;Mouse input data for extra randomness on key creation
ForEncKey dq 0,0

align 16
primes: ;;;first 260
dw      2,     3,     5,     7,    11,    13,    17,    19,    23,    29
dw     31,    37,    41,    43,    47,    53,    59,    61,    67,    71
dw     73,    79,    83,    89,    97,   101,   103,   107,   109,   113
```

```
dw    127,   131,   137,   139,   149,   151,   157,   163,   167,   173
dw    179,   181,   191,   193,   197,   199,   211,   223,   227,   229
dw    233,   239,   241,   251,   257,   263,   269,   271,   277,   281
dw    283,   293,   307,   311,   313,   317,   331,   337,   347,   349
dw    353,   359,   367,   373,   379,   383,   389,   397,   401,   409
dw    419,   421,   431,   433,   439,   443,   449,   457,   461,   463
dw    467,   479,   487,   491,   499,   503,   509,   521,   523,   541
dw    547,   557,   563,   569,   571,   577,   587,   593,   599,   601
dw    607,   613,   617,   619,   631,   641,   643,   647,   653,   659
dw    661,   673,   677,   683,   691,   701,   709,   719,   727,   733
dw    739,   743,   751,   757,   761,   769,   773,   787,   797,   809
dw    811,   821,   823,   827,   829,   839,   853,   857,   859,   863
dw    877,   881,   883,   887,   907,   911,   919,   929,   937,   941
dw    947,   953,   967,   971,   977,   983,   991,   997,  1009,  1013
dw   1019,  1021,  1031,  1033,  1039,  1049,  1051,  1061,  1063,  1069
dw   1087,  1091,  1093,  1097,  1103,  1109,  1117,  1123,  1129,  1151
dw   1153,  1163,  1171,  1181,  1187,  1193,  1201,  1213,  1217,  1223
dw   1229,  1231,  1237,  1249,  1259,  1277,  1279,  1283,  1289,  1291
dw   1297,  1301,  1303,  1307,  1319,  1321,  1327,  1361,  1367,  1373
dw   1381,  1399,  1409,  1423,  1427,  1429,  1433,  1439,  1447,  1451
dw   1453,  1459,  1471,  1481,  1483,  1487,  1489,  1493,  1499,  1511
dw   1523,  1531,  1543,  1549,  1553,  1559,  1567,  1571,  1579,  1583
dw   1597,  1601,  1607,  1609,  1613,  1619,  1621,  1627,  1637,  1657

align 16
;;;File Size To Buffer Size LOOK-UP-TABLE
FS2BS_LUT: ;;;1-4095 bytes (4096)
        dq 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096
        ;;;4K - 16MB (4096 * Log[2](Size)
        dq 4096*2, 4096*4, 4096*8, 4096*16, 4096*32, 4096*64, 4096*128, 4096*256, 4096*512, 4096*1024,
4096*2048, 4096*4096
        ;;; > 16MB (32MB)
        dq 40 dup(4096*8192)
align 16
keyhold rb 128
align 16
keybuffer db 257 dup(0)


;;;concurrent array stack data
;;;Holds WORKER memory structure
align 16
stackPtr     dq 0
stackHead    dq 0
stackEnd     dq 0

;;;Thread Completion and Error Count
CompletionCount dd 0
ErrorCount      dd 0

;;;Error Message
StrEncError db 'Encryption Error',0

;;;PROE Encryption/Decryption Equates

;;;Memory structure used by encyrption/decryption worker thread
```

```
struct WORKER
      encKey   rb 128
      encHash  dq ?
              dq ?
      decKey   rb 128
      decHash  dq ?
              dq ?
      inSize   dq ?
      inFile   dq ?
      outSize  dq ?
      outFile  dq ?
      outKey   dq ?
      encFlags dq ? ;;;number of passes, delete file after successful encryption
      encPath  rb 1000h
ends
WORKER_SIZE     equ 1000h+128+128+(8*10)
WORKER_OUT_SIZE equ WORKER_SIZE
ENC_BUFFER_SIZE equ 4096
ENCPATH_SIZE    equ 1000h

;;double quad word constants for addressing
DQ1 equ 0
DQ2 equ 16
DQ3 equ 32
DQ4 equ 48
DQ5 equ 64
DQ6 equ 80
DQ7 equ 96
DQ8 equ 112

;;Key size 128 bytes
KEYSIZE    equ 128
HASHSIZE   equ 16

;;;UTILITY methods
;;;This code is partially reusuable for non related projects.

;;;return: number of processors on the current system
CountProcessors:
       sub     rsp,8*8
       call    [GetCurrentProcess]
       mov     rcx,rax
       mov     rdx,ProcessAffinity
       mov     r8,SystemAffinity
       call    [GetProcessAffinityMask]
       mov     rdx,qword[SystemAffinity]
       mov     ecx,64
       bsr     rdx,rdx
       mov     eax,ecx
       sub     ecx,edx
       add     eax,1
       sub     eax,ecx
       mov     dword[processorcount],eax
       add     rsp,8*8
       ret     0
```

```
;;;input: rcx=integer rdx=buffer
IntToString:
      sub     rsp,8*5
      mov     r8,rdx
      mov     r10,rdx
      xor     edx,edx
      mov     eax,ecx
      mov     r11,IntToStrLUT
      mov     ecx,10
      dec     r10
  .loop:
      xor     edx,edx
      div     ecx
      mov     r9b,byte[r11+rdx]
      mov     byte[r8],r9b
      inc     r8
      test    eax,eax
      jnz     .loop
      mov     byte[r8],0
  .flip:
      dec     r8
      inc     r10
      cmp     r8,r10
      jbe     .end
      mov     al,byte[r8]
      mov     cl,byte[r10]
      mov     byte[r10],al
      mov     byte[r8],cl
      jmp     .flip
  .end:
      add     rsp,8*5
      ret     0

;;;input: rcx=addr of binary data rdx=addr of string
;;;inputs are align 16
;;;notes: hard coded for 128byte binary data
BinToString:
      movdqa  dqword[rsp-40],xmm8
      movdqa  dqword[rsp-56],xmm9
      movq    xmm7,qword[And0F]
      movq    xmm6,qword[AndF0]
      movdqa  xmm5,dqword[Add30]
      movdqa  xmm4,dqword[Cmp39]
      movdqa  xmm3,dqword[And07]
      mov     eax,8
;;;
  .loop:
      movq    xmm0,qword[rcx]
      movq    xmm8,qword[rcx+8]
      movq    xmm1,xmm0
      movq    xmm9,xmm8
      pand    xmm0,xmm6
      pand    xmm8,xmm6
      pand    xmm1,xmm7
      pand    xmm9,xmm7
      psrlq   xmm0,4
```

```
        psrlq   xmm8,4
        punpcklbw xmm1,xmm0
        punpcklbw xmm9,xmm8
        paddb   xmm1,xmm5
        paddb   xmm9,xmm5
        movdqa  xmm0,xmm1
        movdqa  xmm8,xmm9
        pcmpgtb xmm1,xmm4
        pcmpgtb xmm9,xmm4
        pand    xmm1,xmm3
        pand    xmm9,xmm3
        paddb   xmm0,xmm1
        paddb   xmm8,xmm9
        movdqa  [rdx],xmm0
        movdqa  [rdx+16],xmm8
        add     rcx,16
        add     rdx,32
        dec     eax
        jnz     .loop
;;;
        movdqa  xmm8,dqword[rsp-40]
        movdqa  xmm9,dqword[rsp-56]
        ret     0

;;;input: rcx=addr of string rdx=addr of binary data
;;;inputs are align 16
;;;notes: hard coded for 128byte binary data 256byte string
StringToBin:
        movdqa  xmm7,dqword[Add30]
        movdqa  xmm6,dqword[Cmp09]
        movdqa  xmm5,dqword[And07]
        movdqa  xmm4,dqword[And00FF]
        sub     rdx,16 ;;;optimization in loop
        mov     eax,8
;;;
  .loop:
        movdqa  xmm0,dqword[rcx]
        movdqa  xmm2,dqword[rcx+16]
        psubb   xmm0,xmm7
        psubb   xmm2,xmm7
        movdqa  xmm1,xmm0
        movdqa  xmm3,xmm2
        pcmpgtb xmm0,xmm6
        pcmpgtb xmm2,xmm6
        pand    xmm0,xmm5
        pand    xmm2,xmm5
        psubb   xmm1,xmm0
        psubb   xmm3,xmm2
        movdqa  xmm0,xmm1
        movdqa  xmm2,xmm3
        psrlw   xmm1,4
        psrlw   xmm3,4
        por     xmm0,xmm1
        por     xmm2,xmm3
        pand    xmm0,xmm4
        pand    xmm2,xmm4
```

```
        add     rcx,32
        packuswb xmm0,xmm2
        add     rdx,16
        movdqa  dqword[rdx],xmm0
        dec     eax
        jnz     .loop
        ret     0

;;;input: rcx = number of bytes to allocate
;;;return: address of memory if successful OR exit program with error message
AllocateOrDie:
        sub     rsp,8*5
        mov     rdx,rcx
        mov     r8d,MEM_COMMIT or MEM_RESERVE
        xor     rcx,rcx
        mov     r9d,PAGE_READWRITE
        call    [VirtualAlloc]
        test    rax,rax
        jz      .error
        add     rsp,8*5
        ret     0
   .error:
        xor     ecx,ecx
        mov     r8,StrError
        mov     rdx,StrOutOfMem
        mov     r9,rcx
        call    [MessageBox]
        xor     ecx,ecx
        call    [ExitProcess]

;;;input: address of memory that needs to be freed
Deallocate:
        sub     rsp,8*5
        xor     edx,edx
        mov     r8d,MEM_RELEASE
        call    [VirtualFree]
        add     rsp,8*5
        ret     0

;;;PROE Utility Code Data

;;;count processors
ProcessAffinity dq 0
SystemAffinity dq 0
processorcount dd 0

IntToStrLUT:
db '0','1','2','3','4','5','6','7','8','9'

;;;BIN TO STRING TO BIN
align 16
And0F dq 0F0F0F0F0F0F0F0Fh,0F0F0F0F0F0F0F0Fh
AndF0 dq 0F0F0F0F0F0F0F0F0h,0F0F0F0F0F0F0F0F0h
And00FF dq 00FF00FF00FF00FFh,00FF00FF00FF00FFh
Add30 dq 3030303030303030h,3030303030303030h
Cmp39 dq 3939393939393939h,3939393939393939h
```

```
Cmp09 dq 0909090909090909h,0909090909090909h
And07 dq 0707070707070707h,0707070707070707h

StrError db 'Error',0
StrOutOfMem db 'Out of memory',0

;;;Dialogue Resource

  directory RT_DIALOG,dialogs

  resource dialogs,\
         37,LANG_ENGLISH+SUBLANG_DEFAULT,PROE

  dialog PROE,'PROE (Psuedo Random Optimized
Encyrption)',0,0,400,270,WS_CAPTION+WS_POPUP+WS_SYSMENU+DS_MODALFRAME

    dialogitem 'STATIC','Step 1: Select the file(s) you want encrypted, and then create the encryption key(s) for
them.',-1,10,10,380,8,WS_VISIBLE
    dialogitem
'SysListView32','',ID_INPUT,10,20,380,80,WS_VISIBLE+WS_VSCROLL+WS_HSCROLL+LVS_REPORT
;;;
    dialogitem 'BUTTON','ADD',ID_ADD,10,105,35,13,WS_VISIBLE+WS_TABSTOP
    dialogitem 'BUTTON','REMOVE',ID_REMOVE,50,105,35,13,WS_VISIBLE+WS_TABSTOP
    dialogitem 'BUTTON','CLEAR',ID_CLEAR,90,105,35,13,WS_VISIBLE+WS_TABSTOP
    dialogitem 'BUTTON','Create Key(s)',ID_MAKEKEY,130,105,70,13,WS_VISIBLE+WS_TABSTOP
;;;
    dialogitem 'STATIC','Count:',-1,210,107,35,8,WS_VISIBLE
    dialogitem 'STATIC','0',ID_COUNT,245,107,35,8,WS_VISIBLE
;;;
    dialogitem 'STATIC','Step 2: Select the folder that you want the encryption keys to be saved to.',-
1,10,135,380,8,WS_VISIBLE
    dialogitem
'STATIC','',ID_KEYOUT,10,145,170,13,WS_VISIBLE+WS_BORDER+WS_TABSTOP+ES_AUTOHSCROLL
    dialogitem 'BUTTON','...',ID_KEYDIR,175,145,20,13,WS_VISIBLE+WS_TABSTOP

    dialogitem 'STATIC','Step 3: Select your encryption options.',-1,10,175,380,8,WS_VISIBLE
    dialogitem 'STATIC','Passes (higher values improve security but take longer to process)',-
1,22,190,300,8,WS_VISIBLE
    dialogitem
'EDIT','',ID_PASSES,10,190,10,13,WS_VISIBLE+WS_BORDER+WS_TABSTOP+ES_AUTOHSCROLL
    dialogitem 'BUTTON','&Delete Original File(s) (This is a highly recommended security
option)',ID_DELETE,10,200,300,13,WS_VISIBLE+WS_TABSTOP+BS_AUTOCHECKBOX

    dialogitem 'STATIC','',ID_STATUS,10,220,380,13,WS_VISIBLE+WS_TABSTOP+ES_AUTOHSCROLL
    dialogitem 'STATIC','',ID_ETIME,10,235,380,13,WS_VISIBLE+WS_TABSTOP+ES_AUTOHSCROLL

    dialogitem 'BUTTON','Start',IDOK,10,250,45,15,WS_VISIBLE+WS_TABSTOP+BS_DEFPUSHBUTTON
    dialogitem 'BUTTON','C&ancel',IDCANCEL,60,250,45,15,WS_VISIBLE+WS_TABSTOP+BS_PUSHBUTTON
  enddialog

;;;Dialog IDs for DialogPROE
ID_DIALOG       equ 37

ID_INPUT        equ 100
ID_ADD          equ 101
ID_REMOVE       equ 102
```

```
ID_CLEAR       equ 103
ID_COUNT       equ 104

ID_KEY         equ 200
ID_MAKEKEY     equ 300
ID_KEYOUT      equ 400
ID_KEYDIR      equ 500
ID_PASSES      equ 600
ID_DELETE      equ 700
ID_STATUS      equ 800
ID_ETIME       equ 900

;;;PROE GUI Code
use64
;;;*************************************
;;;input: rcx=addr of string rdx=item#
AddItem:
      sub    rsp,8*7
      mov    qword[LVITEMpszText],rcx
      mov    dword[LVITEMiItem],edx
      ;;;test for duplicates
      mov    qword[LVFINDINFOpsz],rcx
      mov    rcx,qword[lvhandle]
      mov    rdx,LVM_FINDITEM
      mov    r8,-1
      mov    r9,LVFINDINFO
      call   [SendMessage]
      cmp    rax,-1
      jne    .end
      ;;;
      mov    dword[LVITEMiSubItem],0
      mov    rcx,qword[lvhandle]
      mov    edx,LVM_INSERTITEM
      mov    r8d,0
      mov    r9,LVADD
      call   [SendMessage]
      call   UpdateCount
  .end:
      add    rsp,8*7
      ret    0
;;;input rcx=addr of string rdx=item# r8=subitem#
SetItem:
      sub    rsp,8*7
      mov    qword[LVITEMpszText],rcx
      mov    dword[LVITEMiItem],edx
      mov    dword[LVITEMiSubItem],r8d
      mov    rcx,qword[lvhandle]
      mov    edx,LVM_SETITEM
      mov    r8d,0
      mov    r9,LVADD
      call   [SendMessage]
      add    rsp,8*7
      ret    0
;;;**************************************
UpdateCount:
      sub    rsp,8*7
```

```
      xor     r8d,r8d
      mov     rcx,qword[lvhandle]
      mov     edx,LVM_GETITEMCOUNT
      mov     r9,r8
      call    [SendMessage]
      mov     rcx,rax
      mov     rdx,countbuffer
      call    IntToString
      mov     r9,countbuffer
      xor     r8d,r8d
      mov     edx,WM_SETTEXT
      mov     rcx,qword[counthandle]
      call    [SendMessage]
      add     rsp,8*7
      ret     0
;;;*************************************************
;;;Adds user selected files to the list view
GetFiles:
      pxor    xmm0,xmm0
      mov     [ofn.lStructSize],sizeof.OPENFILENAME
      xor     eax,eax
      mov     [ofn.hwndOwner],rax
      mov     qword[ofn.hInstance],0
      mov     qword[ofn.lpstrCustomFilter],0
      mov     [ofn.nFilterIndex],0
      mov     [ofn.nMaxFile],FILE_BUFFER_SIZE
      mov     rax,[name_buffer]
      mov     qword[ofn.lpstrFileTitle],rax
      movdqa  dqword[rax],xmm0
      mov     [ofn.nMaxFileTitle],FILE_BUFFER_SIZE
      mov     qword[ofn.lpstrInitialDir],0
      mov     qword[ofn.lpstrDefExt],0
      mov     rax,[path_buffer]
      mov     qword[ofn.lpstrFile],rax
      mov     ecx,FILE_BUFFER_SIZE-64
  .clear:
      movdqa  dqword[rax+rcx],xmm0
      movdqa  dqword[rax+rcx+16],xmm0
      movdqa  dqword[rax+rcx+32],xmm0
      movdqa  dqword[rax+rcx+48],xmm0
      sub     rcx,64
      jns     .clear
      mov     rax,open_filter
      mov     [ofn.lpstrFilter],rax
      mov     [ofn.Flags],OFN_OPEN_FLAGS
      mov     rax,[name_buffer]
      mov     [ofn.lpstrFileTitle],rax
      mov     [ofn.lpstrTitle],0
      mov     rcx,ofn
      call    [GetOpenFileName]
      test    rax,rax
      jz      .end
      movzx   rcx,word[ofn.nFileOffset]
      mov     rax,[path_buffer]
      test    byte[rax+rcx-1],0FFh
      jnz     .justone
```

```
    mov     rbx,rax
    lea     rsi,[rbx+rcx]
    mov     rdi,rsi
    mov     byte[rsi-1],'\'
  .loopFileNames:
    mov     rcx,rbx
    xor     edx,edx
    call    AddItem
    cmp     word[rsi-1],0
    je      .end
    mov     rcx,rdi
  .lstrcat:
    movzx   rax,byte[rsi]
    inc     rsi
    mov     byte[rcx],al
    inc     rcx
    test    al,al
    jnz     .lstrcat
    jmp     .loopFileNames
  .justone:
    mov     rcx,rax
    xor     edx,edx
    call    AddItem
  .end:
    jmp     processed
;;;*********************************
ClearListView:
    xor     ecx,ecx
    mov     r8,MsgBoxCap
    mov     rdx,MsgBoxClear
    mov     r9d,MB_YESNO
    call    [MessageBox]
    cmp     eax,IDYES
    jne     .end
    mov     edx,LVM_DELETEALLITEMS
    mov     rcx,qword[lvhandle]
    call    [SendMessage]
    call    UpdateCount
  .end:
    jmp     processed
;;;***************************************
RemoveItem:
    mov     rcx,qword[lvhandle]
    mov     edx,LVM_GETSELECTIONMARK
    call    [SendMessage]
    cmp     eax,-1
    je      .end
    xor     r9d,r9d
    mov     r8,rax
    mov     edx,LVM_DELETEITEM
    mov     rcx,qword[lvhandle]
    call    [SendMessage]
    mov     rcx,qword[lvhandle]
    mov     edx,LVM_SETSELECTIONMARK
    mov     r9,-1
    call    [SendMessage]
```

```
      call    UpdateCount
  .end:
      jmp     processed
;;;*********************************************
;;;;;;;;;;;;;;;;Mouse Coords To Improve Randomness
CreateKeys:
      xor     ecx,ecx
      lea     r8,[ThreadForEncKey]
      mov     r9,rcx
      mov     rdx,rcx
      mov     qword[rsp+8*4],rcx
      mov     qword[rsp+8*5],rcx
      call    [CreateThread]
      test    rax,rax
      jz      .error
      mov     rcx,rax
      call    [CloseHandle]
;;;for GUI
      mov     rcx,qword[addhandle]
      mov     rdx,FALSE
      call    [EnableWindow]
      mov     rcx,qword[removehandle]
      mov     rdx,FALSE
      call    [EnableWindow]
      mov     rcx,qword[clearhandle]
      mov     rdx,FALSE
      call    [EnableWindow]
      mov     rcx,qword[makekeyhandle]
      mov     rdx,FALSE
      call    [EnableWindow]
;;;
      jmp     processed
  .error:
      xor     ecx,ecx
      mov     r8,MsgBoxCap
      mov     rdx,MsgBoxError
      mov     r9,rcx
      call    [MessageBox]
      jmp     processed
ThreadForEncKey:
      sub     rsp,8*7
      mov     rcx,UtilsDll
      call    [GetModuleHandle]
      mov     r8,rax
      mov     rcx,WH_MOUSE_LL
      mov     rdx,[CallbackMouseHook]
      mov     r9,0
      call    [SetWindowsHookEx]
      test    rax,rax
      jz      .end
      mov     rbx,rax
      xor     ecx,ecx
      mov     rdx,MsgBoxMouse
      mov     r8,MsgBoxCap
      mov     r9,rcx
      call    [MessageBox]
```

```
        mov     rcx,rbx
        call    [UnhookWindowsHookEx]
        mov     rax,qword[SharedAddr]
        mov     rcx,qword[rax] ;;;1st qword of ForEncKey
        mov     rax,qword[rax+8] ;;;2nd qword
        mov     qword[ForEncKey],rcx
        mov     qword[ForEncKey+8],rax
  .end:
        xor     ecx,ecx
        mov     rdx,MsgBoxMouse2
        mov     r8,MsgBoxCap
        mov     r9,rcx
        call    [MessageBox]
        call    MakeKeys ;;;improve the probability of a random key generation
        call    MakeKeys
        call    MakeKeys
;;;for GUI
        mov     rcx,qword[addhandle]
        mov     rdx,TRUE
        call    [EnableWindow]
        mov     rcx,qword[removehandle]
        mov     rdx,TRUE
        call    [EnableWindow]
        mov     rcx,qword[clearhandle]
        mov     rdx,TRUE
        call    [EnableWindow]
        mov     rcx,qword[makekeyhandle]
        mov     rdx,TRUE
        call    [EnableWindow]
;;;
        add     rsp,8*7
        ret     0
;;;LRESULT CALLBACK
;;;Input: rcx=nCode rdx=wParm r8=lParm
;;;IN UTILS DLL
;;;CallbackMouseHook
;;;;;;;;;;;;;;;;;;;;
MakeKeys:
        sub     rsp,8*5
;;;number of files
        xor     r8d,r8d
        mov     rcx,qword[lvhandle]
        mov     edx,LVM_GETITEMCOUNT
        mov     r9,r8
        call    [SendMessage]
        test    rax,rax
        jz      .end
        dec     rax
        mov     rbx,rax
        mov     rdi,primes
  .outerloop:
        mov     rbp,8
        mov     rsi,keyhold
  .loop:
        rdtsc
        rol     rax,37
```

```
        xor     rax,qword[ForEncKey]
        movzx   rcx,al
        add     qword[rsi],rax
        movzx   rcx,word[rdi+rcx]
        xor     rax,qword[ForEncKey+8]
        mul     rcx
        add     qword[rsi+8],rax
        sub     qword[ForEncKey],rax
        bswap   rax
        sub     qword[ForEncKey],rax
        add     rsi,16
        dec     rbp
        jnz     .loop
        mov     rcx,keyhold
        mov     rdx,keybuffer
        call    BinToString
        mov     rcx,keybuffer
        mov     rdx,rbx
        mov     r8,1
        call    SetItem
        dec     rbx
        jns     .outerloop
  .end:
        add     rsp,8*5
        ret     0

KeySaveDir:
        mov     rax,[browse_buffer]
        mov     qword[browse_dir],rax
        mov     rcx,browseinfo
        call    [SHBrowseForFolder]
        mov     rdx,[browse_buffer]
        mov     rcx,rax
        mov     rbx,rax
        call    [SHGetPathFromIDList]
        mov     r9,[browse_buffer]
        xor     r8d,r8d
        mov     rdx,WM_SETTEXT
        mov     rcx,qword[keyouthandle]
        call    [SendMessage]
        mov     rcx,rbx
        call    [CoTaskMemFree]
        jmp     processed


;;; rbx = file counter
StartEncryption:
;;;reset
        mov     dword[CompletionCount],0
        mov     dword[ErrorCount],0
;;;error checks
        xor     r8d,r8d
        mov     rcx,qword[lvhandle]
        mov     edx,LVM_GETITEMCOUNT
        mov     r9,r8
        call    [SendMessage]
```

```
      mov     rdx,MsgBoxFiles
      test    rax,rax
      jz      .error
      mov     [filecount],rax
      mov     rax,[browse_buffer]
      mov     rdx,MsgBoxKeyDir
      test    byte[rax],0FFh
      jz      .error
  .lenloop:
      inc     rax
      test    byte[rax],0FFh
      jnz     .lenloop
      cmp     byte[rax-1],'\'
      je      .noappend
      cmp     byte[rax-1],'/'
      je      .noappend
      mov     word[rax],005Ch ;;;string '\',0
  .noappend:
      mov     rcx,qword[passeshandle]
      mov     r8,2
      mov     r9,passescount
      mov     rdx,WM_GETTEXT
      mov     word[r9],0
      call    [SendMessage]
      mov     rax,passescount
      mov     rdx,MsgBoxPasses
      cmp     byte[rax],32h ;;; string '2'
      jb      .error
      cmp     byte[rax],39h ;;; string '9'
      ja      .error
      sub     byte[rax],30h ;;;ascii to bin
;;;
      mov     rcx,qword[filecount]
      call    StackInit
      test    rax,rax
      jz      .error
      mov     rbx,qword[filecount]
      dec     rbx
  .loop:
;;; allocate memory for 1 stack node
      mov     rcx,WORKER_SIZE
      call    AllocateOrDie
      mov     rsi,rax
;;;encKey
      mov     dword[lvstring.iItem],ebx
      mov     rax,[lv_buffer]
      mov     dword[lvstring.iSubItem],1
      mov     dword[lvstring.cchTextMax],FILE_BUFFER_SIZE
      mov     qword[lvstring.pszText],rax
      mov     rcx,qword[lvhandle]
      mov     rdx,LVM_GETITEMTEXT
      mov     r8,rbx
      lea     r9,[lvstring]
      call    [SendMessage]
      test    rax,rax
      jz      .error
```

```
        mov     rcx,[lv_buffer]
        lea     rdx,[keyhold]
        call    StringToBin
        lea     rcx,[keyhold]
        movdqa  xmm0,dqword[rcx+DQ1]
        movdqa  xmm1,dqword[rcx+DQ2]
        movdqa  xmm2,dqword[rcx+DQ3]
        movdqa  xmm3,dqword[rcx+DQ4]
        movdqa  xmm4,dqword[rcx+DQ5]
        movdqa  xmm5,dqword[rcx+DQ6]
        movdqa  xmm6,dqword[rcx+DQ7]
        movdqa  xmm7,dqword[rcx+DQ8]
        movdqa  dqword[rsi+WORKER.encKey+DQ1],xmm0
        movdqa  dqword[rsi+WORKER.encKey+DQ2],xmm1
        movdqa  dqword[rsi+WORKER.encKey+DQ3],xmm2
        movdqa  dqword[rsi+WORKER.encKey+DQ4],xmm3
        movdqa  dqword[rsi+WORKER.encKey+DQ5],xmm4
        movdqa  dqword[rsi+WORKER.encKey+DQ6],xmm5
        movdqa  dqword[rsi+WORKER.encKey+DQ7],xmm6
        movdqa  dqword[rsi+WORKER.encKey+DQ8],xmm7
        movdqa  dqword[rsi+WORKER.decKey+DQ1],xmm0
        movdqa  dqword[rsi+WORKER.decKey+DQ2],xmm1
        movdqa  dqword[rsi+WORKER.decKey+DQ3],xmm2
        movdqa  dqword[rsi+WORKER.decKey+DQ4],xmm3
        movdqa  dqword[rsi+WORKER.decKey+DQ5],xmm4
        movdqa  dqword[rsi+WORKER.decKey+DQ6],xmm5
        movdqa  dqword[rsi+WORKER.decKey+DQ7],xmm6
        movdqa  dqword[rsi+WORKER.decKey+DQ8],xmm7
;;;inFile
        mov     dword[lvstring.iItem],ebx
        mov     rax,[lv_buffer]
        mov     dword[lvstring.iSubItem],0
        mov     dword[lvstring.cchTextMax],FILE_BUFFER_SIZE
        mov     qword[lvstring.pszText],rax
        mov     rcx,qword[lvhandle]
        mov     rdx,LVM_GETITEMTEXT
        mov     r8,rbx
        lea     r9,[lvstring]
        call    [SendMessage]
        test    rax,rax
        jz      .error
        mov     rcx,[lv_buffer]
        mov     rdx,GENERIC_READ
        mov     r8,FILE_SHARE_READ
        xor     r9d,r9d
        mov     qword[rsp+4*8],OPEN_EXISTING
        mov     qword[rsp+5*8],FILE_FLAG_SEQUENTIAL_SCAN
        mov     qword[rsp+6*8],0
        call    [CreateFile]
        cmp     rax,INVALID_HANDLE_VALUE
        je      .error
        mov     qword[rsi+WORKER.inFile],rax
        mov     rcx,rax
        lea     rdx,[rsi+WORKER.inSize]
        call    [GetFileSizeEx]
;;;outFile
```

```
        mov     rcx,[lv_buffer]
        lea     rdx,[out_file_ext]
        call    [lstrcat]
        lea     rcx,[rsi+WORKER.encPath]
        mov     rdx,[lv_buffer]
        call    [lstrcat]
        mov     rcx,[lv_buffer]
        mov     rdx,GENERIC_WRITE
        mov     r8,0
        xor     r9d,r9d
        mov     qword[rsp+4*8],CREATE_ALWAYS
        mov     qword[rsp+5*8],0
        mov     qword[rsp+6*8],0
        call    [CreateFile]
        cmp     rax,INVALID_HANDLE_VALUE
        je      .error
        mov     qword[rsi+WORKER.outFile],rax
;;;outKey
        mov     rcx,[browse_buffer]
        xor     edi,edi
  .nullloop:
        inc     edi
        test    byte[rcx+rdi],0FFh
        jnz     .nullloop
        mov     rdx,[lv_buffer]
        xor     eax,eax
  .lastpart:
        inc     rdx
        test    byte[rdx],0FFh
        jz      .lastover
        cmp     byte[rdx],'\'
        je      .saveaddr
        cmp     byte[rdx],'/'
        jne     .lastpart
  .saveaddr:
        mov     rax,rdx
        jmp     .lastpart
  .lastover:
        test    rax,rax
        jz      .error
        inc     rax
        mov     rdx,rax
        call    [lstrcat]
        mov     rcx,[browse_buffer]
        lea     rdx,[enc_file_ext]
        call    [lstrcat]
        mov     rcx,[browse_buffer]
        mov     rdx,GENERIC_WRITE
        mov     r8,0
        xor     r9d,r9d
        mov     qword[rsp+4*8],CREATE_ALWAYS
        mov     qword[rsp+5*8],0
        mov     qword[rsp+6*8],0
        call    [CreateFile]
        cmp     rax,INVALID_HANDLE_VALUE
        je      .error
```

```
        mov     [rsi+WORKER.outKey],rax
        mov     rcx,[browse_buffer]
        mov     qword[rcx+rdi],0 ;;;null after the end of the original key directory
;;;encFlags
        movzx   rcx,byte[passescount]
        mov     byte[rsi+WORKER.encFlags],cl
        mov     rdx,ID_DELETE
        mov     rcx,[rsp+8*8] ;;; dlg box handle
        call    [IsDlgButtonChecked]
        cmp     rax,BST_CHECKED
        jne     .nodelete
        mov     byte[rsi+WORKER.encFlags+1],1
  .nodelete:
;;;push
        mov     rcx,rsi
        call    StackPush
;;;
        dec     rbx
        jns     .loop
;;;
        call    [GetTickCount]
        mov     dword[starttime],eax
        mov     qword[bytecount],0
;;;create status thread and the worker threads
        mov     ebx,dword[processorcount]
        xor     ecx,ecx
        lea     r8,[ThreadStatus]
        mov     r9,rbx
        mov     rdx,rcx
        mov     qword[rsp+8*4],rcx
        mov     qword[rsp+8*5],rcx
        call    [CreateThread]
        test    rax,rax
        jz      .error
        mov     rcx,rax
        call    [CloseHandle]
  .loopworker:
        xor     ecx,ecx
        lea     r8,[ThreadWorker]
        mov     r9,rcx
        mov     rdx,rcx
        mov     qword[rsp+8*4],rcx
        mov     qword[rsp+8*5],rcx
        call    [CreateThread]
        test    rax,rax
        jz      .error
        mov     rcx,rax
        call    [CloseHandle]
        dec     ebx
        jnz     .loopworker
        jmp     processed
  .error:
        xor     ecx,ecx
        mov     r8,MsgBoxCap
        mov     r9,rcx
        call    [MessageBox]
```

```
        jmp     processed

;;;Updates the status information for the GUI
;;;input rcx = number of threads
ThreadStatus:
        mov     rbx,rcx ;;;save number of threads
        mov     rsi,[statushandle]
        mov     rdi,[etimehandle]
        mov     r12,CompletionCount
        mov     r13,ErrorCount
        mov     r14,StatusBuffer
        sub     rsp,8*5
   .loop:
        mov     rcx,STATUS_SLEEP
        xor     edx,edx
        call    [SleepEx] ;;;Sleep 1 second
;;;
        mov     rcx,r14
        mov     rdx,StrStatus1
        mov     r8d,dword[r13]
        call    [wsprintf]
        mov     rcx,rsi
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        call    [GetTickCount]
        sub     eax,dword[starttime]
        xor     edx,edx
        mov     ecx,1000
        div     ecx
        add     eax,1
        mov     rcx,r14
        mov     rdx,StrEtime1
        mov     r8d,eax
        call    [wsprintf]
        mov     rcx,rdi
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        cmp     ebx,dword[r12]
        jg      .loop
   .lastoutput:
        mov     rcx,r14
        mov     rdx,StrStatus2
        mov     r8d,dword[bytecount]
        mov     r9d,dword[r13]
        call    [wsprintf]
        mov     rcx,rsi
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        mov     eax,dword[endtime]
        sub     eax,dword[starttime]
```

```
        xor     edx,edx
        mov     ecx,1000
        div     ecx
        add     eax,1
        mov     r8d,eax
        mov     rax,qword[bytecount]
        xor     edx,edx
        mov     rcx,r8
        div     rcx
        mov     rcx,r14
        mov     rdx,StrEtime2
        mov     r9d,eax
        call    [wsprintf]
        mov     rcx,rdi
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        add     rsp,8*5
        call    [ExitThread]
        ret     0

;;;PROE GUI Data

;;;Window component handles
handle dq 0
keyouthandle dq 0
passeshandle dq 0
deletehandle dq 0
lvhandle dq 0
addhandle dq 0
removehandle dq 0
clearhandle dq 0
makekeyhandle dq 0
counthandle dq 0
statushandle dq 0
etimehandle  dq 0

;;;memory handles
path_buffer dq 0
name_buffer dq 0
lv_buffer   dq 0
browse_buffer dq 0

;;;OPENFILENAME struc
align 16
ofn OPENFILENAME
open_filter  db 'All files',0,'*.*',0,0
align 16
browseinfo dq 0,0
        browse_dir dq 0,0,0,0,0,0
;;;Extra Randomness
MsgBoxMouse db 'Please move your mouse sporadically around the screen for the a few seconds. This will help make your Encryption Key(s) be more secure. Press OK when you are done',0
MsgBoxMouse2 db 'Your Encyrption Key(s) will now be created.',0
UtilsDll db 'MouseHook.DLL',0
```

```
;;;GLOBAL

MsgBoxCap db 'PROE (Psuedo Random Optimized Encryption)',0
MsgBoxError db 'An error occurred, you need more memory to perform the operation.',0
NumProcessors dd 0

;;;DEBUG
fmthh    db '%X %X ',0
fmtf     db '%f ',0

  flags dd ?,?
  caption rb 40h
  message rb 100h


countbuffer db 16 dup(0)
LV_HEADER1 db 'File Name',0
LV_HEADER2 db 'Encryption Key',0
LVCOLUMN_1:
   dd LV_COL_MASK
   dd 0
   dd LV_COL_WIDTH
   dd     0
   LVCOLUMNpszText dq 0
   LVCOLUMNcchTextMax dd 0
   dq 0,0,0,0,0,0,0

LVADD:
   dd LVIF_TEXT
   LVITEMiItem dd 0
   LVITEMiSubItem dd 0
   dd 0
   dd 0
   dd   0
   LVITEMpszText dq 0
   LVITEMcchTextMax dd 0
   dq 0,0,0,0,0

LVFINDINFO:
   dd LVFI_STRING
   dd 0
   LVFINDINFOpsz dq 0
   dq 0,0,0

MsgBoxClear db 'Do you want to clear the file and encyrption key list?',0

starttime dd 0
endtime   dd 0
bytecount dq 0

;;;Encyrption Start
MsgBoxKeyDir db 'Please choose a directory to save the encryption key(s) to. It is recommended you save them to a
removable disk (ie: USB thumbdrive).',0
passescount db 0,0
MsgBoxPasses db 'Please be sure the number of passes is between 2 and 9.',0
```

```
deletecheck db 0
MsgBoxDelete db 'If you do not delete the original file(s) your data will remain insecure.',0
filecount dq 0
MsgBoxFiles db 'You need to select at least one file to encrypt.',0
lvstring LV_ITEM
out_file_ext db '.proe',0
enc_file_ext  db '.prok',0

StrStatus1    db 'Status: Encrypting (%lu Error(s) Encountered).',0
StrStatus2    db 'Status: Finished, %lu byte(s) encrypted (%lu Error(s) Encountered).',0
StrEtime1     db 'Elapsed Time: %lu second(s)',0
StrEtime2     db 'Elapsed Time: %lu second(s) (Avg Speed %lu Byte(s)/Second).',0
StatusBuffer  rb 255

;;;PROE GUI Equates

STATUS_SLEEP equ 1000

FILE_BUFFER_SIZE equ 100000h ;;;1MB
;;;*********************************************
;;;OPEN FILE
OFN_FORCESHOWHIDDEN     equ 10000000h
OFN_OPEN_FLAGS equ OFN_EXPLORER or OFN_ALLOWMULTISELECT or
OFN_NOREADONLYRETURN or OFN_HIDEREADONLY or OFN_FORCESHOWHIDDEN or
OFN_FILEMUSTEXIST

;;;****************************************
;;;List View
  LVS_REPORT = 0x1
  LVS_ICON = 0x0
  LVS_REPORT = 0x1
  LVS_SMALLICON = 0x2
  LVS_LIST = 0x3
  LVS_TYPEMASK = 0x3
  LVS_SINGLESEL = 0x4
  LVS_SHOWSELALWAYS = 0x8
  LVS_SORTASCENDING = 0x10
  LVS_SORTDESCENDING = 0x20
  LVS_SHAREIMAGELISTS = 0x40
  LVS_NOLABELWRAP = 0x80
  LVS_AUTOARRANGE = 0x100
  LVS_EDITLABELS = 0x200

  LVM_FIRST = 0x1000
  LVM_GETHEADER = (LVM_FIRST + 31)
  LVM_GETBKCOLOR = (LVM_FIRST + 0)
  LVM_SETBKCOLOR = (LVM_FIRST + 1)
  LVM_GETIMAGELIST = (LVM_FIRST + 2)
  LVM_SETIMAGELIST = (LVM_FIRST + 3)
  LVM_GETITEMCOUNT = (LVM_FIRST + 4)
  LVM_GETITEMA = (LVM_FIRST + 5)
  LVM_GETITEM = LVM_GETITEMA
  LVM_SETITEMA = (LVM_FIRST + 6)
  LVM_SETITEM = LVM_SETITEMA
  LVM_INSERTITEMA = (LVM_FIRST + 7)
  LVM_INSERTITEM = LVM_INSERTITEMA
```

LVM_DELETEITEM = (LVM_FIRST + 8)
LVM_DELETEALLITEMS = (LVM_FIRST + 9)
LVM_GETCALLBACKMASK = (LVM_FIRST + 10)
LVM_SETCALLBACKMASK = (LVM_FIRST + 11)
LVM_GETNEXTITEM = (LVM_FIRST + 12)
LVM_FINDITEMA = (LVM_FIRST + 13)
LVM_FINDITEM = LVM_FINDITEMA
LVM_GETITEMRECT = (LVM_FIRST + 14)
LVM_SETITEMPOSITION = (LVM_FIRST + 15)
LVM_GETITEMPOSITION = (LVM_FIRST + 16)
LVM_GETSTRINGWIDTHA = (LVM_FIRST + 17)
LVM_GETSTRINGWIDTH = LVM_GETSTRINGWIDTHA
LVM_HITTEST = (LVM_FIRST + 18)
LVM_ENSUREVISIBLE = (LVM_FIRST + 19)
LVM_SCROLL = (LVM_FIRST + 20)
LVM_REDRAWITEMS = (LVM_FIRST + 21)
LVM_ARRANGE = (LVM_FIRST + 22)
LVM_EDITLABELA = (LVM_FIRST + 23)
LVM_EDITLABEL = LVM_EDITLABELA
LVM_GETEDITCONTROL = (LVM_FIRST + 24)
LVM_GETCOLUMNA = (LVM_FIRST + 25)
LVM_GETCOLUMN = LVM_GETCOLUMNA
LVM_SETCOLUMNA = (LVM_FIRST + 26)
LVM_SETCOLUMN = LVM_SETCOLUMNA
LVM_INSERTCOLUMNA = (LVM_FIRST + 27)
LVM_INSERTCOLUMN = LVM_INSERTCOLUMNA
LVM_DELETECOLUMN = (LVM_FIRST + 28)
LVM_GETCOLUMNWIDTH = (LVM_FIRST + 29)
LVM_SETCOLUMNWIDTH = (LVM_FIRST + 30)
LVM_CREATEDRAGIMAGE = (LVM_FIRST + 33)
LVM_GETVIEWRECT = (LVM_FIRST + 34)
LVM_GETTEXTCOLOR = (LVM_FIRST + 35)
LVM_SETTEXTCOLOR = (LVM_FIRST + 36)
LVM_GETTEXTBKCOLOR = (LVM_FIRST + 37)
LVM_SETTEXTBKCOLOR = (LVM_FIRST + 38)
LVM_GETTOPINDEX = (LVM_FIRST + 39)
LVM_GETCOUNTPERPAGE = (LVM_FIRST + 40)
LVM_GETORIGIN = (LVM_FIRST + 41)
LVM_UPDATE = (LVM_FIRST + 42)
LVM_SETITEMSTATE = (LVM_FIRST + 43)
LVM_GETITEMSTATE = (LVM_FIRST + 44)
LVM_GETITEMTEXTA = (LVM_FIRST + 45)
LVM_GETITEMTEXT = LVM_GETITEMTEXTA
LVM_SETITEMTEXTA = (LVM_FIRST + 46)
LVM_SETITEMTEXT = LVM_SETITEMTEXTA
LVM_SETITEMCOUNT = (LVM_FIRST + 47)
LVM_SORTITEMS = (LVM_FIRST + 48)
LVM_SETITEMPOSITION32 = (LVM_FIRST + 49)
LVM_GETSELECTEDCOUNT = (LVM_FIRST + 50)
LVM_GETITEMSPACING = (LVM_FIRST + 51)
LVM_GETISEARCHSTRINGA = (LVM_FIRST + 52)
LVM_GETISEARCHSTRING = LVM_GETISEARCHSTRINGA
LVM_SETICONSPACING = (LVM_FIRST + 53)
LVM_SETEXTENDEDLISTVIEWSTYLE = (LVM_FIRST + 54)
LVM_GETEXTENDEDLISTVIEWSTYLE = (LVM_FIRST + 55)
LVM_GETSUBITEMRECT = (LVM_FIRST + 56)

```
  LVM_SUBITEMHITTEST = (LVM_FIRST + 57)
  LVM_SETCOLUMNORDERARRAY = (LVM_FIRST + 58)
  LVM_GETCOLUMNORDERARRAY = (LVM_FIRST + 59)
  LVM_SETHOTITEM = (LVM_FIRST + 60)
  LVM_GETHOTITEM = (LVM_FIRST + 61)
  LVM_SETHOTCURSOR = (LVM_FIRST + 62)
  LVM_GETHOTCURSOR = (LVM_FIRST + 63)
  LVM_APPROXIMATEVIEWRECT = (LVM_FIRST + 64)
  LVM_GETSELECTIONMARK = (LVM_FIRST + 66)
  LVM_SETSELECTIONMARK = (LVM_FIRST + 67)
  LVS_EX_FULLROWSELECT = 0x20
  LVSCW_AUTOSIZE  = -1
  LVSCW_AUTOSIZE_USEHEADER = -2

  LVIF_TEXT = 1
  LVFI_STRING = 2

  LVCF_TEXT = 4
  LVCF_WIDTH = 2
  LV_COL_MASK = LVCF_WIDTH or LVCF_TEXT

  LV_COL_WIDTH = 285

struct LVCOLUMN
    mask  dd ?
    fmt   dd ?
    cx    dd ?
    pszText dq ?
    cchTextMax dd ?
    iSubItem   dq ?
    iImage dd ?
    iOrder dd ?
    cxMin  dd ?
    cxDefault dd ?
    cxIdeal dd ?
ends

struct LVITEM
    mask dd ?
    iItem dd ?
    iSubItem dd ?
    state dd ?
    stateMask dd ?;
    pszText dq ?
    cchTextMax dd ?
    iImage dd ?
    lParam dd ?
    iIndent dd ?
    iGroupId dd ?
    cColumns dd ?
    puColumns dq ?
    piColFmt dq ?
    iGroup dd ?
ends

;;;PROE MouseHook
```

```
;;;MouseHook.DLL
format PE64 DLL
entry EntryPoint
include '%fasminc%\win64a.inc'

section '.code' code readable executable

EntryPoint:
       mov     qword[hMod],rcx ;;;instance handle
       mov     eax,TRUE
       ret     0

;;;LRESULT CALLBACK
;;;Input: rcx=nCode rdx=wParm r8=lParm
CallbackMouseHook:
       mov     rax,[r8] ;;;POINT struct at MSLLHOOKSTRUCT.pt
       sub     rsp,8*8 ;;; set up stack for later api calls
       cmp     edx,WM_MOUSEMOVE ;;;if NOT mousemove event skip logic
       jne     .skip
       add     qword[ForEncKey],rax
       add     qword[ForEncKey+4],rax
       bswap   rax
       sub     qword[ForEncKey+8],rax
       add     dword[ForEncKey+12],eax
       mov     qword[rsp+8*6],rdx
       rdtsc
       xor     dword[ForEncKey],eax
       xor     dword[ForEncKey+4],eax
       xor     dword[ForEncKey+8],eax
       xor     dword[ForEncKey+12],eax
       add     eax,edx
       bswap   eax
       add     dword[ForEncKey],eax
       sub     dword[ForEncKey+4],eax
       add     dword[ForEncKey+8],eax
       sub     dword[ForEncKey+12],eax
       mov     rdx,qword[rsp+8*6]
  .skip:
       call    [CallNextHookEx]
       add     rsp,8*8
       ret     0


section '.data' data readable writeable
;;;module handle
hMod      dq 0
;;;Shared information
SharedAddr:
ForEncKey dq 0,0

section '.idata' import data readable writeable
  library kernel32,'KERNEL32.DLL',\
          user32,'USER32.DLL'
  include '%fasminc%\apia\kernel32.inc'
  include '%fasminc%\apia\user32.inc'
```

```
section '.edata' export data readable

  export 'MouseHook.DLL',\
       SharedAddr,'SharedAddr',\
       CallbackMouseHook,'CallbackMouseHook'
section '.reloc' fixups data discardable
```

Appendix D: PROE File Decryption source code

```
;;;LOUIS RICCI
;;;PROE Dialog GUI Main
;;;PROE Decrypt.exe
format PE64 GUI 4.0
entry start

include '%fasminc%\win64a.inc'
include 'DialogPROEEqu.asm'
include 'PROEDecEqu.asm'
include 'PROEGUIEqu.asm'

section '.data' data readable writeable
include 'UtilData.asm'
include 'PROEDecData.asm'
include 'PROEGUIData.asm'

section '.code' code readable executable

  start:
        sub     rsp,8*5
;;;
        xor     ecx,ecx
        call    [GetModuleHandle]
        mov     qword[handle],rax
;;;
        mov     qword [rsp+8*4],0
        lea     r9,[DialogProc]
        mov     r8d,HWND_DESKTOP
        mov     edx,ID_DIALOG
        mov     rcx,rax
        call    [DialogBoxParam]
        or      rax,rax
        jz      exit
  exit:
        xor     ecx,ecx
        call    [ExitProcess]
;;;input: rcx=handle rdx=msg r8=wParm r9=lParm
DialogProc:
        mov     rbx,rcx ;;; save for use later in the dialog proc and in "processed" functions
        push    rcx rdx r8 r9
        ;;;rsp+8* 8  7  6  5
        sub     rsp,8*5
        cmp     rdx,WM_INITDIALOG
        jz      wminitdialog
        cmp     rdx,WM_COMMAND
        jz      wmcommand
        cmp     rdx,WM_CLOSE
        jz      wmclose
        xor     eax,eax
        jmp     finish
  wminitdialog:
;;;general setup
        call    CountProcessors
        mov     rcx,FILE_BUFFER_SIZE
```

```
        call    AllocateOrDie
        mov     qword[path_buffer],rax
        mov     rcx,FILE_BUFFER_SIZE
        call    AllocateOrDie
        mov     qword[name_buffer],rax
        mov     rcx,FILE_BUFFER_SIZE
        call    AllocateOrDie
        mov     qword[lv_buffer],rax
        mov     rcx,FILE_BUFFER_SIZE
        call    AllocateOrDie
        mov     qword[browse_buffer],rax
        xor     ecx,ecx
        call    [OleInitialize]
;;;dialog controls, handle grabbing
        mov     edx,ID_STATUS
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     [statushandle],rax
        mov     edx,ID_ETIME
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     [etimehandle],rax
;;;setup the list view control
        mov     edx,ID_INPUT
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[lvhandle],rax
        mov     rcx,rax
        mov     edx,LVM_INSERTCOLUMN
        mov     r8d,1
        mov     r9,LVCOLUMN_1
        mov     rax,LV_HEADER1
        mov     qword[LVCOLUMNpszText],rax
        call    [SendMessage]
        mov     rcx,qword[lvhandle]
        mov     edx,LVM_INSERTCOLUMN
        mov     r8d,2
        mov     r9,LVCOLUMN_1
        mov     rax,LV_HEADER2
        mov     qword[LVCOLUMNpszText],rax
        call    [SendMessage]
        mov     edx,ID_COUNT
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[counthandle],rax
        mov     edx,ID_ADD
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[addhandle],rax
        mov     edx,ID_REMOVE
        mov     rcx,rbx
        call    [GetDlgItem]
        mov     qword[removehandle],rax
        mov     edx,ID_CLEAR
        mov     rcx,rbx
        call    [GetDlgItem]
```

```
      mov     qword[clearhandle],rax
      jmp     processed
  wmcommand:
;;;GUI INPUT PROCESSING
      cmp     r8d,BN_CLICKED shl 16 + ID_ADD
      jz      GetFiles
      cmp     r8d,BN_CLICKED shl 16 + ID_CLEAR
      jz      ClearListView
      cmp     r8d,BN_CLICKED shl 16 + ID_REMOVE
      jz      RemoveItem
      cmp     r8d,BN_CLICKED shl 16 + IDCANCEL
      jz      wmclose
      cmp     r8d,BN_CLICKED shl 16 + IDOK
      jz      StartDecryption
      jmp     processed
  wmclose:
      xor     edx,edx
      mov     rcx,[rsp + 8*8]
      call    [EndDialog]
  processed:
      mov     eax,1
  finish:
      add     rsp,8*5
      pop     r9 r8 rdx rcx
      ret

include 'PROEGUICode.asm'
include 'PROEDecCode.asm'
include 'UtilCode.asm'

section '.idata' import data readable writeable
;;;API imports
  library kernel32,'KERNEL32.DLL',\
        msvcrt,'MSVCRT.DLL',\
        comdlg32,'COMDLG32.DLL',\
        shell32,'SHELL32.DLL',\
        ole32,'OLE32.DLL',\
        user32,'USER32.DLL',\
        PROELib,'PROE_Lib_Win64.DLL'

      import PROELib,\
           PROESelfTest,'PROESelfTest',\
           PROEValidateParameters,'PROEValidateParameters',\
           PROEEncryptBuffer,'PROEEncryptBuffer',\
           PROEDecryptBuffer,'PROEDecryptBuffer',\
           PROEEncryptBufferWithChecksum,'PROEEncryptBufferWithChecksum',\
           PROEDecryptBufferWithChecksum,'PROEDecryptBufferWithChecksum'
      import comdlg32,\
           GetOpenFileName,'GetOpenFileNameA',\
           GetSaveFileName,'GetSaveFileNameA'
      import shell32,\
           SHBrowseForFolder,'SHBrowseForFolderA',\
           SHGetPathFromIDList,'SHGetPathFromIDListA'
      import ole32,\
           OleInitialize,'OleInitialize',\
           CoTaskMemFree,'CoTaskMemFree'
```

```
;;;DEBUGGING function(s)
       import msvcrt,\
             printf,'printf'

  include '%fasminc%\apia\kernel32.inc'
  include '%fasminc%\apia\user32.inc'

section '.rsrc' resource data readable
include 'DialogPROERes.asm'

;;;PROE Encyrption Code

;;;input: rcx = file size
;;;return: buffer size to use
FileSizeToBufferSize:
       add     rcx,64
       lea     rax,[FS2BS_LUT]
       bsr     rcx,rcx
       mov     rax,qword[rax+rcx*8]
       ret     0

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ThreadWorker:
       sub     rsp,8*8
       ;;;[rsp+8*6+1] ;;;counter for number of passes
       ;;;[rsp+8*7]   ;;;hold area for number of bytes to write out
       ;;;[rsp+8*4]-[rsp+8*5] ;;;temp hold for first 16bytes of key for hash
  .loop:
       mov     byte[rsp+8*6],0 ;;;boolean value for readInFile
;;;*********
       call    StackPop
       test    rax,rax
       jz      .finish
;;;store info in registers
       mov     rbp,rax
       mov     rdi,[rax+WORKER.inFile]
       mov     rbx,[rax+WORKER.inSize]
       mov     r12,[rax+WORKER.outFile]
       lea     r13,[rax+WORKER.decKey]
       lock add qword[bytecount],rbx
       movzx   r14,byte[rax+WORKER.encFlags]
       mov     rcx,[rax+WORKER.outSize]
       call    FileSizeToBufferSize
       mov     r15,rax
       mov     rcx,rax
       call    AllocateOrDie
       mov     rsi,rax
;;;calc size and ReadFile
  .readInFile:
       test    rbx,rbx
       jz      .done
       cmp     rbx,r15
       jae     .else
  .then:
       mov     r8,rbx
       mov     rcx,[rbp+WORKER.outSize]
```

```
    lea     rdx,[r15-1]
    lea     r15,[rbx+63]
    and     rcx,rdx
    mov     qword[rsp+8*7],rcx
    and     r15,0FFFFFFFFFFFFFFC0h ;;;Ceiling multiple of 64
    pxor    xmm0,xmm0
    lea     rcx,[r15-64]
    movdqa  dqword[rsi+rcx+DQ1],xmm0
    movdqa  dqword[rsi+rcx+DQ2],xmm0
    movdqa  dqword[rsi+rcx+DQ3],xmm0
    movdqa  dqword[rsi+rcx+DQ4],xmm0
    xor     ebx,ebx
    jmp     .endIf
  .else:
    mov     r8,r15
    mov     [rsp+8*7],r15
    sub     rbx,r15
  .endIf:
;;;
    mov     rcx,rdi
    mov     rdx,rsi
    mov     qword[rsp+8*4],0
    lea     r9,[rsp+8*5]
    call    [ReadFile]
    test    rax,rax
    jz      .error
;;;
    mov     rcx,rsi
    mov     rdx,r15
    mov     r8,r13
    mov     r9,r14
    call    [PROEDecryptBufferWithChecksum]
;;;hash and dencrypt the block of data
  .writeOutFile:
    mov     rcx,r12
    mov     rdx,rsi
    mov     r8,qword[rsp+8*7]
    lea     r9,[rsp+8*5]
    mov     qword[rsp+8*4],0
    call    [WriteFile]
    test    rax,rax
    jz      .error
    jmp     .readInFile
  .done:
;;;check hash
    mov     r8,[rbp+WORKER.encHash+DQ1]
    mov     r9,[rbp+WORKER.encHash+DQ1+8]
    mov     r10,[rbp+WORKER.decHash+DQ1]
    mov     r11,[rbp+WORKER.decHash+DQ1+8]
;       cmp     r8,r10
;       jne     .error
;       cmp     r9,r11
;       jne     .error
;;;clean up memory and file alloations
    mov     rcx,r12  ;;;outFile handle
    call    [CloseHandle]
```

```
        mov     rcx,rdi  ;;;inFile handle
        call    [CloseHandle]
        mov     rcx,rsi
        call    Deallocate
        mov     rcx,rbp
        call    Deallocate
        jmp     .loop
    .error:
        lock add dword[ErrorCount],1
        mov     rcx,ENCPATH_SIZE
        call    AllocateOrDie
        mov     rcx,rax
        mov     rbx,rax
        lea     rdx,[rbp+WORKER.encPath]
        mov     r8,8080808080808080h
    .strCopy:
        mov     rax,[rdx]
        add     rdx,8
        mov     [rcx],rax
        add     rcx,8
        lea     rax,[rax-0101010101010101h]
        and     rax,r8
        jz      .strCopy
        mov     rcx,rbx
        call    DecryptionError
        mov     rcx,[rbp+WORKER.inFile]
        call    [CloseHandle]
        mov     rcx,[rbp+WORKER.outFile]
        call    [CloseHandle]
        mov     rcx,rsi
        call    Deallocate
        mov     rcx,rbp
        call    Deallocate
        jmp     .loop
    .finish:
        call    [GetTickCount]
        mov     dword[endtime],eax
        lock add dword[CompletionCount],1
        add     rsp,8*9
        call    [ExitThread]
        ret     0

;;;INPUT rcx = size to make array stack
;;;returns NONZERO = SUCCESS  0 = FAIL
StackInit:
        shl     rcx,3
        mov     rdx,rcx ;;; size to qword
        xor     ecx,ecx
        mov     qword[rsp+8*4],rdx ;;; save
        mov     r8d,MEM_COMMIT or MEM_RESERVE
        mov     r9d,PAGE_READWRITE
        call    [VirtualAlloc]
        test    rax,rax
        jz      .fail
        mov     qword[stackPtr],rax
        mov     qword[stackHead],rax
```

```
        mov     rcx,qword[rsp+8*4]
        add     rax,rcx
        mov     qword[stackEnd],rax
        ret     0
  .fail:
        xor     eax,eax
        ret     0
;;;return NONZERO = SUCCESS  0 = FAIL
StackDelete:
        mov     rcx,qword[stackHead]
        xor     edx,edx
        mov     r8d,MEM_RELEASE
        call    [VirtualFree]
        test    rax,rax
        jz      .fail
        ret     0
  .fail:
        xor     eax,eax
        ret     0

;;;Single Thread PUSH (not thread safe)
;;;RCX = addr of data to put to the stack
;;;returns NONZERO = SUCCESS  0 = FAILURE
StackPush:
        mov     rax,8
        xadd    qword[stackPtr],rax  ;;; LOCKED exchange and add
        cmp     rax,qword[stackEnd]
        jae     .fail
        mov     qword[rax],rcx ;;;store addr of data
        ret     0
  .fail:
        xor     eax,eax
        ret     0

;;;Concurrent (Thread Safe) POP
;;;return data addr
;;;return if underflow 0 = failure
StackPop:
        mov     rax,-8
  lock xadd     qword[stackPtr],rax ;;; locked exchange and add(sub)
        cmp     rax,qword[stackHead]
        jbe     .fail
        mov     rax,qword[rax-8]  ;;;get addr of data
        ret     0
  .fail:
        xor     eax,eax
        ret     0

;;;input rcx = ptr to error string
DecryptionError:
        sub     rsp,8*7
        mov     r9,rcx
        xor     edx,edx
        lea     r8,[ThreadErrorMsg]
        mov     qword[rsp+8*4],rdx
        mov     qword[rsp+8*5],rdx
```

```
        mov     rcx,rdx
        call    [CreateThread]
        mov     rcx,rax
        call    [CloseHandle]
        add     rsp,8*7
        ret     0

;;;Creates an error message box in a seperate thread so encryption can continue
;;;input rcx = message string pointer
ThreadErrorMsg:
        sub     rsp,8*5
        mov     rbx,rcx ;;;save
        lea     r8,[StrDecError]
        mov     rdx,rcx
        xor     ecx,ecx
        mov     r9,rcx
        call    [MessageBox]
        mov     rcx,rbx
        call    Deallocate
        call    [ExitThread]
        add     rsp,8*5
        ret     0
        _fmth db '%x %x -',0

;;;PROE Decryption data

align 16
;;;File Size To Buffer Size LOOK-UP-TABLE
FS2BS_LUT: ;;;1-4095 bytes (4096)
          dq 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096, 4096
          ;;;4K - 16MB (4096 * Log[2](Size)
          dq 4096*2, 4096*4, 4096*8, 4096*16, 4096*32, 4096*64, 4096*128, 4096*256, 4096*512, 4096*1024,
4096*2048, 4096*4096
          ;;; > 16MB (32MB)
          dq 40 dup(4096*8192)
align 16
keyhold rb 128
align 16
keybuffer db 257 dup(0)

;;;Psuedo Random Number algorithm
align 16
SSERndMask dq 8000000000000000h
          dq 8000000000000000h

;;;concurrent array stack data
;;;Holds WORKER memory structure
align 16
stackPtr     dq 0
stackHead    dq 0
stackEnd     dq 0

;;;Thread Completion and Error Count
CompletionCount dd 0
ErrorCount      dd 0
```

```
;;;Error Message
StrDecError db 'Dencryption Error',0

;;;PROE Encryption/Decryption Equates

;;;Memory structure used by encyrption/decryption worker thread
struct WORKER
      encKey   rb 128
      encHash  dq ?
              dq ?
      decKey   rb 128
      decHash  dq ?
              dq ?
      inSize   dq ?
      inFile   dq ?
      outSize  dq ?
      outFile  dq ?
      outKey   dq ?
      encFlags dq ? ;;;number of passes, delete file after successful encryption
      encPath  rb 1000h
ends
WORKER_SIZE     equ 1000h+128+128+(8*10)
WORKER_OUT_SIZE equ WORKER_SIZE
ENC_BUFFER_SIZE equ 4096
ENCPATH_SIZE    equ 1000h

;;double quad word constants for addressing
DQ1 equ 0
DQ2 equ 16
DQ3 equ 32
DQ4 equ 48
DQ5 equ 64
DQ6 equ 80
DQ7 equ 96
DQ8 equ 112

;;Key size 128 bytes
KEYSIZE    equ 128
HASHSIZE   equ 16

;;;UTILITY methods
;;;This code is partially reusuable for non related projects.

;;;return: number of processors on the current system
CountProcessors:
       sub     rsp,8*8
       call    [GetCurrentProcess]
       mov     rcx,rax
       mov     rdx,ProcessAffinity
       mov     r8,SystemAffinity
       call    [GetProcessAffinityMask]
       mov     rdx,qword[SystemAffinity]
       mov     ecx,64
       bsr     rdx,rdx
       mov     eax,ecx
       sub     ecx,edx
```

```
        add     eax,1
        sub     eax,ecx
        mov     dword[processorcount],eax
        add     rsp,8*8
        ret     0

;;;input: rcx=integer rdx=buffer
IntToString:
        sub     rsp,8*5
        mov     r8,rdx
        mov     r10,rdx
        xor     edx,edx
        mov     eax,ecx
        mov     r11,IntToStrLUT
        mov     ecx,10
        dec     r10
   .loop:
        xor     edx,edx
        div     ecx
        mov     r9b,byte[r11+rdx]
        mov     byte[r8],r9b
        inc     r8
        test    eax,eax
        jnz     .loop
        mov     byte[r8],0
   .flip:
        dec     r8
        inc     r10
        cmp     r8,r10
        jbe     .end
        mov     al,byte[r8]
        mov     cl,byte[r10]
        mov     byte[r10],al
        mov     byte[r8],cl
        jmp     .flip
   .end:
        add     rsp,8*5
        ret     0

;;;input: rcx=addr of binary data rdx=addr of string
;;;inputs are align 16
;;;notes: hard coded for 128byte binary data
BinToString:
        movdqa  dqword[rsp-40],xmm8
        movdqa  dqword[rsp-56],xmm9
        movq    xmm7,qword[And0F]
        movq    xmm6,qword[AndF0]
        movdqa  xmm5,dqword[Add30]
        movdqa  xmm4,dqword[Cmp39]
        movdqa  xmm3,dqword[And07]
        mov     eax,8
;;;
   .loop:
        movq    xmm0,qword[rcx]
        movq    xmm8,qword[rcx+8]
        movq    xmm1,xmm0
```

```
        movq    xmm9,xmm8
        pand    xmm0,xmm6
        pand    xmm8,xmm6
        pand    xmm1,xmm7
        pand    xmm9,xmm7
        psrlq   xmm0,4
        psrlq   xmm8,4
        punpcklbw xmm1,xmm0
        punpcklbw xmm9,xmm8
        paddb   xmm1,xmm5
        paddb   xmm9,xmm5
        movdqa  xmm0,xmm1
        movdqa  xmm8,xmm9
        pcmpgtb xmm1,xmm4
        pcmpgtb xmm9,xmm4
        pand    xmm1,xmm3
        pand    xmm9,xmm3
        paddb   xmm0,xmm1
        paddb   xmm8,xmm9
        movdqa  [rdx],xmm0
        movdqa  [rdx+16],xmm8
        add     rcx,16
        add     rdx,32
        dec     eax
        jnz     .loop
;;;
        movdqa  xmm8,dqword[rsp-40]
        movdqa  xmm9,dqword[rsp-56]
        ret     0

;;;input: rcx=addr of string rdx=addr of binary data
;;;inputs are align 16
;;;notes: hard coded for 128byte binary data 256byte string
StringToBin:
        movdqa  xmm7,dqword[Add30]
        movdqa  xmm6,dqword[Cmp09]
        movdqa  xmm5,dqword[And07]
        movdqa  xmm4,dqword[And00FF]
        sub     rdx,16 ;;;optimization in loop
        mov     eax,8
;;;
  .loop:
        movdqa  xmm0,dqword[rcx]
        movdqa  xmm2,dqword[rcx+16]
        psubb   xmm0,xmm7
        psubb   xmm2,xmm7
        movdqa  xmm1,xmm0
        movdqa  xmm3,xmm2
        pcmpgtb xmm0,xmm6
        pcmpgtb xmm2,xmm6
        pand    xmm0,xmm5
        pand    xmm2,xmm5
        psubb   xmm1,xmm0
        psubb   xmm3,xmm2
        movdqa  xmm0,xmm1
        movdqa  xmm2,xmm3
```

```
        psrlw   xmm1,4
        psrlw   xmm3,4
        por     xmm0,xmm1
        por     xmm2,xmm3
        pand    xmm0,xmm4
        pand    xmm2,xmm4
        add     rcx,32
        packuswb xmm0,xmm2
        add     rdx,16
        movdqa  dqword[rdx],xmm0
        dec     eax
        jnz     .loop
        ret     0

;;;input: rcx = number of bytes to allocate
;;;return: address of memory if successful OR exit program with error message
AllocateOrDie:
        mov     rdx,rcx
        mov     r8d,MEM_COMMIT or MEM_RESERVE
        xor     rcx,rcx
        mov     r9d,PAGE_READWRITE
        call    [VirtualAlloc]
        test    rax,rax
        jz      .error
        ret     0
   .error:
        xor     ecx,ecx
        mov     r8,StrError
        mov     rdx,StrOutOfMem
        mov     r9,rcx
        call    [MessageBox]
        xor     ecx,ecx
        call    [ExitProcess]

;;;input: address of memory that needs to be freed
Deallocate:
        xor     edx,edx
        mov     r8d,MEM_RELEASE
        call    [VirtualFree]
        ret     0
;;;PROE Utility Code Data

;;;count processors
ProcessAffinity dq 0
SystemAffinity dq 0
processorcount dd 0

IntToStrLUT:
db '0','1','2','3','4','5','6','7','8','9'

;;;BIN TO STRING TO BIN
align 16
And0F dq 0F0F0F0F0F0F0F0Fh,0F0F0F0F0F0F0F0Fh
AndF0 dq 0F0F0F0F0F0F0F0F0h,0F0F0F0F0F0F0F0F0h
And00FF dq 00FF00FF00FF00FFh,00FF00FF00FF00FFh
Add30 dq 3030303030303030h,3030303030303030h
```

```
Cmp39 dq 3939393939393939h,3939393939393939h
Cmp09 dq 0909090909090909h,0909090909090909h
And07 dq 0707070707070707h,0707070707070707h

StrError db 'Error',0
StrOutOfMem db 'Out of memory',0

;;;PROE Decrypt Resource
 directory RT_DIALOG,dialogs

 resource dialogs,\
        37,LANG_ENGLISH+SUBLANG_DEFAULT,PROE

 dialog PROE,'PROE (Psuedo Random Optimized Encyrption)
Decrypt',0,0,400,170,WS_CAPTION+WS_POPUP+WS_SYSMENU+DS_MODALFRAME

   dialogitem 'STATIC','Select your decryption key file(s) (they should be on a removable storage device).',-
1,10,10,380,8,WS_VISIBLE
   dialogitem
'SysListView32','',ID_INPUT,10,20,380,80,WS_VISIBLE+WS_VSCROLL+WS_HSCROLL+LVS_REPORT
;;;
   dialogitem 'BUTTON','ADD',ID_ADD,10,105,35,13,WS_VISIBLE+WS_TABSTOP
   dialogitem 'BUTTON','REMOVE',ID_REMOVE,50,105,35,13,WS_VISIBLE+WS_TABSTOP
   dialogitem 'BUTTON','CLEAR',ID_CLEAR,90,105,35,13,WS_VISIBLE+WS_TABSTOP
;;;
   dialogitem 'STATIC','Count:',-1,210,107,35,8,WS_VISIBLE
   dialogitem 'STATIC','0',ID_COUNT,245,107,35,8,WS_VISIBLE
;;;
   dialogitem 'STATIC','',ID_STATUS,10,120,380,13,WS_VISIBLE+WS_TABSTOP+ES_AUTOHSCROLL
   dialogitem 'STATIC','',ID_ETIME,10,135,380,13,WS_VISIBLE+WS_TABSTOP+ES_AUTOHSCROLL

   dialogitem 'BUTTON','Start',IDOK,10,150,45,15,WS_VISIBLE+WS_TABSTOP+BS_DEFPUSHBUTTON
   dialogitem 'BUTTON','C&ancel',IDCANCEL,60,150,45,15,WS_VISIBLE+WS_TABSTOP+BS_PUSHBUTTON
 enddialog

;;;Dialog IDs for DialogPROE
ID_DIALOG      equ 37

ID_INPUT       equ 100
ID_ADD         equ 101
ID_REMOVE      equ 102
ID_CLEAR       equ 103
ID_COUNT       equ 104

ID_KEY         equ 200
ID_MAKEKEY     equ 300
ID_KEYOUT      equ 400
ID_KEYDIR      equ 500
ID_PASSES      equ 600
ID_DELETE      equ 700
ID_STATUS      equ 800
ID_ETIME       equ 900

;;;PROE Decrypt GUI Code
use64
;;;************************************
```

```
;;;input: rcx=addr of string rdx=item#
AddItem:
    sub     rsp,8*7
    ;;;[rsp+8*6] ;;;hold for file handle
    mov     qword[LVITEMpszText],rcx
    mov     dword[LVITEMiItem],edx
    mov     qword[LVFINDINFOpsz],rcx
    ;;;load the key file into memory
    mov     rdx,GENERIC_READ
    mov     r8,FILE_SHARE_READ
    xor     r9d,r9d
    mov     qword[rsp+4*8],OPEN_EXISTING
    mov     qword[rsp+5*8],0
    mov     qword[rsp+6*8],0
    call    [CreateFile]
    cmp     rax,INVALID_HANDLE_VALUE
    je      .end
    mov     qword[rsp+8*6],rax
    mov     rcx,rax
    mov     rdx,tempWORKER
    mov     r8,WORKER_SIZE
    mov     qword[rsp+8*4],0
    lea     r9,[rsp+8*5]
    call    [ReadFile]
    test    rax,rax
    jz      .end
    mov     rcx,qword[rsp+8*6]
    call    [CloseHandle]
    mov     rcx,tempWORKER
    mov     rdx,GENERIC_READ
    mov     r8,FILE_SHARE_READ
    lea     rcx,[rcx+WORKER.encPath]
    xor     r9d,r9d
    mov     qword[rsp+4*8],OPEN_EXISTING
    mov     qword[rsp+5*8],0
    mov     qword[rsp+6*8],0
    call    [CreateFile]
    cmp     rax,INVALID_HANDLE_VALUE
    je      .end
    mov     rcx,rax
    call    [CloseHandle]
    ;;;test for duplicates
    mov     rcx,qword[lvhandle]
    mov     rdx,LVM_FINDITEM
    mov     r8,-1
    mov     r9,LVFINDINFO
    call    [SendMessage]
    cmp     rax,-1
    jne     .end
    ;;; add the key file name
    mov     dword[LVITEMiSubItem],0
    mov     rcx,qword[lvhandle]
    mov     edx,LVM_INSERTITEM
    mov     r8d,0
    mov     r9,LVADD
    call    [SendMessage]
```

```
    call    UpdateCount
    ;;; add the 'file to decrypt' name
    mov     rax,tempWORKER
    mov     dword[LVITEMiSubItem],1
    lea     rax,[rax+WORKER.encPath]
    mov     rcx,qword[lvhandle]
    mov     edx,LVM_SETITEM
    mov     qword[LVITEMpszText],rax
    mov     r8d,0
    mov     r9,LVADD
    call    [SendMessage]
  .end:
    add     rsp,8*7
    ret     0
;;;input rcx=addr of string rdx=item# r8=subitem#
SetItem:
    sub     rsp,8*7
    mov     qword[LVITEMpszText],rcx
    mov     dword[LVITEMiItem],edx
    mov     dword[LVITEMiSubItem],r8d
    mov     rcx,qword[lvhandle]
    mov     edx,LVM_SETITEM
    mov     r8d,0
    mov     r9,LVADD
    call    [SendMessage]
    add     rsp,8*7
    ret     0
;;;****************************************
UpdateCount:
    sub     rsp,8*7
    xor     r8d,r8d
    mov     rcx,qword[lvhandle]
    mov     edx,LVM_GETITEMCOUNT
    mov     r9,r8
    call    [SendMessage]
    mov     rcx,rax
    mov     rdx,countbuffer
    call    IntToString
    mov     r9,countbuffer
    xor     r8d,r8d
    mov     edx,WM_SETTEXT
    mov     rcx,qword[counthandle]
    call    [SendMessage]
    add     rsp,8*7
    ret     0
;;;****************************************
;;;Adds user selected files to the list view
GetFiles:
    pxor    xmm0,xmm0
    mov     [ofn.lStructSize],sizeof.OPENFILENAME
    xor     eax,eax
    mov     [ofn.hwndOwner],rax
    mov     qword[ofn.hInstance],0
    mov     qword[ofn.lpstrCustomFilter],0
    mov     [ofn.nFilterIndex],1
    mov     [ofn.nMaxFile],FILE_BUFFER_SIZE
```

```
    mov     rax,[name_buffer]
    mov     qword[ofn.lpstrFileTitle],rax
    movdqa  dqword[rax],xmm0
    mov     [ofn.nMaxFileTitle],FILE_BUFFER_SIZE
    mov     qword[ofn.lpstrInitialDir],0
    mov     qword[ofn.lpstrDefExt],0
    mov     rax,[path_buffer]
    mov     qword[ofn.lpstrFile],rax
    mov     ecx,FILE_BUFFER_SIZE-64
  .clear:
    movdqa  dqword[rax+rcx],xmm0
    movdqa  dqword[rax+rcx+16],xmm0
    movdqa  dqword[rax+rcx+32],xmm0
    movdqa  dqword[rax+rcx+48],xmm0
    sub     rcx,64
    jns     .clear
    mov     rax,open_filter
    mov     [ofn.lpstrFilter],rax
    mov     [ofn.Flags],OFN_OPEN_FLAGS
    mov     rax,[name_buffer]
    mov     [ofn.lpstrFileTitle],rax
    mov     [ofn.lpstrTitle],0
    mov     rcx,ofn
    call    [GetOpenFileName]
    test    rax,rax
    jz      .end
    movzx   rcx,word[ofn.nFileOffset]
    mov     rax,[path_buffer]
    test    byte[rax+rcx-1],0FFh
    jnz     .justone
    mov     rbx,rax
    lea     rsi,[rbx+rcx]
    mov     rdi,rsi
    mov     byte[rsi-1],'\'
  .loopFileNames:
    mov     rcx,rbx
    xor     edx,edx
    call    AddItem
    cmp     word[rsi-1],0
    je      .end
    mov     rcx,rdi
  .lstrcat:
    movzx   rax,byte[rsi]
    inc     rsi
    mov     byte[rcx],al
    inc     rcx
    test    al,al
    jnz     .lstrcat
    jmp     .loopFileNames
  .justone:
    mov     rcx,rax
    xor     edx,edx
    call    AddItem
  .end:
    jmp     processed
;;;******************************
```

```
ClearListView:
      xor     ecx,ecx
      mov     r8,MsgBoxCap
      mov     rdx,MsgBoxClear
      mov     r9d,MB_YESNO
      call    [MessageBox]
      cmp     eax,IDYES
      jne     .end
      mov     edx,LVM_DELETEALLITEMS
      mov     rcx,qword[lvhandle]
      call    [SendMessage]
      call    UpdateCount
  .end:
      jmp     processed
;;;*************************************
RemoveItem:
      mov     rcx,qword[lvhandle]
      mov     edx,LVM_GETSELECTIONMARK
      call    [SendMessage]
      cmp     eax,-1
      je      .end
      xor     r9d,r9d
      mov     r8,rax
      mov     edx,LVM_DELETEITEM
      mov     rcx,qword[lvhandle]
      call    [SendMessage]
      mov     rcx,qword[lvhandle]
      mov     edx,LVM_SETSELECTIONMARK
      mov     r9,-1
      call    [SendMessage]
      call    UpdateCount
  .end:
      jmp     processed

;;; rbx = file counter
StartDecryption:
;;;reset
      mov     dword[CompletionCount],0
      mov     dword[ErrorCount],0
;;;error checks
      xor     r8d,r8d
      mov     rcx,qword[lvhandle]
      mov     edx,LVM_GETITEMCOUNT
      mov     r9,r8
      call    [SendMessage]
      mov     rdx,MsgBoxFiles
      test    rax,rax
      jz      .error
      mov     [filecount],rax
;;;
      mov     rcx,qword[filecount]
      call    StackInit
      test    rax,rax
      jz      .error
      mov     rbx,qword[filecount]
      dec     rbx
```

```
  .loop:
;;; allocate memory for 1 stack node
      mov     rcx,WORKER_SIZE
      call    AllocateOrDie
      mov     rsi,rax
;;;rest of WORKER structure
      mov     dword[lvstring.iItem],ebx
      mov     rax,[lv_buffer]
      mov     dword[lvstring.iSubItem],0
      mov     dword[lvstring.cchTextMax],FILE_BUFFER_SIZE
      mov     qword[lvstring.pszText],rax
      mov     rcx,qword[lvhandle]
      mov     rdx,LVM_GETITEMTEXT
      mov     r8,rbx
      lea     r9,[lvstring]
      call    [SendMessage]
      test    rax,rax
      jz      .error
      mov     rcx,[lv_buffer]
      mov     rdx,GENERIC_READ
      mov     r8,FILE_SHARE_READ
      xor     r9d,r9d
      mov     qword[rsp+4*8],OPEN_EXISTING
      mov     qword[rsp+5*8],0
      mov     qword[rsp+6*8],0
      call    [CreateFile]
      cmp     rax,INVALID_HANDLE_VALUE
      je      .error
      mov     rdi,rax
      mov     rcx,rax
      lea     rdx,[rsi]
      mov     r8,WORKER_SIZE
      mov     qword[rsp+8*4],0
      lea     r9,[rsp+8*5]
      call    [ReadFile]
      test    rax,rax
      jz      .error
      mov     rcx,rdi
      call    [CloseHandle]
;;;inFile
      mov     dword[lvstring.iItem],ebx
      mov     rax,[lv_buffer]
      mov     dword[lvstring.iSubItem],1
      mov     dword[lvstring.cchTextMax],FILE_BUFFER_SIZE
      mov     qword[lvstring.pszText],rax
      mov     rcx,qword[lvhandle]
      mov     rdx,LVM_GETITEMTEXT
      mov     r8,rbx
      lea     r9,[lvstring]
      call    [SendMessage]
      test    rax,rax
      jz      .error
      mov     rcx,[lv_buffer]
      mov     rdx,GENERIC_READ
      mov     r8,FILE_SHARE_READ
      xor     r9d,r9d
```

```
    mov     qword[rsp+4*8],OPEN_EXISTING
    mov     qword[rsp+5*8],FILE_FLAG_SEQUENTIAL_SCAN
    mov     qword[rsp+6*8],0
    call    [CreateFile]
    cmp     rax,INVALID_HANDLE_VALUE
    je      .error
    mov     rdx,[rsi+WORKER.inSize]
    mov     [rsi+WORKER.outSize],rdx
    mov     qword[rsi+WORKER.inFile],rax
    mov     rcx,rax
    lea     rdx,[rsi+WORKER.inSize]
    call    [GetFileSizeEx]
;;;outFile
    mov     rax,[lv_buffer]
    xor     ecx,ecx
  .removeExt:
    inc     rax
    test    byte[rax],0FFh
    jnz     .removeExt
    ;;; .proe is 5 bytes
    mov     dword[rax-5],ecx ;;;null out the last extension
    mov     rcx,[lv_buffer]
    mov     rdx,GENERIC_WRITE
    mov     r8,0
    xor     r9d,r9d
    mov     qword[rsp+4*8],CREATE_ALWAYS
    mov     qword[rsp+5*8],0
    mov     qword[rsp+6*8],0
    call    [CreateFile]
    cmp     rax,INVALID_HANDLE_VALUE
    je      .error
    mov     qword[rsi+WORKER.outFile],rax
;;;push
    mov     rcx,rsi
    call    StackPush
;;;
    dec     rbx
    jns     .loop
;;;
    call    [GetTickCount]
    mov     dword[starttime],eax
    mov     qword[bytecount],0
;;;create status thread and the worker threads
    mov     ebx,dword[processorcount]
    xor     ecx,ecx
    lea     r8,[ThreadStatus]
    mov     r9,rbx
    mov     rdx,rcx
    mov     qword[rsp+8*4],rcx
    mov     qword[rsp+8*5],rcx
    call    [CreateThread]
    test    rax,rax
    jz      .error
    mov     rcx,rax
    call    [CloseHandle]
  .loopworker:
```

```
        xor     ecx,ecx
        lea     r8,[ThreadWorker]
        mov     r9,rcx
        mov     rdx,rcx
        mov     qword[rsp+8*4],rcx
        mov     qword[rsp+8*5],rcx
        call    [CreateThread]
        test    rax,rax
        jz      .error
        mov     rcx,rax
        call    [CloseHandle]
        dec     ebx
        jnz     .loopworker
        jmp     processed
   .error:
        xor     ecx,ecx
        mov     r8,MsgBoxCap
        mov     r9,rcx
        call    [MessageBox]
        jmp     processed

;;;Updates the status information for the GUI
;;;input rcx = number of threads
ThreadStatus:
        mov     rbx,rcx ;;;save number of threads
        mov     rsi,[statushandle]
        mov     rdi,[etimehandle]
        mov     r12,CompletionCount
        mov     r13,ErrorCount
        mov     r14,StatusBuffer
        sub     rsp,8*5
   .loop:
        mov     rcx,STATUS_SLEEP
        xor     edx,edx
        call    [SleepEx] ;;;Sleep 1 second
;;;
        mov     rcx,r14
        mov     rdx,StrStatus1
        mov     r8d,dword[r13]
        call    [wsprintf]
        mov     rcx,rsi
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        call    [GetTickCount]
        sub     eax,dword[starttime]
        xor     edx,edx
        mov     ecx,1000
        div     ecx
        add     eax,1
        mov     rcx,r14
        mov     rdx,StrEtime1
        mov     r8d,eax
        call    [wsprintf]
        mov     rcx,rdi
```

```
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        cmp     ebx,dword[r12]
        jg      .loop
   .lastoutput:
        mov     rcx,r14
        mov     rdx,StrStatus2
        mov     r8d,dword[bytecount]
        mov     r9d,dword[r13]
        call    [wsprintf]
        mov     rcx,rsi
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        mov     eax,dword[endtime]
        sub     eax,dword[starttime]
        xor     edx,edx
        mov     ecx,1000
        div     ecx
        add     eax,1
        mov     r8d,eax
        mov     rax,qword[bytecount]
        xor     edx,edx
        mov     rcx,r8
        div     rcx
        mov     rcx,r14
        mov     rdx,StrEtime2
        mov     r9d,eax
        call    [wsprintf]
        mov     rcx,rdi
        mov     edx,WM_SETTEXT
        xor     r8d,r8d
        mov     r9,r14
        call    [SendMessage]
        add     rsp,8*5
        call    [ExitThread]
        ret     0

;;;PROE Decrypt GUI Data

align 16
tempWORKER WORKER

;;;Window component handles
handle dq 0
lvhandle dq 0
addhandle dq 0
removehandle dq 0
clearhandle dq 0
counthandle dq 0
statushandle dq 0
etimehandle  dq 0
```

```
;;;memory handles
path_buffer dq 0
name_buffer dq 0
lv_buffer   dq 0
browse_buffer dq 0

;;;OPENFILENAME struc
align 16
ofn OPENFILENAME
open_filter  db 'Key files',0,'*.prok',0,0
align 16
browseinfo dq 0,0
        browse_dir dq 0,0,0,0,0,0
;;;Extra Randomness
MsgBoxMouse db 'Please move your mouse sporadically around the screen for the a few seconds. This will help
make your Encryption Key(s) be more secure. Press OK when you are done',0
MsgBoxMouse2 db 'Your Encyrption Keys will now be created.',0
UtilsDll db 'MouseHook.DLL',0

;;;GLOBAL

MsgBoxCap db 'PROE (Psuedo Random Optimized Encryption)',0
MsgBoxError db 'An error occurred, you need more memory to perform the operation.',0
NumProcessors dd 0

;;;DEBUG
fmthh    db '%X %X ',0
fmtf     db '%f ',0

  flags dd ?,?
  caption rb 40h
  message rb 100h


countbuffer db 16 dup(0)
LV_HEADER1 db 'Key File Name',0
LV_HEADER2 db 'File To Decrypt',0
LVCOLUMN_1:
   dd LV_COL_MASK
   dd 0
   dd LV_COL_WIDTH
   dd     0
   LVCOLUMNpszText dq 0
   LVCOLUMNcchTextMax dd 0
   dq 0,0,0,0,0,0,0

LVADD:
   dd LVIF_TEXT
   LVITEMiItem dd 0
   LVITEMiSubItem dd 0
   dd 0
   dd 0
   dd   0
   LVITEMpszText dq 0
   LVITEMcchTextMax dd 0
   dq 0,0,0,0,0
```

```
LVFINDINFO:
   dd LVFI_STRING
   dd 0
   LVFINDINFOpsz dq 0
   dq 0,0,0

MsgBoxClear db 'Do you want to clear the file list?',0

starttime dd 0
endtime   dd 0
bytecount dq 0

;;;Encyrption Start
MsgBoxKeyDir db 'Please choose a directory to save the encryption keys to. It is recommended you save them to a
removable disk (ie: USB thumbdrive).',0
passescount db 0,0
MsgBoxPasses db 'Please be sure the number of passes is between 2 and 9.',0
deletecheck db 0
MsgBoxDelete db 'If you do not delete the original files your data will remain insecure.',0
filecount dq 0
MsgBoxFiles db 'You need to select at least one file to decrypt.',0
lvstring LV_ITEM
out_file_ext db '.proe',0
enc_file_ext  db '.prok',0

StrStatus1    db 'Status: Decrypting (%lu Error(s) Encountered).',0
StrStatus2    db 'Status: Finished, %lu byte(s) decrypted (%lu Error(s) Encountered).',0
StrEtime1     db 'Elapsed Time: %lu second(s)',0
StrEtime2     db 'Elapsed Time: %lu second(s) (Avg Speed %lu Byte(s)/Second).',0
StatusBuffer  rb 255

;;;PROE Decrypt GUI Equates

STATUS_SLEEP equ 1000

FILE_BUFFER_SIZE equ 100000h ;;;1MB
;;;*********************************************
;;;OPEN FILE
OFN_FORCESHOWHIDDEN     equ 10000000h
OFN_OPEN_FLAGS equ OFN_EXPLORER or OFN_ALLOWMULTISELECT or
OFN_NOREADONLYRETURN or OFN_HIDEREADONLY or OFN_FORCESHOWHIDDEN or
OFN_FILEMUSTEXIST

;;;****************************************
;;;List View
  LVS_REPORT = 0x1
  LVS_ICON = 0x0
  LVS_REPORT = 0x1
  LVS_SMALLICON = 0x2
  LVS_LIST = 0x3
  LVS_TYPEMASK = 0x3
  LVS_SINGLESEL = 0x4
  LVS_SHOWSELALWAYS = 0x8
  LVS_SORTASCENDING = 0x10
  LVS_SORTDESCENDING = 0x20
```

LVS_SHAREIMAGELISTS = 0x40
LVS_NOLABELWRAP = 0x80
LVS_AUTOARRANGE = 0x100
LVS_EDITLABELS = 0x200

LVM_FIRST = 0x1000
LVM_GETHEADER = (LVM_FIRST + 31)
LVM_GETBKCOLOR = (LVM_FIRST + 0)
LVM_SETBKCOLOR = (LVM_FIRST + 1)
LVM_GETIMAGELIST = (LVM_FIRST + 2)
LVM_SETIMAGELIST = (LVM_FIRST + 3)
LVM_GETITEMCOUNT = (LVM_FIRST + 4)
LVM_GETITEMA = (LVM_FIRST + 5)
LVM_GETITEM = LVM_GETITEMA
LVM_SETITEMA = (LVM_FIRST + 6)
LVM_SETITEM = LVM_SETITEMA
LVM_INSERTITEMA = (LVM_FIRST + 7)
LVM_INSERTITEM = LVM_INSERTITEMA
LVM_DELETEITEM = (LVM_FIRST + 8)
LVM_DELETEALLITEMS = (LVM_FIRST + 9)
LVM_GETCALLBACKMASK = (LVM_FIRST + 10)
LVM_SETCALLBACKMASK = (LVM_FIRST + 11)
LVM_GETNEXTITEM = (LVM_FIRST + 12)
LVM_FINDITEMA = (LVM_FIRST + 13)
LVM_FINDITEM = LVM_FINDITEMA
LVM_GETITEMRECT = (LVM_FIRST + 14)
LVM_SETITEMPOSITION = (LVM_FIRST + 15)
LVM_GETITEMPOSITION = (LVM_FIRST + 16)
LVM_GETSTRINGWIDTHA = (LVM_FIRST + 17)
LVM_GETSTRINGWIDTH = LVM_GETSTRINGWIDTHA
LVM_HITTEST = (LVM_FIRST + 18)
LVM_ENSUREVISIBLE = (LVM_FIRST + 19)
LVM_SCROLL = (LVM_FIRST + 20)
LVM_REDRAWITEMS = (LVM_FIRST + 21)
LVM_ARRANGE = (LVM_FIRST + 22)
LVM_EDITLABELA = (LVM_FIRST + 23)
LVM_EDITLABEL = LVM_EDITLABELA
LVM_GETEDITCONTROL = (LVM_FIRST + 24)
LVM_GETCOLUMNA = (LVM_FIRST + 25)
LVM_GETCOLUMN = LVM_GETCOLUMNA
LVM_SETCOLUMNA = (LVM_FIRST + 26)
LVM_SETCOLUMN = LVM_SETCOLUMNA
LVM_INSERTCOLUMNA = (LVM_FIRST + 27)
LVM_INSERTCOLUMN = LVM_INSERTCOLUMNA
LVM_DELETECOLUMN = (LVM_FIRST + 28)
LVM_GETCOLUMNWIDTH = (LVM_FIRST + 29)
LVM_SETCOLUMNWIDTH = (LVM_FIRST + 30)
LVM_CREATEDRAGIMAGE = (LVM_FIRST + 33)
LVM_GETVIEWRECT = (LVM_FIRST + 34)
LVM_GETTEXTCOLOR = (LVM_FIRST + 35)
LVM_SETTEXTCOLOR = (LVM_FIRST + 36)
LVM_GETTEXTBKCOLOR = (LVM_FIRST + 37)
LVM_SETTEXTBKCOLOR = (LVM_FIRST + 38)
LVM_GETTOPINDEX = (LVM_FIRST + 39)
LVM_GETCOUNTPERPAGE = (LVM_FIRST + 40)
LVM_GETORIGIN = (LVM_FIRST + 41)

```
  LVM_UPDATE = (LVM_FIRST + 42)
  LVM_SETITEMSTATE = (LVM_FIRST + 43)
  LVM_GETITEMSTATE = (LVM_FIRST + 44)
  LVM_GETITEMTEXTA = (LVM_FIRST + 45)
  LVM_GETITEMTEXT = LVM_GETITEMTEXTA
  LVM_SETITEMTEXTA = (LVM_FIRST + 46)
  LVM_SETITEMTEXT = LVM_SETITEMTEXTA
  LVM_SETITEMCOUNT = (LVM_FIRST + 47)
  LVM_SORTITEMS = (LVM_FIRST + 48)
  LVM_SETITEMPOSITION32 = (LVM_FIRST + 49)
  LVM_GETSELECTEDCOUNT = (LVM_FIRST + 50)
  LVM_GETITEMSPACING = (LVM_FIRST + 51)
  LVM_GETISEARCHSTRINGA = (LVM_FIRST + 52)
  LVM_GETISEARCHSTRING = LVM_GETISEARCHSTRINGA
  LVM_SETICONSPACING = (LVM_FIRST + 53)
  LVM_SETEXTENDEDLISTVIEWSTYLE = (LVM_FIRST + 54)
  LVM_GETEXTENDEDLISTVIEWSTYLE = (LVM_FIRST + 55)
  LVM_GETSUBITEMRECT = (LVM_FIRST + 56)
  LVM_SUBITEMHITTEST = (LVM_FIRST + 57)
  LVM_SETCOLUMNORDERARRAY = (LVM_FIRST + 58)
  LVM_GETCOLUMNORDERARRAY = (LVM_FIRST + 59)
  LVM_SETHOTITEM = (LVM_FIRST + 60)
  LVM_GETHOTITEM = (LVM_FIRST + 61)
  LVM_SETHOTCURSOR = (LVM_FIRST + 62)
  LVM_GETHOTCURSOR = (LVM_FIRST + 63)
  LVM_APPROXIMATEVIEWRECT = (LVM_FIRST + 64)
  LVM_GETSELECTIONMARK = (LVM_FIRST + 66)
  LVM_SETSELECTIONMARK = (LVM_FIRST + 67)
  LVS_EX_FULLROWSELECT = 0x20
  LVSCW_AUTOSIZE  = -1
  LVSCW_AUTOSIZE_USEHEADER = -2

  LVIF_TEXT = 1
  LVFI_STRING = 2

  LVCF_TEXT = 4
  LVCF_WIDTH = 2
  LV_COL_MASK = LVCF_WIDTH or LVCF_TEXT

  LV_COL_WIDTH = 285

struct LVCOLUMN
   mask  dd ?
   fmt   dd ?
   cx    dd ?
   pszText dq ?
   cchTextMax dd ?
   iSubItem   dq ?
   iImage dd ?
   iOrder dd ?
   cxMin  dd ?
   cxDefault dd ?
   cxIdeal dd ?
ends

struct LVITEM
```

```
    mask dd ?
    iItem dd ?
    iSubItem dd ?
    state dd ?
    stateMask dd ?;
    pszText dq ?
    cchTextMax dd ?
    iImage dd ?
    lParam dd ?
    iIndent dd ?
    iGroupId dd ?
    cColumns dd ?
    puColumns dq ?
    piColFmt dq ?
    iGroup dd ?
ends
```