FLAXOR: A Symmetric-Key Block Cipher Plug-in using USB Mass Storage Device for Extracted Block Storage

Cristopher Ian S. Uy [1,2], Joseff Anthony L. Angeles [1], Nikolai A. Flores [1], Ariel Kelly D. Balan [1], Jose F. Abisado [1]
[1] School of Information Technology
[1] Mapúa Institute of Technology, Makati City, Philippines
[2] Email: iansuy@gmail.com

Passwords have already outlived their usefulness as a serious security device. Current block cipher design relies on purely reversible mathematical procedures to obstruct the plaintext using a cryptographic key. As time progress, cryptanalytic attacks against these block ciphers gets more complicated and more effective. In this study, we propose a plug-in for existing encryption algorithm using a new encryption method called "Block Extraction" where blocks of ciphertext are chained, extracted and then stored into a USB Mass Storage device. The new encryption method is supplemented by an innovative key generation technique wherein a Two-factor authentication using a USB Mass Storage device that acts as a security token and a password was used. The proposed Block Extraction method is based on the fact that cryptanalysis on an incomplete copy of the ciphertext will not be able to produce the original and complete copy of the plaintext. The technical feasibility of the proposed system has been tested by simulating a brute-force attack to exhaustively search or produce the missing block of the ciphertext. The test showed that the proposed Block Extraction method is $1.73\ x\ 10^{19,653}\%$ more secure than the leading encryption algorithm. Also, the runtime performance of the Block Extraction method has been analyzed, measured and compared to other existing algorithms. The analysis shows that the Block Extraction method is at least 500% faster than the leading encryption algorithm. In conclusion, the Block Extraction method is found to be highly effective when used in a Personal File Encryption utility.

Keywords**:** block extraction, cryptography, flaxor, personal file encryption, security token, two-factor authentication

Introduction

Block Ciphers works by dividing the plaintext into separate blocks of fixed size (e.g., 64 or 128 bits), and encrypts each of them independently using the same key-dependent transformation. Generally speaking, a block cipher takes n-bits of plaintext as input and output n-bits of ciphertext [1]. This means that the ciphertext still contains the same amount of data or message length as the plaintext – but obfuscated or encrypted. Cryptanalysis works by analyzing the output bits of a cipher to establish a pattern. Since the amount of data available in the ciphertext is greater than or equal to the amount of data in the plaintext, the cryptanalyst will just need to break the cipher to acquire the plaintext.

In Symmetric Key Cryptography, a single key is used in both encryption and decryption process. The key or the pass phrase is the only basis of the encryption, the alterations made on the ciphertext is based on the pass phrase. Brute force attack works by defeating a cryptographic scheme by trying a large number of possibilities; for example, exhaustively working through all possible keys in order to decrypt a message. By trying all the keys in the keyspace, the attacker will definitely be able to break the cipher when the right key is found because the amount of data found in the ciphertext is the same as the amount of data contained in the plaintext.

The proponents had identified two major problems in symmetric-key block cipher design. The first problem is related to the block cipher design wherein the plaintext and the ciphertext still contain the same amount of data or the same message length even after the encryption process has taken effect thus rendering the ciphertext susceptible to cryptanalysis and brute force attack. The second problem is related to the symmetric-key design wherein most current encryption algorithms only require the user to supply a pass phrase or password to encrypt or decrypt a file. According to Schneier, passwords have already outlived their usefulness as a serious security device. Schneier also stresses the fact that security by password alone is pretty risky [2].

The primary objective of this study is to construct a set of algorithms that will act as a plug-in to existing block ciphers. The said plug-in will implement a new encryption process called "Block Extraction" where blocks or parts of the ciphertext will be isolated or stored into an existing removable media such as the USB Mass Storage Device. The "Block Extraction" process will add another layer of security on top of an existing encryption algorithm like the

Advanced Encryption Standard or AES-256 to dramatically increase its resistance against cryptanalysis and brute force attacks. The secondary objective of this study is to present an innovative way of key generation wherein a pass phrase, in addition of a tangible or a physical key, is required to encrypt or decrypt a file. In particular, the tangible key that would be used is a USB Mass Storage device. The USB Mass Storage device shall act as a security token that will represent the first authentication factor while the user-supplied password will represent the second authentication factor thereby completing the Two-factor authentication.

The study produced a Personal File Encryption that could encrypt or decrypt any file type thru the use of an existing encryption standard like the AES-256 supplemented by the Flaxorization Process, the Key Generation process and the Block Extraction Module. By providing another layer of security thru block extraction, the cryptosystem was made more resistant to cryptanalysis and brute force attacks. This was the result of the design of the block extraction module wherein a Binary-Tree structure was utilized to deliver the maximum "Avalanche Effect" when a node (block of ciphertext) was removed or extracted. For instance, the cryptanalyst was able to break the cipher but the recovered plaintext or message will still deliver a ciphertext. Additionally, the requirement of a tangible key in the key generation module would make the key management more secure. As previously stated, the tangible key that was used is a USB Mass Storage Device. Thru research, the proponents were able to extract certain device descriptors from the USB Mass Storage devices. As a result, the Device Descriptors were used as a supplement to the pass phrase when generating the 256-bit key. If the pass phrase is lost, stolen or cracked by a brute force attack, it would not totally compromise the security of the file. The encryption process is shown in Figure 1 while the decryption process follows at Figure 2.

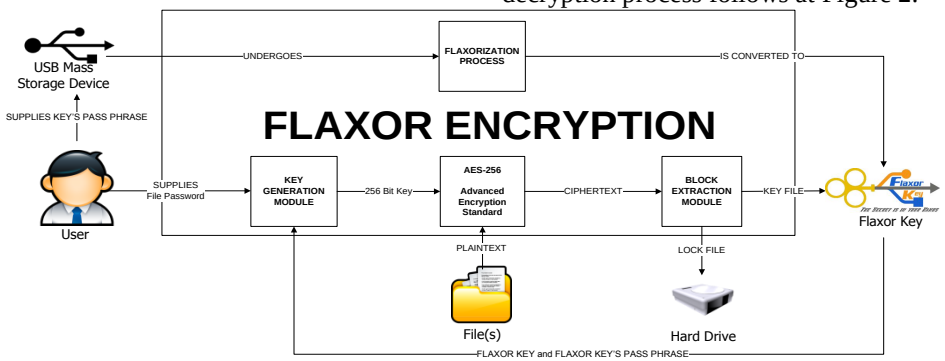

Figure 1 Theoretical Model of the Flaxor Encryption

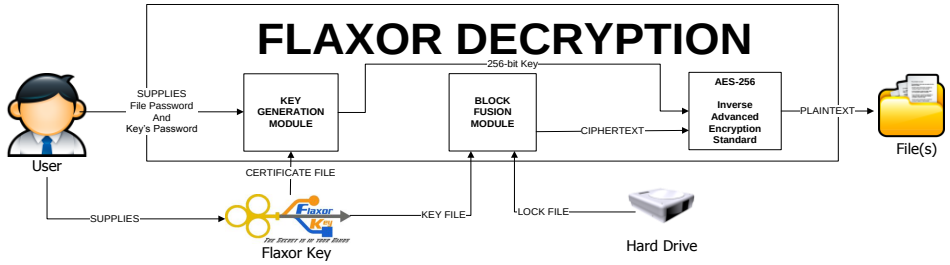

Figure 2 Theoretical Model of the Flaxor Decryption

Before undergoing the encryption process, a compression process was applied to the file(s) to produce a single compressed output file that will be fed to the encryption process. As previously stated, the encryption process used the current AES-256 bit Cipher as its foundation and was interfaced with the Block Extraction Module. The Block Extraction Module utilized a Binary Search Tree and a random node extraction process. Due to the nature of the Binary Search Trees, an Incremental Mapping procedure was used to provide a unique code for each node, the unique code was mapped to a block of ciphertext. Each Node contains 8,192 bytes of ciphered data. The extracted nodes were then placed as a file to the USB Mass Storage Device now termed as the "Flaxor Key". In order to encrypt or decrypt a file, a Flaxor Key was needed to be plugged into the computer. A "Flaxorization" process is needed to convert a normal USB Mass Storage Device to a Flaxor Key. The Flaxorization process reads the

unique device descriptors present on all USB Mass Storage devices and a user-supplied password for the Flaxor Key, all of these variables was fed to the SHA-256 Hash Generator and produced a unique 256-bit message digest. The message digest was then converted to a base64 string and then written as a "Certificate File" in the root directory of the USB Mass Storage Device. The 256-bit key generation process reads the Certificate File from the Flaxor Key and the user-supplied salted pass phrase for the file(s) to be encrypted in order to produce a unique 256-bit key that was fed to the AES-256 cipher. The salt for the pass phrase was then stored in the Flaxor Key via the Key File. In essence, the user supplied the pass phrase for the file, the pass phrase for the Flaxor Key and the Flaxor Key itself in order to encrypt or decrypt a file.

FLAXOR ALGORITHM DESIGN

Flaxorization Module

The Flaxorization Module or FM is responsible for the *one-time* "conversion" of an ordinary USB Mass Storage Device to a "Flaxor Key". A Flaxor Key is a USB Mass Storage device that acts as a tangible key and storage for the extracted blocks of chained ciphertext (Key File). The conversion of a USB Mass Storage Device to a Flaxor Key will not affect its normal operation. The Flaxorization Module is illustrated in Figure 3.

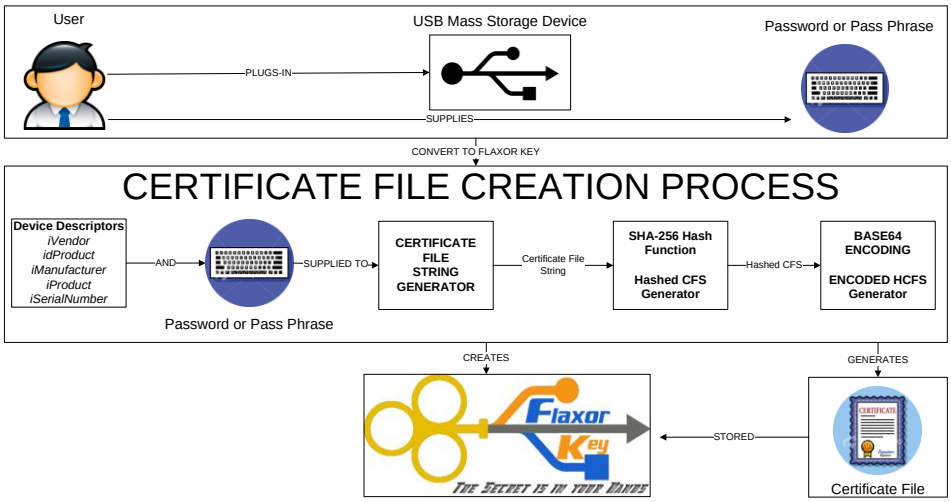

Figure 3 The Flaxorization Module

The FM will extract certain device descriptors from the USB Mass Storage Device (figure 3) thru the use of an Application Programming Interface or API that is provided by Microsoft Windows. The device descriptors that will be extracted are the iVendor, idProduct, iManufacturer, iProduct and iSerialNumber.

The Certificate File Creation Process or CFCP will require the end-user to supply a pass phrase or a password for the Flaxor Key. The CFCP will concatenate the device descriptors and the pass phrase together using the CFCP String to produce a unique Certificate File String or CFS.

Afterwards, the CF String will be fed to a SHA-256 Hash Function to produce a unique 256-bit string called a Hashed CFS or HCFS.

Finally, the produced 256-bit hash or HCFS will undergo a Base64-Encoding transform.

This 256-bit Base64 Encoded string called the Encoded HCFS or EHCFS will be written as a file on the root directory of the USB Mass Storage Device with the file name "FlaxorKey.crt".

Key Generation Module

The goal of the Key Generation Module or KGM is to generate a unique 256-bit cryptographic key that will be fed to the AES-256 Cipher. The KGM will require the end-user to supply a Flaxor Key, Flaxor Key's Password and the Password of the File to be encrypted. The Key Generation Module is illustrated in Figure 4.

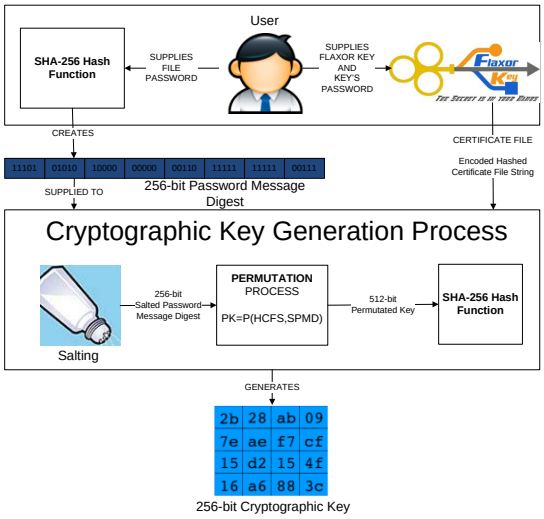

Figure 4 The Key Generation Module

The KGPM is a Four-step process. First, a pass phrase or a password for the file(s) to be encrypted will need to be supplied by the user. The supplied password will then be hashed by the SHA-256 Hash Function to produce a 256-bit Password Message Digest or PMD.

Second, the PMD that was generated on the first process will then be salted. The Salting process will produce a 256-bit random salt by using .NET's RNGCryptoServiceProvider. The 256-bit salt will then be XOR'ed to the PMD to produce a Salted Password Message Digest or SPMD.

To be able to reproduce the same random salt on the decryption process, the 256-bit random salt will be placed in the Key File that is stored in the Flaxor Key.

Third, the EHCF string found in the Certificate File will be decoded to a 256-bit string (HCF) from a Base64 encoding. The PMD that was generated on the first process will then be permutated to the decoded EHCF (HCF) string to produce a permutated 512-bit key.

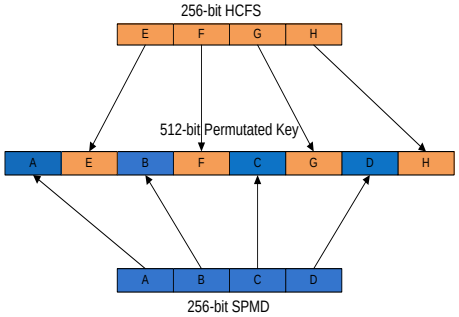

Figure 5 Permutation Process

Finally, the 512-bit permutated key will be fed into a SHA-256 Hash function to produce a unique 256-bit message digest. The 256-bit message digest will then be used as the cryptographic key for the AES-256 Cipher.

Block Extraction Module

The Block Extraction Module or BEM is what makes Flaxor much more secure. The BEM takes the ciphertext generated by the AES-256 as input and outputs two files: The Lock File which contains most of the blocks (90% - 99%) and The Key File which contains some of the blocks (10% - 1%). Both Lock and Key Files also contains metadata such as the SHA-1 hash of the input file and the remainder block.

The BEM is divided into six (6) stages that is executed one after the other. The six stages are Incremental Mapping, Node Insertion Queue, Binary Search Tree Formation, Bottom-Up Chaining, Random Node Extraction and Lock and Key Distribution

The first stage is the Incremental Mapping Stage. Its goal is to divide the ciphertext into chunks of 8 kilobytes (8,192 bytes) ciphered data and assign a unique key for each 8KB chunk. Each 8KB chunk is called a Node. After the chunking process, a Hash Table will be initialized.

After the initialization of the Incremental Mapping Array, each 8KB Node will then be assigned a unique incremental "key" consecutively until all nodes are given or assigned their own unique key. Once each node has a unique key, the key-node data will then be added to the Incremental Mapping Array. If there is a remainder node (Non-8KB Node), it will be placed on the end of the Remainder Pile. Figure 6 illustrates the mapping part of the Incremental Mapping Stage.

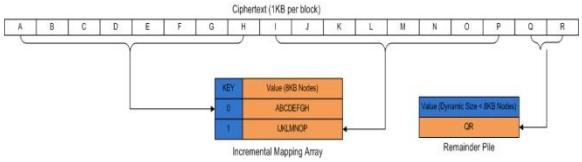

Figure 6 The Mapping Part of the Incremental Mapping Stage

The next part of the Incremental Mapping Stage is the Substitution-Permutation Portion (SP Portion). The SP Portion applies the initial encryption primitives to make sure that the "avalanche" effect will be fully executed when a cryptographic attack is applied. The Substitution method of the SP Portion makes use of Rijndael S-Box. The said method converts each byte of each block to the appropriate byte defined by the S-Box. Figure 7 illustrates the Substitution Method.

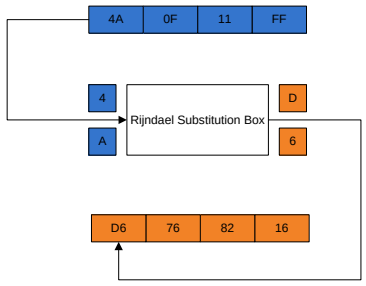

Figure 7 The Substitution Method of the Incremental Mapping Stage

Next, the Permutation Method of the Incremental Mapping Stage called the Horizontal Chaining will be executed. The Horizontal Chaining XORs each byte of each block to its neighboring byte; the last byte on each block is not XORed. Figure 8 illustrates the process.

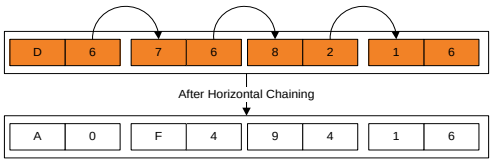

Figure 8 Horizontal Chaining Method

The second stage is the Node Insertion Queue. The goal of the Node Insertion Queue is to create an "order" that will instruct the Binary Search Tree on how to form itself. The Node Insertion Queue will be used as an "ordering" tool in which nodes are inserted into the binary search tree. The first step is to get the median of the incremental mapping array to produce a fairly balanced tree.

The key located at the median of the Incremental Mapping Array will then be placed as the first key in the Node Insertion Queue. The rest of the keys excluding the Median will be shuffled at random then added to the Node Insertion Queue. Figure 9 illustrates this method.

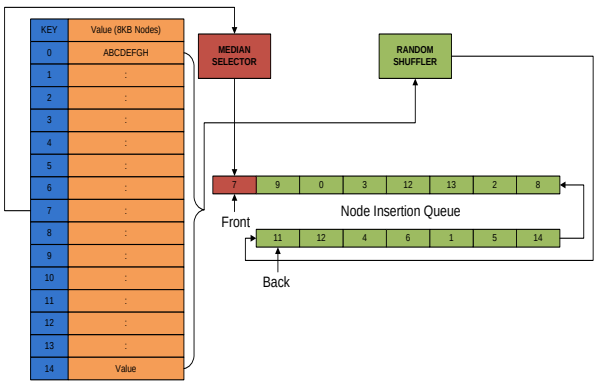

Figure 9 Node Insertion Queue Creation

The third stage is the Binary Search Tree Formation. The proponents chose the Binary Search Tree because it's an efficient data structure that provides the maximum "Avalanche Effect" when a node is removed or extracted. The goal of this stage is to construct a binary search tree depending on the order of the keys in the Node Insertion Queue. Figure 10 illustrates the Binary Search Tree Formation using the Node Insertion Queue.

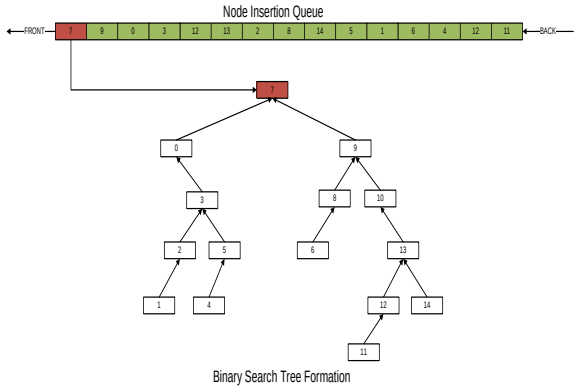

Figure 10 Binary Search Tree Formation using Node Insertion Queue

The fourth stage is the Bottom-Up Chaining Process or BUCP. The BUCP is vital to the Block Extraction Module because the BUCP is the one that actually gives the Block Extraction Module its "Avalanche Effect". The goal of the BUCP is to "chain" each node to their "parent" node. Chaining, in this effect, will mean that each child node's value (not the key) will be XOR'ed to their "parent" node's value. The result of the chaining will be stored in the Incremental Mapping Array. The BUCP will start at the left-most leaf going to the right, when all the leaves are XOR'ed, it will move one level up and so on. The root node would not be XOR'ed. Figure 11 illustrates the process.

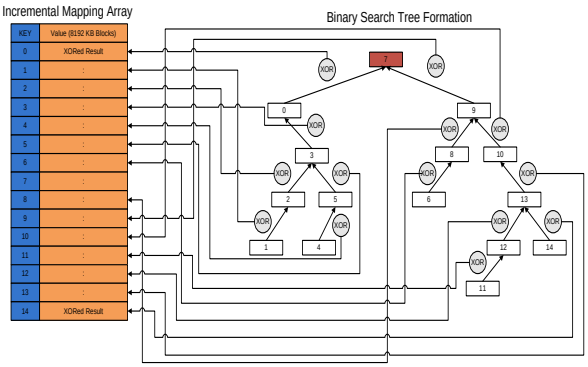

Figure 11 Bottom-Up Chaining Process and Value Storing

The fifth stage is the Random Node Extraction stage. The goal of this stage is to remove or extract

nodes at random and place them temporarily in an array. The root node will always be extracted. There are 3 different preset levels of security that is available for the end-user: Crazy which extracts 1% of the total nodes, Insane which extracts 5% of the total nodes and Paranoid which extracts 10% of the total nodes; however, the user may still choose a security level from 1 to 10.

The Random Node Extraction stage follows the same fashion on how nodes are removed on an ordinary binary search tree. For instance, the tree contains 15 nodes and the user selected Crazy as the security level, so a total of one (1) node will be extracted from the tree – which is the root node. If the user selected Paranoid, the total node that will be extracted is two (2) – that is, 1.5 nodes rounded-up. When a node is extracted, its value from the Incremental Mapping Array will be transferred to the Extracted Node Array (Hash Table). Figure 12 illustrates the Random Node Extraction stage.

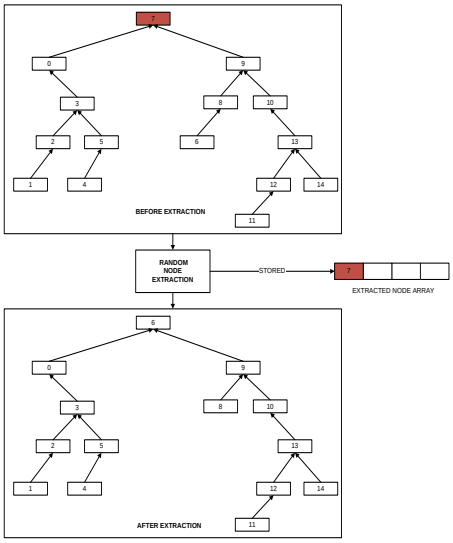

Figure 12 Random Node Extraction Stage

The sixth and final stage is the Lock and Key distribution. The main goal of this stage is to output the contents of the Incremental Mapping Array and the Extracted Node Array as files. The contents of the Incremental Mapping Array will be stored in the Lock File while the contents of the Extracted Node Array will be stored in the Key File. The Lock File will be written in the Hard Disk Drive while the Key File will be stored in the Flaxor Key inside the folder labeled as "Flaxor" in the root directory.

The Lock File will be labeled as "Filename.flk" where Filename is the name of the original file. If multiple file is encrypted, the user will need to give a Filename for the output file. Along with the Incremental Mapping Array, the Lock File will also contain the following metadata: Magic Number (FlaxorLock) and the Filename of the original file. Figure 13 shows the data structure of the Lock File.

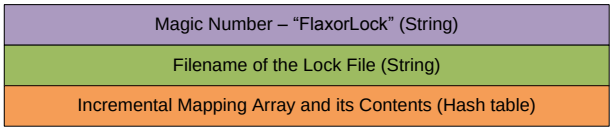

Figure 13 Data Structure of the Lock File

The Key File will be labeled as "Filename.fky" where Filename is the filename of the Lock File. The Key File primarily contains the Extracted Node Array along with the content of the Remainder Pile. Additionally, the metadata that are crucial for the proper decryption of the ciphertext is also contained in the key file. The said metadata are the following: Filesize of the input (Original Ciphertext), Node Insertion Queue, Random Salt used in the Key Generation Module and the SHA-1 Hash of the Original Ciphertext. The Filename of the Key File is also stored in the file. Finally, the magic number "FlaxorKey" is written in the header portion of the Key File. Figure 14 illustrates the data structure of the Key File.

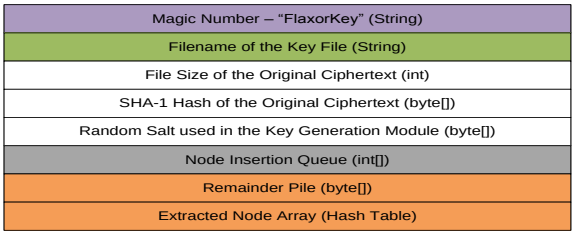

Figure 14 Data Structure of the Key File

The Serialized binary structure of both the Lock and Key File are then written to the hard disk and Flaxor Key, respectively.

RUNTIME ANALYSIS OF THE BLOCK EXTRACTION MODULE

To measure the time complexity of the Block Extraction Module, the proponents have conducted a runtime analysis of the Block Extraction Module. The Asymptotic Notation (Big O Notation) was used to describe how the size of the input data (Ciphertext input) affects the algorithm's usage of computational resources or running time.

To compute for the asymptotic notation of the Block Extraction Module, the proponents have divided the computations into the six (6) different stages of the BEM. To get the final time complexity of the BEM, the different Big O notation of the six stages will be combined. The combined time complexity of the BEM will then be compared to existing cryptographic algorithms.

Table 1 Runtime Performance of the different stages of the BEM

Stage	Asymptotic Name	Asymptotic Notation (Worst Case)
Incremental Mapping Stage	Cubic	$O(n^3)$
Node Insertion Queue Stage	Linear	$O(n)$
Binary Search Tree Formation Stage	Quadratic	$O(n^2)$
Bottom-Up Chaining Process Stage	Quadratic	$O(n^2)$
Random Node Extraction Stage	Logarithmic	$O(\log n)$
Lock and Key Distribution Stage	Linearithmic	$O(n \log n)$
Total Runtime performance of BEM	Cubic	$O(n^3)$

Table 2 Comparison of the runtime performance of the different cryptographic algorithms

Cryptographic Algorithm	Asymptotic Name	Asymptotic Notation (Worst Case)
DES	Quadratic	$O(n^2)$
Blowfish	Cubic	$O(n^3)$
Twofish	Quadratic	$O(n^2)$
Rijndael	Polynomial	$O(n^4)$
Block Extraction Module	Cubic	$O(n^3)$

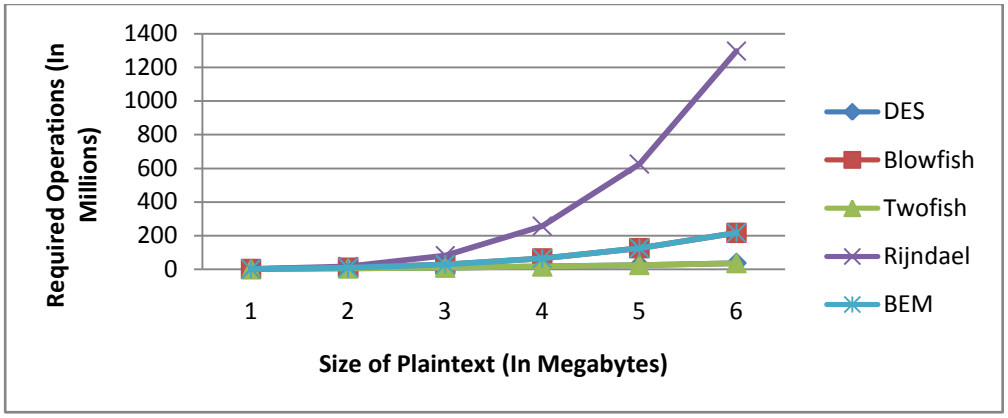

Figure 15 Asymptotic Growth of the different cryptographic algorithms with respect to the input size

Table 1 Required number of operations to be performed with respect to the input size.

Encryption Algorithms	Size of Plaintext					
	1 MB	2 MB	3 MB	4 MB	5 MB	6 MB
DES	1	4	9	16	25	36
Blowfish	1	8	27	64	125	216
Twofish	1	4	9	16	25	36
Rijndael	1	16	81	256	625	1296
BEM	1	8	27	64	125	216

As illustrated in Figure 15, DES and Twofish with a runtime performance of $O(n^2)$ grows slower than the rest of the cryptographic algorithms with respect to the input size. This means that DES and Twofish are the fastest among the cryptographic algorithms being compared in respect to the input size in bytes.

As revealed by Table 3, if a user encrypts a 6 megabyte file using DES, DES will only need to perform exactly 36 million operations.

On the other hand, Rijndael with a runtime performance of $O(n^4)$ grows fastest among the group. For instance, if a user encrypts a 6 megabyte file using Rijndael, it will first need to perform around 1.3 billion operations. Rijndael is the slowest algorithm among the group.

Finally, Flaxor's BEM and Blowfish with a runtime performance of $O(n^3)$ grew moderately with respect to the input size. For example, if a user encrypts a 6 megabyte file using BEM, it will only take around

200 million operations. This means that Flaxor's BEM is 83.33% slower than DES and Twofish but is 500% faster than Rijndael.

BRUTE-FORCE ATTACK ON THE BLOCK EXTRACTION MODULE

In this section, the proponents will discuss the possibility to break the cipher using a brute-force attack.

In order to simulate a brute-force attack, the proponents had assumed a scenario wherein the attacker has access to the Lock File and not the Key File (A modified ciphertext-only attack). It is also assumed that the plaintext is exactly 100 kilobytes in size and prior encryption was not applied to the plaintext. Additionally, the security level of the BEM was set to 1 (the lowest possible level), which means that 1% of the total blocks will be extracted. Also, it is assumed the Node Insertion Queue Order is already known. Finally, the Block Size is set to the normal size which is 8,192 bytes or 8 kilobytes. The said set-up was carefully chosen to simulate a worst-case scenario. A summary of the assumed set-up is illustrated in Table 4.

Table 3 Assumed Brute-Force attack set-up

Brute-Force Attack Assumed Set-up	
Plaintext File Type	Text File
Plaintext Size	100 Kilobytes or 102,400 bytes
Encrypted	No
Security Level	1
Block Size	8 Kilobytes or 8,192 bytes
Total Blocks	12
Remainder	Yes. 4,096 bytes.
Extracted Block(s)	1 block (The root node is always extracted)
NIQ Order Known	Yes

To calculate the total time needed to break BEM using the above set-up, the proponents had decided to calculate the crack time in two platforms: A Desktop Computer and A Grid Computer.

The rationale in using a desktop computer was to give a realistic value of how long it would take to extract the plaintext from the Lock File using a Brute Force attack. To remain faithful to the rationale, the proponents had decided to use a processor that is highly available on the market at a very affordable price.

On the other hand, the proponents had also considered the use of a "super computer" in order to break the BEM. However, finding a "super computer" or its specification has proven difficult; not to mention the price. In order to compensate, the proponents used a better alternative to a super computer – a grid computer. Grid computing is a form of distributed computing whereby a "super and virtual computer" is composed of a cluster of networked, loosely-coupled computers, acting in concert to perform very large tasks. For this purpose, the proponents have chosen Distributed.NET's Grid Computers.

Distributed.NET is the Internet's first general-purpose distributed computing project. Distributed.NET is responsible for cracking the DES algorithm in just less than 24 hours (Distributed Computing Technologies, 1999). The rationale for choosing Distributed.NET as the benchmark in computing the crack time of BEM was that their clients were specifically designed for exhaustive search (brute force attack) and that their project statistics are readily available on the internet. Table 5 summarizes the information about the two platforms.

Table 2 Information about the platforms that were used for benchmarking

	Desktop Computer	**Grid Computer (Distributed.NET)**
Processor	Intel Core 2 Quad (3.01 Ghz)	Multiple Processors
Cores	4	Multiple Cores
Overall Rate (Keys per Second)	22,940,000	134,636,054,338

The information shown in Table 5 was taken from the official Distributed.NET project website. The overall rate for the desktop computer was computed using Distributed.NET's calculator found on their website (Distributed Computing Technologies, 2008). The overall rate for the grid computer was taken directly from the RC5-72 project statistics of Distributed.NET (Distributed Computing Technologies, 2008).

In order to compute the total crack time for the BEM using the set-up described in Table 4 the proponents must first compute the total number of combinations that needed to be generated in order to crack the BEM. In other words, the first step is to compute for the total keyspace. The equation to solve for the total keyspace is defined by Equation 1 (Determining the size of the Keyspace).

$$K = 2^b * M; \quad \text{(Equation 1)}$$

Where:

K is the total number of keys in the keyspace
b is the block size (in bits)
M is the total number of missing or extracted blocks

By using Equation 1 and plugging in the values described in Table 5 we arrive at $(\mathbf{2^{65536} * 1})$ or 2 x $10^{19,728}$ keys. To put things in perspective, there are 2 x $10^{19,728}$ keys in the keyspace and on average; the attacker needs to brute force at least half of the keyspace until the correct key is found [3]. In the set-up described in Table 4, the attacker needs to generate at least 1 x $10^{19,728}$ keys.

As described previously, to compute for the total time needed to crack the BEM using a brute force attack, the proponents have decided to use both a desktop computer and a grid computer to translate the number of keys needed to be generated to the total time needed to crack the BEM. In order to get the total crack time, the proponents have used the simple formula shown in Equation 2.

$$T = \frac{K * 0.50}{R}; \quad \text{(Equation 2)}$$

Where:

T is the total computer time needed to crack the BEM (in seconds)
K is the total number of keys in the keyspace
R is the rate at which the computer generates a key (in seconds)

By using Equation 2 and plugging in the Keyspace (K) from Equation 1 and, the Overall Rate (R) indicated in Table 5 the proponents were able to compute that it would take 1.38 x $10^{19,713}$ years or 4.37 x $10^{19,720}$ $(\frac{\mathbf{2 \text{ x } 10^{19,728} * 0.50}}{\mathbf{22,940,000}})$ seconds to brute force the BEM using the set-up described in Table 4 using a desktop computer. On the other hand, it would take 2.36 x $10^{19,709}$ years or 7.44 x $10^{19,716}$ $(\frac{2 \text{ x } 10^{19,728} * 0.50}{134,636,054,338})$ seconds to brute force the BEM using a grid computer. Table 6 summarizes these results.

Table 6 Computer Time needed to crack the BEM

Platform	Keyspace (K)	Overall Rate (R)	Needed Computer Time (T)
Desktop Computer	2 x $10^{19,728}$	22,940,000	1.38 x $10^{19,713}$ years
Grid Computer	2 x $10^{19,728}$	134,636,054,338	2.36 x $10^{19,709}$ years

In conclusion, the BEM was secure enough in the sense that even under a worst-case scenario it would take attackers an eternity to retrieve the plaintext from the Lock File. Table 7 shows the need computer time to brute force the most prominent cryptographic algorithms using Distributed.NET's Grid Computer.

Table 7 Comparison chart of the crack time needed for different cryptographic algorithms

Cryptographic Algorithm	Key Size	Keyspace	Needed Computer Time
DES	56 bits	72,057,594,037,927,936	3.10 days
RC5	72 bits	4,722,366,482,869,645,213,696	557.61 years
Twofish	128 bits	3.40 x 10^{38}	4.00 x 10^{19} years
3DES (TDES)	168 bits	3.74 x 10^{50}	1.40 x 10^{24} years
Serpent	192 bits	6.28 x 10^{57}	7.39 x 10^{38} years
Rijndael	256 bits	1.16 x 10^{77}	1.36 x 10^{58} years
BEM	65,536 bits	2.00 x $10^{19,728}$	2.36 x $10^{19,709}$ years

Conclusion

The proponents have presented Flaxor, its design, its implementation and the result of the formal evaluation of the Block Extraction method and Flaxor Technologies.

Thru formal evaluation, the proponents have concluded that Flaxor's Block Extraction Module was able to dramatically increase (at least in theory) the amount of security of the AES cipher by protecting it against linear and differential cryptanalytic attacks, and more importantly, brute-force attacks. On the security analysis via simulating a Brute-Force attack, it was shown that the BEM is $1.73\ x\ 10^{19,653}\%$ more secure than the leading encryption algorithm, Rijndael. This has led to the conclusion that the proposed "block extraction" method, using the underlying "avalanche-effect" principle, is highly effective when used in a personal file encryption utility. On the other hand, in the

runtime analysis of the Block Extraction Module, the proponents have shown that the block extraction method used by Flaxor is at least 500% faster than the leading encryption algorithm, Rijndael, in encrypting a 6 MB plaintext. As a result, the proponents conclude that the extreme security measures implemented in the Block Extraction Module had little or no effect on the runtime performance of the proposed system.

Thru software testing, the proponents were able to conclude that the Key Generation Module using the Two-factor authentication technique was able to protect the sensitive data by forcing the user to provide the Flaxor Key in-order to access data. On the other hand, thru security testing, the proponents conclude that the only way to unlock the sensitive data without the Flaxor Key is to brute-force the content of the Key File and that the Key Generation Module is instrumental in increasing the security of the Block Extraction Module by at least $1.73\ x\ 10^{19,653}\%$.

References

[1] Stallings, W. (2006). *Cryptography and Network Security*. Upper Saddle River, NJ: Pearson/Prentice Hall

[2] Schneier B. (2006) Customer Passwords and Websites *IEEE Security and Privacy.* Retrieved January 19, 2008 from http://www.schneier.com/essay-048.pdf

[3] Schneier B. (1996) *Applied Cryptography 2nd ed*. p. 51. John Wiley and Sons

[4] Distributed Computing Technologies (2008) http://www.distributed.net/