**Introduction**

What follows is the product of my study of several encryption algorithms. I decided to write this library, out of a desire to learn more about them, and encryption in general. I have in the past adapted classes from popular libraries like Mono and Bouncy Castle, but this time I wanted to write my own implementations, ones that were optimized for the C# language, and possibly faster, and more flexible than these popular C# versions. As I was writing the base classes, I also began thinking about various attack vectors, and how they might be mitigated, and also how the existing primitives might be improved upon from a security perspective.

It is important to note, that using the base ciphers with their original key sizes, output from those classes will be exactly the same as any other valid implementation of that cipher; RDX (Rijndael) with a 256 bit key is Rijndael, as TFX (Twofish) with a standard key size is Twofish, and SPX (Serpent) is a valid Serpent implementation. This is proven. The Tests section contains the most complete and authoritative test suites available for each of these ciphers. So if you choose to stick with standard key lengths, you can use configurations that have been thoroughly cryptanalyzed.

One has to consider that these ciphers were designed more than 17 years ago; at the time, Windows 95 was the predominant operating system, and computer hardware was quite primitive by today's standards. So, concessions had to be made in cipher design in regards to speed and memory footprint. We are not so constrained with the hardware of today, so adding rounds to a cipher, or using a larger key size is less a consideration now, and will have even less impact in the future.

Speed remains an important design criterion with this project. The CTR mode and the decryption function of the CBC mode have been parallelized. If a block size of 1024 bytes is passed to the mode, and the hardware utilizes multiple processor cores, the processing is automatically parallelized. On my middle tier quad-core Acer, I have reached speeds of over 6 gigabytes per minute with this library using Rijndael, making this by far the fastest implementation of these ciphers in the C# language that I have found.

I definitely have some strong reservations about publishing this code, not the least of which is that it is likely to spawn a number of so called 'AES 512' copies by people who may not understand enough about the algorithms to evaluate, produce, or maintain a secure encryption software. It's kind of a quandary though, if I leave it on Github, no one will ever see it, if I publish it, chances are it could be used irresponsibly.. I would urge anyone considering using one of the extended algorithms to study the work, thoroughly evaluate the implementations, and make an informed choice.

As for my part, I wrote these implementations based on well-known versions, and made as few changes to the ciphers as possible to extend the key size. I have confidence in the library itself, because I took care to test it every step of the way, and feel I am developing a good understanding of the cryptographic primitives used in its construction. This should however, be considered as what it was intended to be; experiments..

I hope to expand this library in the future, as I continue my exploration of encryption technologies, and I welcome input from cryptographers and programmers. If you have a comment or concern, I'd be glad to hear from you. My goals include moving what I feel are the best and strongest implementations to a Java library.

Documentation has been added to the library as a website included with the project distribution; located in the Help folder of the test project, and accessible directly, or though the sample forms Help menu.

Before downloading the source files, it is your responsibility to check if these extended key lengths (512 bit and higher) are legal in your country. If you use this code, please do so responsibly and in accordance to law in your region.

For a full featured implementation of these algorithms, including key management, authentication controls, anti-tampering measures, encrypted assets, a very cool interface, and many more features; check out the version on my website: [CEX on vtdev.com](http://www.vtdev.com/cex.html).

**Library Components**

The library contains the following components, as it evolves, some will be added, some removed, and when possible, changes will be made to improve upon performance, security and documentation.

#### Encryption Engines

#### ****Base Algorithms****

The three base ciphers; **Rijndael**, **Serpent**, and **Twofish** have all underwent thorough testing to ensure that they align with valid implementations that use a smaller maximum key size. The same algorithms are used to transform data at any key size; only the key schedule itself has been extended, (a key schedule takes a small user key, and expands it into a larger working array, used in the rounds function to create a unique output). These changes to the key schedule, and a flexible rounds assignment, increase the potential security of the cipher, make it more difficult to cryptanalyze, and more resistant to brute force attacks.

* **RDX:** (Rijndael) This is an implementation of the Rijndael algorithm used in AES, extended to a 512 bit key.
* **SPX:** (Serpent) An implementation of the Serpent encryption algorithm, extended to accept a 512 bit key.
* **TFX:** (Twofish) An implementation of the Twofish encryption algorithm, extended to accept a 512 bit key.

#### ****HX Ciphers: Hash based Key Schedules****

The HX Series Ciphers use the identical encryption and decryption algorithms (transforms), of the standard ciphers, the difference is that the key schedule has been replaced by a Hash based Key Derivation Function (**HKDF**). The HKDF is powered by an **SHA512 HMAC**, and is one of the most cryptographically strong methods available to generate pseudo-random output. The HKDF based key schedule takes a minimum 192 bytes (1536 bits) of input as a user key, and uses that keying material to generate a cryptographically strong working key array.

There are several advantages to using a hash based KDF; the stronger working keys are less susceptible to attack vectors that leverage weak or related keys. The larger user key size, also makes brute force attacks practically impossible; 2(256-1) compared to a minimum of 2(1536-1) iterations. Another advantage of the HX ciphers is that the number of diffusion rounds, (transformation cycles within the rounds function), are configurable independent of the initial key size.

* **RHX:** Minimum 1536 bit key, and up to 38 rounds of diffusion.
* **SHX:** Minimum 1536 bit key, and up to 128 rounds of diffusion.
* **THX:** Minimum 1536 bit key, and up to 32 rounds of diffusion.

#### ****Super Ciphers****

These are merged ciphers; where two ciphers are combined during the rounds processing stage. There are a number of existing implementations that use this combined cipher technique, they either encrypt a file twice, each time using a different key and cipher, or use the combined output from counter encryptions to create a pseudo random key stream which is combined with the input to create the cipher-text. Both of these methods share the same weakness, called a ‘meet in the middle’ attack. This is where it may be theoretically possible to unwind both ciphers with little more than the computational effort required to break one alone. I got around this by ‘merging’ the ciphers within the transform function. For example, RSM combines the Rijndael and Serpent ciphers. In the rounds function, the input data undergoes a round of Rijndael, the product of that round is processed as a round of Serpent, then another round of Rijndael etc. So if set to 18 rounds, it processes the state with 18 rounds of Rijndael and 16 rounds of Serpent. This should effectively mitigate the meet in the middle attack, and make most forms of differential or linear analysis far more difficult.

* **RSM:** Rijndael and Serpent merged. HKDF key scheduler and up to 42 rounds of diffusion.
* **TSM:** Twofish and Serpent merged. HKDF key scheduler and up to 32 rounds of diffusion.
* **Fusion:** Rijndael and Twofish merged. HKDF key scheduler and up to 32 rounds of diffusion.
* **DCS:** Two separate Rijndael cipher instances combined and used as a random key stream.

#### Cipher Modes

The project focuses on two modes; CTR, a segmented integer counter mode, and CBC, cipher block chaining. These are considered two of the most secure cipher modes. The CTR mode is automatically parallelized, as is the decryption function of the CBC mode. The project also includes ECB and CFB modes that are not currently implemented.

#### Padding

Some modes, like CBC, require block aligned input lengths. If at the end of an array of input data, the last block is less than the cipher block size, padding is added to complete the block. The project currently implements X9.23 and PKCS7 padding modes.

#### Hash Algorithms

* **SHA256:** An implementation of SHA-2 with a 256 bit hash output.
* **SHA512:** An implementation of SHA-2 with a 512 bit hash output.
* **SHA-3:** An implementation of the Keccak based SHA-3, with variable output sizes.

#### MAC

* **HMAC:** Wrapper for Hash based Message Authentication Code, works with all 3 hash algorithms.
* **SHA256HMAC:** HMAC and SHA256 combined in a class.
* **SHA512HMAC:** HMAC and SHA512 combined in a class.

#### Queuing

* **WaitQueue** class, demonstrates using a queue to create a constant time implementation.

#### Security

* **SecureDelete:** a 6 stage secure file deletion class.

#### Utilities

* **Compression:** a fully implemented compression and folder archiving class.
* **FileUtilities:** a variety of file, folder, and drive functions.

**Overview**

Before we start looking at some of the ciphers and getting into implementation details, I think it helps to 'break it down' a bit, give you a general idea of what has been done, and clarify some of the concepts and terminology used in the article.

First off, the ***key schedule***: A [key schedule](http://en.wikipedia.org/wiki/Key_schedule) is a function that takes a small amount of user supplied data (the cipher key), and expands it, usually into a larger integer array. For example; Rijndael takes a 32 byte key (256 bit), and expands that into 60 integers, or 240 bytes worth of keying material. That array of integers is sometimes called an array of 'rounded' keys, 'subkeys' or 'working' keys, I'll use the term working key, because it makes it clear that it is a derived key. Some key schedules have a simple algebraic expression; Rijndael for example, derives most of the working keys with a simple exclusive OR of two previous keys. Serpent uses a much more elaborate key schedule, one that was designed to resist some forms of cryptanalysis. These working keys, created by the key schedule are used to create a unique cipher text, and a good cipher design is one in which a change of just a single bit in the cipher key, results in a completely different output, this is known as the 'avalanche' property. The working keys are usually added to the state (input data at some stage of transformation), with a simple addition or XOR.

Larger keys play an important role in a ciphers security. Many of the techniques used to 'break' a cipher involve the reduction in the number of times a unique key is tested against the ciphers decryption output, in other words, they reduce the number of brute force attempts required to decrypt the output. When thinking of key sizes in this context, it helps to understand some binary math.

Keys are measured in bits for a reason, because the sum of the integer the key represents is the 2 square sum of its bits. Think of it like the penny a day riddle; I loan you a dollar, you agree to pay me back a penny on the first day, and then double that each day for the rest of the month. By the last day of the month, you'd owe more than 10 million dollars! That's binary math, each time you add a bit, you double the size of the previous sum. So, a 256 bit key represents an integer with a maximum size of 1.1579208923731619542357098500869e+77, or rounded as 1.15 times 10 to the power of 77. A simply monstrous number.. and one might think that computers will never be fast enough to run a decryption cycle that many times, and at this time that is almost certainly true, but some cryptanalytic attacks aim to reduce that number, sometimes quite substantially. A larger key puts this further out of reach; given that the cipher key and the working key produced is done in a cryptographically strong way, much larger keys are feasible, and by using those longer keys, data can be kept beyond the capabilities of technology for a longer period of time.

There is also evidence that the key schedule plays a part in providing strength against linear and differential cryptanalysis, and there have been some serious attacks on Rijndael that leverage the weak key schedule. So it follows that a cryptographically strong key schedule, can help create a stronger cipher.

In this article, a ***transform*** is a function that performs the actual encryption of data, just as the inverse transform performs the decryption in a reversible iterative block cipher. In a rounds based cipher, (sometimes called a [product cipher](http://en.wikipedia.org/wiki/Product_cipher)), a ***round*** can be thought of as one complete sequence of transformation, whereas the transform function, (or rounds function), may loop through a number of rounds and use [whitening](http://en.wikipedia.org/wiki/Key_whitening) stages.

In the three block ciphers presented here (Rijndael, Serpent, and Twofish), The input bytes are first copied into four integers. These integers are (in the case of Rijndael and Twofish), XORd with members of the working key, (key whitening). These state integers are then processed in a series of rounds, which change the state via a series of substitutions, permutations, and modular arithmetic, (with the key added in stages), finally the processed state is whitened with the remaining key members and copied into the output byte array.

Let's look at a round of Serpent:

R0 ^= \_exKey[keyCtr++];

R1 ^= \_exKey[keyCtr++];

R2 ^= \_exKey[keyCtr++];

R3 ^= \_exKey[keyCtr++];

Sb0(ref R0, ref R1, ref R2, ref R3);

LinearTransform(ref R0, ref R1, ref R2, ref R3);

R0 through R3 are state integers. Before each round the state is XORd with a member of the working key. The state is then processed through one of eight bit slicing S-Boxes before undergoing a linear transform. This clearly illustrates the role of the working key during a round cycle; the working keys are used to mix with the state in a way that will produce an output that is unique to that key, this is their purpose, and in these ciphers, they do not interact with the transformation functions in any other way.

One often hears of the term rounds in the context of the number of rounds that can be broken using an attack on the cipher; Rijndael has been shown to be vulnerable to a known-key [distinguishing attack](http://en.wikipedia.org/wiki/Distinguishing_attack) against a reduced 8-round version of AES-128. These attacks are often aimed at reduced versions, where a smaller number of rounds can be broken, as both a means of providing proof with limited computing power, and positing the method by which a full transformation might be reversed. This is because with most ciphers, adding rounds increases the security of the cipher by making differential or linear cryptanalysis more difficult. There have been a number of noted cryptographers who have stated that the number of rounds used in Rijndael should be increased, that its simple algebraic description makes it vulnerable with the current round counts, (10, 12, and 14), and it should be increased to 18 or more rounds to ensure its continued integrity..

#### The Base Ciphers

**RDX** (RijnDael eXtended) is a Rijndael implementation that can process up to a 512 bit key. Up to a 256 bit key, it will produce the exact same output as any other valid implementation of Rijndael. This is proven. In the tests section of the project, the AesAvsVector class tests the complete set of [AESAVS](http://csrc.nist.gov/groups/STM/cavp/documents/aes/AESAVS.pdf) (Advanced Encryption Standard Algorithm Validation Suite) known answer vector and Monte Carlo tests. These are the same tests used to get an AES implementation certified by NIST. Further tests from the AES submission package and KATs (Known Answer Tests) testing a 32 byte block size are also included.

**SPX** (SerPent eXtended) is a Serpent implementation. It can also use up to a 512 bit key. The number of rounds in SPX is also configurable; from the default 32 rounds, to a full 64 transformation rounds. Just like with Rijndael, I used the most complete and authoritative test suite I could find; the complete Nessie [Serpent](http://www.cs.technion.ac.il/~biham/Reports/Serpent/) test suite. The tests include 100 thousand rounds of Monte Carlo tests, and is the recommended test suite for the Serpent cipher. This means that up to a 256 bit key, the output from SPX is identical to any other valid version of Serpent.

**TFX** (TwoFish eXtended) is a Twofish implementation. Just as with RDX and SPX it can process the larger 512 bit key size. The number of rounds is also configurable; from the default 16 rounds to a maximum of 32 rounds of transformation. Just as with the other two, the most complete tests available are run on the standard key lengths, in this case the official [Twofish KATs](https://www.schneier.com/twofish.html).

With all three of these ciphers, I first analyzed the existing patterns within the key schedules, then sought to extend them using as few changes as possible to the original algorithms. In the case of Rijndael and Serpent, the changes were almost trivial, Twofish, because of the keyed S-Box, required a more thorough examination. In all cases, I did my best to understand the nature of the function, both programmatically and as mathematical expression, implementing the extensions in the way I thought was closest to the original, but also best leveraged the additional cipher key entropy provided by a larger initial key size.

#### The HX Ciphers

One of the central goals of this project has been to create the strongest ciphers possible, using existing and proven cryptographic primitives. Another important goal was to try to better understand various attack vectors, and create something that was more resistant to these attacks.

The three base ciphers all have something in common; they all use the working key in a similar way; to change or 'whiten' the state values to create a unique output, other than that, they do not interact with the actual computational processes used to transform the state. What that means is that how that cipher key is expanded, (so long as it is done in a secure way), does not directly impact the data transformation. Creating that expanded key using a more secure means, like a hash function, can increase the overall security of the cipher itself.

The HX ciphers; **RHX**, **SHX**, and **THX** all use **HKDF**, that's a Hash based Key Derivation Function, a kind of pseudo-random generator. HKDF is powered by an **SHA-2 512 HMAC**, a keyed hash function. This is one of the most cryptographically strong methods of creating a pseudo-random output; even a strong key schedule like the one used in Serpent, is not as secure as using this method to generate the working keys. Aside from the increased security, there are two additional advantages to replacing the key schedule with a hash based KDF; it is more resistant to weak key and sliding attacks, and the longer cipher key size, makes brute force attacks impossible.

**Timing attacks** use discrete differences in the length of time it takes to perform a task with a given set of parameters. In the case of an attack on a key schedule, it measures the distance in timing of things like branching and table lookups to make predictions about the key; like the slight difference between looking up the first or last value in a table of integers, or the computational time averaged to compute an output given the value of a specific table member. SHA-2 is less vulnerable to timing attacks, because the amount of time required to run is typically more constant than say.. the Rijndael key schedule.

The other advantage is the **key size**; the minimum key size for an HX cipher is the block size of the hash function (SHA512 = 128 bytes) + the IKM, or the HMAC key material (64 bytes). So the key for these ciphers is a minimum of 192 bytes (1536 bits), but expandable up to any size in multiples of the hash functions block size. This might seem like a very large key, but consider; my 256GB thumb drive could easily store over a billion keys.. the benefit is obvious; even when quantum computers are made that can break a 256 bit key, it will still be many years (decades) from that point before they could brute force a 1536 bit key.

**Super Ciphers**

Super Ciphers have been around for a while, and there are a number of different implementations of software that double, or even triple encrypt an input using different ciphers and keys. There are also implementations that use multiple instances of the same cipher, (think Triple-DES). This is done to extend key size and make the output more resistant to some forms of [Linear](http://en.wikipedia.org/wiki/Linear_cryptanalysis) and [Differential](http://en.wikipedia.org/wiki/Differential-linear_attack) cryptanalysis. The problem with this approach is that it is subject to a '[meet in the middle attack'](http://en.wikipedia.org/wiki/Meet-in-the-middle_attack). Some theoretical models project that decryption could be performed with little more computational energy than brute forcing only one of the cipher instances. One way to mitigate this attack, is instead of encrypting successively with different cryptographic instances, it makes more sense to combine the primitives at the rounds processing level.

Let's look at two rounds of RSM:

// serpent sbox and transform

Sb0(ref R0, ref R1, ref R2, ref R3);

LinearTransform(ref R0, ref R1, ref R2, ref R3);

// rijndael round

R0 = T0[C0 >> 24] ^ T1[(byte)(C1 >> 16)] ^ T2[(byte)(C2 >> 8)] ^ T3[(byte)C3] ^ \_exKey[keyCtr++];

R1 = T0[C1 >> 24] ^ T1[(byte)(C2 >> 16)] ^ T2[(byte)(C3 >> 8)] ^ T3[(byte)C0] ^ \_exKey[keyCtr++];

R2 = T0[C2 >> 24] ^ T1[(byte)(C3 >> 16)] ^ T2[(byte)(C0 >> 8)] ^ T3[(byte)C1] ^ \_exKey[keyCtr++];

R3 = T0[C3 >> 24] ^ T1[(byte)(C0 >> 16)] ^ T2[(byte)(C1 >> 8)] ^ T3[(byte)C2] ^ \_exKey[keyCtr++];

Sb1(ref C0, ref C1, ref C2, ref C3);

LinearTransform(ref C0, ref C1, ref C2, ref C3);

C0 = T0[R0 >> 24] ^ T1[(byte)(R1 >> 16)] ^ T2[(byte)(R2 >> 8)] ^ T3[(byte)R3] ^ \_exKey[keyCtr++];

C1 = T0[R1 >> 24] ^ T1[(byte)(R2 >> 16)] ^ T2[(byte)(R3 >> 8)] ^ T3[(byte)R0] ^ \_exKey[keyCtr++];

C2 = T0[R2 >> 24] ^ T1[(byte)(R3 >> 16)] ^ T2[(byte)(R0 >> 8)] ^ T3[(byte)R1] ^ \_exKey[keyCtr++];

C3 = T0[R3 >> 24] ^ T1[(byte)(R0 >> 16)] ^ T2[(byte)(R1 >> 8)] ^ T3[(byte)R2] ^ \_exKey[keyCtr++];

The rounds in RSM run in a loop; all eight of the Serpent S-Boxes and linear transforms are processed in one complete loop cycle, along with eight rounds of Rijndael. Looking at the code above, you can see how the state (Rn and Cn), first passes through a full round of Serpent, then the product of that transformation undergoes a round of Rijndael. Manipulating the state directly, and this combining of ciphers with very different algebraic compositions, should make the output more resistant to meet in the middle attacks, as well as making other forms of cryptanalysis far more difficult.

It was even possible to create an invertible cipher in the case of RSM (Rijndael/Serpent Merge), and TSM (Twofish/Serpent Merge). The third 'super cipher' is a stream cipher named Fusion. Fusion combines full rounds of Twofish and Rijndael, including the working key processing for each. It uses a random 128 bit integer counter to create a pseudo-random key stream, in parallel, and XORd with the input to create cipher text.

**Benchmarks**

Speed tests were performed on an AMD ASD-3600 Quad-Core, 4GB RAM (3.47 GB usable), compiled Release/Any CPU on Windows 7.  
Test is a transform of a byte array in a Monte Carlo method.  
Sizes are in MBs (1000000 bytes). Time format sec.ms, key sizes in bits. Rate is MB per minute.  
HX series will have similar times, as they use the same transformation engines.  
CTR mode and CBC decrypt are run in parallel mode. CBC encrypt is in linear (single processor) mode.  
Highest rate was RDX with a 128 bit key: 6.052 GB per minute!

**Block Ciphers**

**RDX** (Rijndael): 256 Key, 14 Rounds  
Mode  State   Size    Time    Rate  
----     -----     ----     ----     ----  
CTR     ENC     100     1.18    5084  
CTR     DEC     100     1.18    5043  
CBC     ENC     100     3.03    1980  
CBC     DEC     100     1.59    3773

**RSM** (Rijndael/Serpent Merged): 1536 Key, 18 Rounds  
Mode  State   Size    Time    Rate  
----     -----     ----     ----     ----  
CTR     ENC     100     2.69    2230   
CTR     DEC     100     2.68    2238  
CBC     ENC     100     8.15    736  
CBC     DEC     100     3.17    1892

**SPX** (Serpent): 256 Key, 32 Rounds  
Mode  State   Size    Time    Rate  
----     -----     ----     ----     ----  
CTR     ENC     100     2.22    2702  
CTR     DEC     100     2.21    2714  
CBC     ENC     100     6.45    930  
CBC     DEC     100     2.39    2510

**TFX** (Twofish): 256 Key, 16 Rounds  
Mode  State   Size    Time    Rate  
----     -----     ----     ----     ----  
CTR     ENC     100     1.33    4511  
CTR     DEC     100     1.35    4444  
CBC     ENC     100     3.55    1690  
CBC     DEC     100     1.74    3448

**Stream Ciphers**

**ChaCha** 256 Key, 20 Rounds  
Size    Time    Rate  
----     ----      ----  
100     2.26    2027

**DCS** 768 Key  
Size    Time    Rate  
----     ----      ----  
100     2.12    2830

**Fusion** 2560 Key  
Size    Time    Rate  
----     ----      ----  
100     2.16    2777

**Salsa20** 256 Key, 20 Rounds  
Size    Time    Rate  
----     ----      ----  
100     2.68    2238