**Introduction**

There are three algorithms included in the project, all based to some degree on the Rijndael encryption algorithm. The first algorithm is named **RDX**; that stands for Rijndael Extended. It is the Rijndael algorithm extended to accept the longer key length of 512 bits (64 bytes). Aside from this change in available key sizes, it is the exact same algorithm used in Rijndael and AES implementations.

The second algorithm, **RSX** (Rijndael/Serpent Extended), is based on RDX, but uses an adapted key scheduler from the Serpent encryption algorithm.

The third is **DCS** (Dual CTR Stream), which is two independent AES CTR implementations that create a pseudo-random stream by XORing the two CTRs, and encrypts by XORing that stream with plaintext, all implemented using processer parallelization.

I wrote RDX with a specific set of design criteria; I wanted something that was faster and more flexible than the most common C# implementations of Rijndael. Another goal was to add a 512 bit key, without altering the cryptographic primitives in any way. One of the first tests I performed was to create random key, iv, and 128 bytes of plaintext within a loop, and then perform an equivalency test against the output of RijndaelManaged on encrypt cycles, and compare the decryption output to the plaintext. I ran this test a million times on each key size without failure. Now obviously this took hours to run, so it is not included in the project, but would be fairly straightforward to reproduce. The next step was to apply standard tests. The NIST AESAVS known answer tests are used to certify an AES implementation, all of the plaintext and key tests, 960 in total, are performed in the sample application, as well as the FIPS 197, and some of the Nessie unverified vector tests. So, up to a 256 bit key, RDX will produce the same output as any other valid implementation of Rijndael. Now of course, there are no tests for a 512 bit key, as this has not been adopted, and as far as I know, this is the only open source implementation that can use a 512 bit key. So is a 512 bit key safe? Well, there are no certainties in encryption, and some clever fellow is sure to create an attack that targets the longer key length with some success. But the real question is, how much could the security be reduced? Remember that a single bit doubles the number of calculations required to brute force a key, so as long as it could not be reduced by the full 256 bits, plus whatever amount that attack could reduce a 256 bit key, it is still stronger than the 256 bit key. A 512 bit key also adds 8 more rounds to the diffusion algorithm, something that has been suggested before.

RSX is simply a response to Rijndaels weak key scheduler. The diffusion algorithm in Rijndael has an AddRoundKey step, which is just a simple Xor of the round key and the result of numerous table lookups. The forward cipher, (encryption), has no direct bond to how those round keys are generated, that is, so long as the round keys are of the correct number, and sufficiently random and distant from each other, it does not matter how they are generated. Most of the keys generated by Rijndaels key scheduler use a very weak formula, it is simply the Xor of the previous key, and a key Nk positions previous. This isn't even a weak PRF, and a number of serious attacks have been created that exploit the weak key scheduler. Now because there is no direct correlation between how the diffusion algorithm processes the key, and how the expanded keys are generated, it is possible to replace the key scheduler. In fact, you could use a secure PRNG to generate the entire 60 or 92 member integer array and apply that as a key directly, (it's a sledgehammer approach, granted, but not so impractical given current wire speeds and media sizes). But I didn't want to go too far, but rather use a well understood key scheduler, adapt it for Rijndael, and mitigate some weak key based attacks. I looked at each of the key schedulers among the AES finalists, and finally decided on Serpent's, a very strong scheduler, that does a much more thorough job of dispersing the initial entropy. If I had to choose between RDX and RSX, given any key size, it would be RSX. If I had to choose between all three of these algorithms, it would be DCS..

The idea of combining two encryption algorithms, or even two unique instances of the same algorithm, has been around for some time now, it has been studied, there are accepted security proofs, and if implemented properly, it is generally regarded as a safe way to build a strong encryptor. DCS uses two Rijndael transforms, each with unique counters and keys to generate a pseudo random stream. It is fully parallelized, written to take advantage of any number of processor cores. Because it is written using an AES configuration, it could be ported to C or C++ and adapted to use the AES-NI api, the processor level instruction set for AES.

**RDX**

**Overview**

**RDX** uses a variation of the Rijndael encryption algorithm, the same one used in the AES standard. What I have done is to extend Rijndael so that it now accepts the longer key length (512 bits). The extended key length provides more security against attacks that attempt to brute force the key, and also adds eight more rounds of diffusion.

The increased number of rounds brings the total from 14 rounds with a 256 bit key, to 22 rounds with the 512 bit key size. These added passes through the diffusion algorithm further disperse the input through row and column transpositions, and XOR’s with a longer rounded key array.

**Comparison to the Rijndael**

**Key Scheduler:**

The key scheduler is where this version departs from the standard. The maximum key length has been extended to allow for a 512 bit (64 byte) key size. Creating this extension requires two conditions; the diffusion rounds should be written as a loop, (to accommodate the additional rounds of mixing), and the key scheduler code must be written in a way that it can process the ***user key*** into a larger ***working key***, (the integer array that the diffusion algorithm uses as part of the input transformation process). The key scheduler I have used here is based on the [Mono library](http://www.mono-project.com/) implementation used in the class [RijndaelManagedTransform.cs](http://github.com/mono/mono/blob/effa4c07ba850bedbe1ff54b2a5df281c058ebcb/mcs/class/corlib/System.Security.Cryptography/RijndaelManagedTransform.cs).

private void ExpandKey(byte[] Key, bool Encryption)

{

int pos = 0;

// block and key in 32 bit words

Nb = this.BlockSize / 4;

Nk = Key.Length / 4;

// rounds calculation

if (Nk == 16)

Nr = 22;

else if (Nb == 8 || Nk == 8)

Nr = 14;

else if (Nk == 6)

Nr = 12;

else

Nr = 10;

// setup expanded key

int keySize = Nb \* (Nr + 1);

\_exKey = new UInt32[keySize];

// add bytes to beginning of working key array

for (int i = 0; i < Nk; i++)

{

UInt32 value = ((UInt32)Key[pos++] << 24);

value |= ((UInt32)Key[pos++] << 16);

value |= ((UInt32)Key[pos++] << 8);

value |= ((UInt32)Key[pos++]);

\_exKey[i] = value;

}

// build the remaining round keys

for (int i = Nk; i < keySize; i++)

{

UInt32 temp = \_exKey[i - 1];

// if it is a 512 bit key, maintain step 8 interval for

// additional processing steps, equal to a 256 key distribution

if (Nk > 8)

{

if (i % Nk == 0 || i % Nk == 8)

{

// round the key

UInt32 rot = (UInt32)((temp << 8) | ((temp >> 24) & 0xff));

// subbyte step

temp = SubByte(rot) ^ Rcon[i / Nk];

}

// step ik + 4

else if ((i % Nk) == 4 || (i % Nk) == 12)

{

temp = SubByte(temp);

}

}

else

{

if (i % Nk == 0)

{

// round the key

UInt32 rot = (UInt32)((temp << 8) | ((temp >> 24) & 0xff));

// subbyte step

temp = SubByte(rot) ^ Rcon[i / Nk];

}

// step ik + 4

else if (Nk > 6 && (i % Nk) == 4)

{

temp = SubByte(temp);

}

}

// w[i-Nk] ^ w[i]

\_exKey[i] = (UInt32)\_exKey[i - Nk] ^ temp;

}

// inverse cipher

if (!Encryption)

{

// reverse key

for (int i = 0, k = keySize - Nb; i < k; i += Nb, k -= Nb)

{

for (int j = 0; j < Nb; j++)

{

UInt32 temp = \_exKey[i + j];

\_exKey[i + j] = \_exKey[k + j];

\_exKey[k + j] = temp;

}

}

// sbox inversion

for (int i = Nb; i < keySize - Nb; i++)

{

\_exKey[i] = IT0[SBox[(\_exKey[i] >> 24)]] ^

IT1[SBox[(byte)(\_exKey[i] >> 16)]] ^

IT2[SBox[(byte)(\_exKey[i] >> 8)]] ^

IT3[SBox[(byte)\_exKey[i]]];

}

}

this.IsInitialized = true;

}

The first thing to note is the rounds calculation; I have added an additional clause of

***if (Nk == 16) Nr = 22***

**Nk** is equal to the number of 32 bit words in the user supplied key

**Nr** is the number of rounds as a function of Nk and Nb

**Nb** is the size of the state in 32 bit words, in this implementation Nb is either 4 (16 byte block), or 8 (32 byte block).

The number of round keys created is a function of ***Nb (Nr + 1)***, which is the number of rounds plus one, multiplied by the number of 32 bit words in the block size. A 512 bit user key with a 16 byte block size will generate 92 working keys, or 184 working keys with a 32 byte block. A 256 bit key generates 60 or 120 working keys. A better dispersion ratio of user key to expanded key size is achieved with the larger 512 bit key; bytes(32 byte key: 32/240 and 32/480, 64 byte key: 64/368 and 64/736).

The next step is adding the user key to the beginning of the working key, right shifting the user key bytes into the working key, which adds the first eight integers with a 256 bit key, or 16 integers with a 512 bit key.

We then create the additional working keys. The working key, exKey[i], is equal to the XOR of the previous word, exKey[i-1], and the word Nk positions earlier, exKey[i-Nk]. E

With a 256 bit key; every Nk interval, an additional step is added that first rounds the key, then processes it with SubByte(), (SBox applied to each byte), then Xors this with a round constant from the Rcon table. This extra step happens on a modulus of i % Nk, with a 256 bit key that is step 8, or every 8 passes through the loop.

The expansion routine for a 256 bit key uses some additional processing; if *i % Nk* yields a remainder of 4, then SubByte() is applied to exKey[i-1] prior to the XOR with exKey[i-Nk]. The designers implemented this to further disperse the larger 256 bit key.

With a 512 bit key; Nk = 16, which would double the interval between these additional dispersal steps, and create a weaker expanded key. I have compensated for this by maintaining the same intervals in the dispersion pattern; just as with a 256 bit key every 8 keys, the Rcon ^ SubByte step executes, with the SubByte step at the same alternating offset 4 interval.

If the transform is for decryption then a routine performs the additional key reversal and SBox inversion, required by the inverse cipher.

As I mentioned this implementation of the key expansion routine is based on the C# Mono version, the only real change I have made was to the rounds calculation, adding the additional 8 rounds of diffusion when a 512 bit key is used, other than that the key expansion routine is exactly the same.

**Diffusion Algorithm:**

When writing this method I first took a look at a number of implementations in various languages, to try and get a better idea of the different ways in which the diffusion algorithm could be expressed programmatically; and how that related to the strengths and weaknesses of the C# language, (though these classes could very easily be ported to Java or C).

The Bouncy Castle [version](https://github.com/bcgit/bc-csharp/blob/master/crypto/src/crypto/engines/AesFastEngine.cs) processes a multi-dimensional array of unsigned integers rather than bytes, (which is more in keeping with the specification outline). It converts the bytes to integers and back again after the transformation.

The Mono version processes bytes, using method level integers for row and column transpositions. My first thought was to avoid a multi-dimensional array, as these are slow in almost every language. I did some speed comparisons, and the two versions were close to equal, but there were some things that could be done to speed up processing in the Mono version, and because we are using a variable key size, we need the diffusion rounds to run in a loop.

To get some idea of how different the methods are, look at the Tests\CompareEngines.cs speed test, the diffusion algorithm from Bouncy Castle, Mono, and RDX are all there, and they are all quite different, (and RDX is fastest). This is what the 16 byte block version of the encryption algorithm looks like in RDX:

private void Encrypt16(byte[] Input, byte[] Output)

{

int ct = 0;

UInt32 R0, R1, R2, R3, C0, C1, C2, C3;

// Round 0

R0 = (((UInt32)Input[0] << 24) | ((UInt32)Input[1] << 16) | ((UInt32)Input[2] << 8) | (UInt32)Input[3]) ^ \_exKey[ct++];

R1 = (((UInt32)Input[4] << 24) | ((UInt32)Input[5] << 16) | ((UInt32)Input[6] << 8) | (UInt32)Input[7]) ^ \_exKey[ct++];

R2 = (((UInt32)Input[8] << 24) | ((UInt32)Input[9] << 16) | ((UInt32)Input[10] << 8) | (UInt32)Input[11]) ^ \_exKey[ct++];

R3 = (((UInt32)Input[12] << 24) | ((UInt32)Input[13] << 16) | ((UInt32)Input[14] << 8) | (UInt32)Input[15]) ^ \_exKey[ct++];

// Round 1

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

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

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

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

while (ct < Nr \* Nb)

{

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

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

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

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

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

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

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

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

}

// Final Round

Output[0] = (byte)(SBox[C0 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[1] = (byte)(SBox[(byte)(C1 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[2] = (byte)(SBox[(byte)(C2 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[3] = (byte)(SBox[(byte)C3] ^ (byte)\_exKey[ct++]);

Output[4] = (byte)(SBox[C1 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[5] = (byte)(SBox[(byte)(C2 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[6] = (byte)(SBox[(byte)(C3 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[7] = (byte)(SBox[(byte)C0] ^ (byte)\_exKey[ct++]);

Output[8] = (byte)(SBox[C2 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[9] = (byte)(SBox[(byte)(C3 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[10] = (byte)(SBox[(byte)(C0 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[11] = (byte)(SBox[(byte)C1] ^ (byte)\_exKey[ct++]);

Output[12] = (byte)(SBox[C3 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[13] = (byte)(SBox[(byte)(C0 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[14] = (byte)(SBox[(byte)(C1 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[15] = (byte)(SBox[(byte)C2] ^ (byte)\_exKey[ct]);

}

This is based on the Mono version with some important differences; in the Mono version, the rounds transpositions are laid out sequentially with the key index as a fixed integer, and *if ei >* clauses that control the number of rounds processed based on the extended key size. In this version, a single incrementing integer *ct* is used as the key index, and the rounds are processed in a while loop based on the formula *ct < Nr \* Nb*. Using a single counter, and eliminating the rounds count clauses increases the speed significantly.

As you can see this version uses the byte oriented approach and is optimized by combining the SubBytes and ShiftRows steps with the MixColumns step by transforming them into a sequence of table lookups using the byte values as table indices. This requires four 256 member pre-calculated lookup tables to perform the byte multiplication. The AddRoundKey step is then performed with an additional Xor, and a round is completed with just with 16 table lookups, and 16 Xor operations.

The 32 byte block size also conforms to a standard Rijndael configuration, by shifting the lookup value on the second, third and fourth row by 1 byte, 3 bytes and 4 bytes respectively. This is done to keep columns linearly independent.

private void Encrypt32(byte[] Input, byte[] Output)

{

int ct = 0;

UInt32 R0, R1, R2, R3, R4, R5, R6, R7, C0, C1, C2, C3, C4, C5, C6, C7;

// Round 0

R0 = (((UInt32)Input[0] << 24) | ((UInt32)Input[1] << 16) | ((UInt32)Input[2] << 8) | (UInt32)Input[3]) ^ \_exKey[ct++];

R1 = (((UInt32)Input[4] << 24) | ((UInt32)Input[5] << 16) | ((UInt32)Input[6] << 8) | (UInt32)Input[7]) ^ \_exKey[ct++];

R2 = (((UInt32)Input[8] << 24) | ((UInt32)Input[9] << 16) | ((UInt32)Input[10] << 8) | (UInt32)Input[11]) ^ \_exKey[ct++];

R3 = (((UInt32)Input[12] << 24) | ((UInt32)Input[13] << 16) | ((UInt32)Input[14] << 8) | (UInt32)Input[15]) ^ \_exKey[ct++];

R4 = (((UInt32)Input[16] << 24) | ((UInt32)Input[17] << 16) | ((UInt32)Input[18] << 8) | (UInt32)Input[19]) ^ \_exKey[ct++];

R5 = (((UInt32)Input[20] << 24) | ((UInt32)Input[21] << 16) | ((UInt32)Input[22] << 8) | (UInt32)Input[23]) ^ \_exKey[ct++];

R6 = (((UInt32)Input[24] << 24) | ((UInt32)Input[25] << 16) | ((UInt32)Input[26] << 8) | (UInt32)Input[27]) ^ \_exKey[ct++];

R7 = (((UInt32)Input[28] << 24) | ((UInt32)Input[29] << 16) | ((UInt32)Input[30] << 8) | (UInt32)Input[31]) ^ \_exKey[ct++];

// Round 1

C0 = T0[R0 >> 24] ^ T1[(byte)(R1 >> 16)] ^ T2[(byte)(R3 >> 8)] ^ T3[(byte)R4] ^ \_exKey[ct++];

C1 = T0[R1 >> 24] ^ T1[(byte)(R2 >> 16)] ^ T2[(byte)(R4 >> 8)] ^ T3[(byte)R5] ^ \_exKey[ct++];

C2 = T0[R2 >> 24] ^ T1[(byte)(R3 >> 16)] ^ T2[(byte)(R5 >> 8)] ^ T3[(byte)R6] ^ \_exKey[ct++];

C3 = T0[R3 >> 24] ^ T1[(byte)(R4 >> 16)] ^ T2[(byte)(R6 >> 8)] ^ T3[(byte)R7] ^ \_exKey[ct++];

C4 = T0[R4 >> 24] ^ T1[(byte)(R5 >> 16)] ^ T2[(byte)(R7 >> 8)] ^ T3[(byte)R0] ^ \_exKey[ct++];

C5 = T0[R5 >> 24] ^ T1[(byte)(R6 >> 16)] ^ T2[(byte)(R0 >> 8)] ^ T3[(byte)R1] ^ \_exKey[ct++];

C6 = T0[R6 >> 24] ^ T1[(byte)(R7 >> 16)] ^ T2[(byte)(R1 >> 8)] ^ T3[(byte)R2] ^ \_exKey[ct++];

C7 = T0[R7 >> 24] ^ T1[(byte)(R0 >> 16)] ^ T2[(byte)(R2 >> 8)] ^ T3[(byte)R3] ^ \_exKey[ct++];

// rounds loop

while (ct < Nr \* Nb)

{

R0 = T0[C0 >> 24] ^ T1[(byte)(C1 >> 16)] ^ T2[(byte)(C3 >> 8)] ^ T3[(byte)C4] ^ \_exKey[ct++];

R1 = T0[C1 >> 24] ^ T1[(byte)(C2 >> 16)] ^ T2[(byte)(C4 >> 8)] ^ T3[(byte)C5] ^ \_exKey[ct++];

R2 = T0[C2 >> 24] ^ T1[(byte)(C3 >> 16)] ^ T2[(byte)(C5 >> 8)] ^ T3[(byte)C6] ^ \_exKey[ct++];

R3 = T0[C3 >> 24] ^ T1[(byte)(C4 >> 16)] ^ T2[(byte)(C6 >> 8)] ^ T3[(byte)C7] ^ \_exKey[ct++];

R4 = T0[C4 >> 24] ^ T1[(byte)(C5 >> 16)] ^ T2[(byte)(C7 >> 8)] ^ T3[(byte)C0] ^ \_exKey[ct++];

R5 = T0[C5 >> 24] ^ T1[(byte)(C6 >> 16)] ^ T2[(byte)(C0 >> 8)] ^ T3[(byte)C1] ^ \_exKey[ct++];

R6 = T0[C6 >> 24] ^ T1[(byte)(C7 >> 16)] ^ T2[(byte)(C1 >> 8)] ^ T3[(byte)C2] ^ \_exKey[ct++];

R7 = T0[C7 >> 24] ^ T1[(byte)(C0 >> 16)] ^ T2[(byte)(C2 >> 8)] ^ T3[(byte)C3] ^ \_exKey[ct++];

C0 = T0[R0 >> 24] ^ T1[(byte)(R1 >> 16)] ^ T2[(byte)(R3 >> 8)] ^ T3[(byte)R4] ^ \_exKey[ct++];

C1 = T0[R1 >> 24] ^ T1[(byte)(R2 >> 16)] ^ T2[(byte)(R4 >> 8)] ^ T3[(byte)R5] ^ \_exKey[ct++];

C2 = T0[R2 >> 24] ^ T1[(byte)(R3 >> 16)] ^ T2[(byte)(R5 >> 8)] ^ T3[(byte)R6] ^ \_exKey[ct++];

C3 = T0[R3 >> 24] ^ T1[(byte)(R4 >> 16)] ^ T2[(byte)(R6 >> 8)] ^ T3[(byte)R7] ^ \_exKey[ct++];

C4 = T0[R4 >> 24] ^ T1[(byte)(R5 >> 16)] ^ T2[(byte)(R7 >> 8)] ^ T3[(byte)R0] ^ \_exKey[ct++];

C5 = T0[R5 >> 24] ^ T1[(byte)(R6 >> 16)] ^ T2[(byte)(R0 >> 8)] ^ T3[(byte)R1] ^ \_exKey[ct++];

C6 = T0[R6 >> 24] ^ T1[(byte)(R7 >> 16)] ^ T2[(byte)(R1 >> 8)] ^ T3[(byte)R2] ^ \_exKey[ct++];

C7 = T0[R7 >> 24] ^ T1[(byte)(R0 >> 16)] ^ T2[(byte)(R2 >> 8)] ^ T3[(byte)R3] ^ \_exKey[ct++];

}

// Final Round

Output[0] = (byte)(SBox[C0 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[1] = (byte)(SBox[(byte)(C1 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[2] = (byte)(SBox[(byte)(C3 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[3] = (byte)(SBox[(byte)C4] ^ (byte)\_exKey[ct++]);

Output[4] = (byte)(SBox[C1 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[5] = (byte)(SBox[(byte)(C2 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[6] = (byte)(SBox[(byte)(C4 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[7] = (byte)(SBox[(byte)C5] ^ (byte)\_exKey[ct++]);

Output[8] = (byte)(SBox[C2 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[9] = (byte)(SBox[(byte)(C3 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[10] = (byte)(SBox[(byte)(C5 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[11] = (byte)(SBox[(byte)C6] ^ (byte)\_exKey[ct++]);

Output[12] = (byte)(SBox[C3 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[13] = (byte)(SBox[(byte)(C4 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[14] = (byte)(SBox[(byte)(C6 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[15] = (byte)(SBox[(byte)C7] ^ (byte)\_exKey[ct++]);

Output[16] = (byte)(SBox[C4 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[17] = (byte)(SBox[(byte)(C5 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[18] = (byte)(SBox[(byte)(C7 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[19] = (byte)(SBox[(byte)C0] ^ (byte)\_exKey[ct++]);

Output[20] = (byte)(SBox[C5 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[21] = (byte)(SBox[(byte)(C6 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[22] = (byte)(SBox[(byte)(C0 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[23] = (byte)(SBox[(byte)C1] ^ (byte)\_exKey[ct++]);

Output[24] = (byte)(SBox[C6 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[25] = (byte)(SBox[(byte)(C7 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[26] = (byte)(SBox[(byte)(C1 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[27] = (byte)(SBox[(byte)C2] ^ (byte)\_exKey[ct++]);

Output[28] = (byte)(SBox[C7 >> 24] ^ (byte)(\_exKey[ct] >> 24));

Output[29] = (byte)(SBox[(byte)(C0 >> 16)] ^ (byte)(\_exKey[ct] >> 16));

Output[30] = (byte)(SBox[(byte)(C2 >> 8)] ^ (byte)(\_exKey[ct] >> 8));

Output[31] = (byte)(SBox[(byte)C3] ^ (byte)\_exKey[ct]);

}

To summarize; this version (RDX) of Rijndael is based on a well-known and accepted implementation model, all I have done is to write it in a way that it accepts a longer key length and that the numbers of rounds processed is determined by the length of the expanded key. Both the Rijndael and AES specification documents makes it fairly clear that the authors of Rijndael designed the algorithm with extensibility in mind, to quote Section 6.3 of the AES specification document Fips 197:

*"This standard explicitly defines the allowed values for the key length (Nk), block size (Nb), and number of rounds (Nr). However, future reaffirmations of this standard could include changes or additions to the allowed values for those parameters. Therefore, implementers may choose to design their AES implementations with future flexibility in mind."*

So this is what I have written, a more flexible implementation, one that can accommodate the larger 512 bit key size. Will this larger key size make it vulnerable to certain attack vectors? Yes, just as the 256 bit key is vulnerable. The real question however, is will such an attack negate the full 256 bits of security added with the larger key? That is extremely unlikely, and if such an attack were devised, it would almost certainly have a devastating effect on the 256 bit key as well.. Rijndael has been around for some time now, and thoroughly scrutinized, the addition of 8 rounds of diffusion only makes the implementation stronger, the longer key length makes it more resistant to brute force attacks. But the weak key scheduler did give me some pause here, which is why I wrote RSX..

**RSX**

**Overview**

**RSX** is a hybrid of the Rijndael and Serpent encryption algorithms. Most encryption algorithms can be thought of as having two main parts; the key scheduler, and the diffusion algorithm.  The key scheduler takes a small amount of initial entropy, (the user key), and expands it into a larger working array that is used in the diffusion algorithm. Rijndael has what is considered a weak key scheduler; it relies on a strong diffusion algorithm to thoroughly whiten the input data. One of the strongest key schedulers is a part of the Serpent algorithm, (which was the 2nd place AES finalist). This joining of two algorithms has been done before; [Sosemanuk](http://www.ecrypt.eu.org/stream/e2-sosemanuk.html), an [eSTREAM](http://www.ecrypt.eu.org/stream/) cipher finalist uses a combination of Serpent and the stream cipher Snow. The key scheduler in Serpent is much more sophisticated than Rijndael, and does a better job at dispersing the initial entropy and eliminating weak and related keys. Quote from the authors of Serpent: “Serpent has none of the simpler vulnerabilities that can result from exploitable symmetries in the key schedule: there are no weak keys, semi-weak keys, equivalent keys, or complementation properties.”

The result is the best parts of both ciphers have been combined into a hybrid that can encrypt using up to a 512 bit key length.

My first version of RSX used SHA256 and HKDF Expand to generate the working key from the user key, I think this is a reasonable configuration if security is the primary consideration. I opted instead for the expansion routine from Serpent because; though it is not nearly as good at distributing the initial key entropy as a hash based PRNG, it is still one of the strongest key schedulers among block ciphers, and speed and portability were also important design considerations in this implementation.

**The Serpent Key Scheduler**

This implementation of the key scheduler is based on the version in the Bouncy Castle [SerpentEngine.cs](https://github.com/bcgit/bc-csharp/blob/master/crypto/src/crypto/engines/SerpentEngine.cs) class, an explanation of the algorithm can be found in the [Serpent documentation](http://www.cl.cam.ac.uk/~rja14/Papers/ventura.pdf). My implementation is considerably different from Bouncy Castle's version, (but on a 256 bit key the output is tested equivalent). I made several changes to the method to increase performance, to process the larger key size of 512 bits, and create the correct number of rounded keys. I have also made a change to the algorithm itself to take advantage of the larger 512 bit key to produce better overall dispersion by extending the polynomial primitive used in the key rotation; for a 256 bit key it is:

**w*i* :=(w*i*-8 ^ w*i*-5 ^ w*i*-3 ^ w*i*-1 ^ PHI ^ *i*) <<< 11**

For a 512 bit key this becomes:

**w*i* :=(w*i*-16 ^ w*i*-13 ^ w*i*-11 ^ w*i*-10 ^ w*i*-8 ^ w*i*-5 ^ w*i*-3 ^ w*i*-1 ^ PHI ^ *i*) <<< 11**

This extension of the polynomial creates a more even distribution of the key bits, while eliminating weak and related keys.

private uint[] SerpentKey(byte[] Key)

{

int ct = 0;

int pos = 0;

int index = 0;

int padSize = Key.Length / 2;

uint[] Wp = new uint[padSize];

int keySize = Key.Length == 64 ? 92 : 60;

// rijndael uses 2x keys on 32 block

if (this.BlockSize == 32)

keySize \*= 2;

// step 1: reverse copy key to temp array

for (int offset = Key.Length; offset > 0; offset -= 4)

Wp[index++] = BytesToWord(Key, offset - 4);

// initialize the key

uint[] Wk = new uint[keySize];

if (padSize == 16)

{

// 32 byte key

// step 2: rotate k into w(k) ints

for (int i = 8; i < 16; i++)

Wp[i] = RotateLeft((uint)(Wp[i - 8] ^ Wp[i - 5] ^ Wp[i - 3] ^ Wp[i - 1] ^ PHI ^ (i - 8)), 11);

// copy to expanded key

Array.Copy(Wp, 8, Wk, 0, 8);

// step 3: calculate remainder of rounds with rotating primitive

for (int i = 8; i < keySize; i++)

Wk[i] = RotateLeft((uint)(Wk[i - 8] ^ Wk[i - 5] ^ Wk[i - 3] ^ Wk[i - 1] ^ PHI ^ i), 11);

}

else

{

// \*extended\*: 64 byte key

// step 3: rotate k into w(k) ints, with extended polynomial primitive

// Wp := (Wp-16 ^ Wp-13 ^ Wp-11 ^ Wp-10 ^ Wp-8 ^ Wp-5 ^ Wp-3 ^ Wp-1 ^ PHI ^ i) <<< 11

for (int i = 16; i < 32; i++)

Wp[i] = RotateLeft((uint)(Wp[i - 16] ^ Wp[i - 13] ^ Wp[i - 11] ^ Wp[i - 10] ^ Wp[i - 8] ^ Wp[i - 5] ^ Wp[i - 3] ^ Wp[i - 1] ^ PHI ^ (i - 8)), 11);

// copy to expanded key

Array.Copy(Wp, 16, Wk, 0, 16);

// step 3: calculate remainder of rounds with rotating primitive

for (int i = 16; i < keySize; i++)

Wk[i] = RotateLeft((uint)(Wk[i - 16] ^ Wk[i - 13] ^ Wk[i - 11] ^ Wk[i - 10] ^ Wk[i - 8] ^ Wk[i - 5] ^ Wk[i - 3] ^ Wk[i - 1] ^ PHI ^ i), 11);

}

// step 4: create the working keys by processing with the Sbox and IP

while (ct < keySize - 32)

{

Sb3(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb2(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb1(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb0(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb7(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb6(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb5(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb4(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

}

// last rounds

Sb3(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb2(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb1(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb0(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb7(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

Sb6(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct++]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16); pos += 16;

// different offset on 16 block

if (this.BlockSize != 32)

{

Sb5(Wk[ct++], Wk[ct++], Wk[ct++], Wk[ct]);

Buffer.BlockCopy(\_registers, 0, Wk, pos, 16);

}

return Wk;

}

Just as with the Rijndael key scheduler, the user key bytes are first shifted into a temporary integer array; *wp*.  The bytes in that array then undergo a transformation using a variation of the key rotation polynomial before being added to the beginning of the working key array. The remainder of the pre-keys are then calculated and added to the working key array *wk*. These pre-keys are then processed through a series of SBox calculations, with the resulting registers being copied into the corresponding key positions.

If you compare this to Rijndael's key scheduler, it is clear that there is a great deal more processing in Serpents key scheduler; in Rijndael, the user key is first copied straight into the working key, whereas in this scheduler, it undergoes pre-processing first. Most of Rijndaels rounded keys are generated with the simple formula of k[i-1] ^ k[i-Nk], whereas Serpent uses the key rotation and then an SBox step on each key. The result is a much stronger expanded key, and one that is more resistant to various weak and related key attacks.

**DCS**

**Overview**

**DCS** is a stream cipher that uses two Rijndael streams in an AES configuration; that is a 256 bit key and 128 bit block size. It creates two AES SIC (Segmented Integer Counter) streams using unique keys and 128 bit counters. These two independent streams are combined using a logical exclusive OR operation (XOR) to produce a pseudo random output stream. That stream is than XOR’d again with the input data to produce the cipher text. DCS uses a single 768 bit key to generate the random stream, making it impervious to brute force attacks. It is also automatically parallelized, intended to run at high speed on multi processer systems.

**Initialization**

public void Init(byte[] Seed)

{

if (Seed == null)

throw new ArgumentOutOfRangeException("Invalid seed! Seed can not be null.");

if (Seed.Length != 96)

throw new ArgumentOutOfRangeException("Invalid seed size! Seed must be 96 bytes.");

if (ZerosFrequency(Seed) > 32)

throw new ArgumentException("Bad seed! Seed material contains too many zeroes.");

if (!EvaluateSeed(Seed))

throw new ArgumentException("Bad seed! Seed material contains repeating squence.");

// copy seed

Buffer.BlockCopy(Seed, 0, \_seedBuffer, 0, \_seedBuffer.Length);

byte[] keyBuffer1 = new byte[KEY\_BYTES];

byte[] keyBuffer2 = new byte[KEY\_BYTES];

// copy seed to keys

Buffer.BlockCopy(Seed, 0, keyBuffer1, 0, KEY\_BYTES);

Buffer.BlockCopy(Seed, KEY\_BYTES, keyBuffer2, 0, KEY\_BYTES);

if (keyBuffer1.SequenceEqual(keyBuffer2))

throw new ArgumentException("Bad seed! Seed material is a repeating sequence.");

// copy seed to counters

Buffer.BlockCopy(Seed, KEY\_BYTES \* 2, \_ctrBuffer1, 0, BLOCK\_SIZE);

Buffer.BlockCopy(Seed, (KEY\_BYTES \* 2) + BLOCK\_SIZE, \_ctrBuffer2, 0, BLOCK\_SIZE);

if (\_ctrBuffer1.SequenceEqual(\_ctrBuffer2))

throw new ArgumentException("Bad seed! Seed material is a repeating sequence.");

// expand AES keys

\_exKey1 = ExpandKey(keyBuffer1);

\_exKey2 = ExpandKey(keyBuffer2);

}

The Init() method tests for weak and invalid keys; this is because DCS requires a strong key, one that does not contain repeating sequences or high numbers of repeating bytes. Keys should be created with a hash function or PRNG as demonstrated in the class: Cryptographic\Helpers\KeyHeader.cs.

The ZerosFrequency() test counts the frequency of zero bytes in the 96 byte key, the EvaluateSeed() method tests for recurring byte frequency as well as the frequency of ascending 4 byte pattern runs in the seed.

The first 64 bytes of the seed or 'key' is then copied into two 32 byte key buffers. These buffers are first checked for equality before being used to create the two unique AES working keys (exKey1 and exKey2). The remaining 32 bytes is copied into two 128 bit segmented counters and checked for equality.

**Random Generation**

Where C is an incrementing counter, and P is the plaintext, the algorithmic representation of DCS could be expressed as:

**(Aes\_k1(C1) ^ Aes\_k2(C2)) ^ P**

The random generator portion of that is the method Generate(). It takes the expected return size in bytes, and two 128 bit counters as parameters, and returns a number of psuedo-random bytes. This p-rand is generated by calling the diffusion algorithm: AesTransform(c, t, k) twice, with unique keys and counters derived in the Init() method. The output from these calls (the encrypted counter byte arrays), is then Xor'd and added to the output array. Both segmented counters are incremented on each iteration of the processing loop.

private byte[] Generate(Int32 Size, byte[] Ctr1, byte[] Ctr2)

{

// align to upper divisible of block size

Int32 alignedSize = (Size % BLOCK\_SIZE == 0 ? Size : Size + BLOCK\_SIZE - (Size % BLOCK\_SIZE));

Int32 lastBlock = alignedSize - BLOCK\_SIZE;

byte[] randBlock1 = new byte[BLOCK\_SIZE];

byte[] randBlock2 = new byte[BLOCK\_SIZE];

byte[] outputData = new byte[Size];

for (int i = 0; i < alignedSize; i += BLOCK\_SIZE)

{

// encrypt counter1 (aes: ctr1, out1, key1)

AesTransform(Ctr1, randBlock1, \_exKey1);

// encrypt counter2 (aes: ctr2, out2, key2)

AesTransform(Ctr2, randBlock2, \_exKey2);

// xor the two transforms

for (int j = 0; j < BLOCK\_SIZE; j++)

randBlock1[j] ^= randBlock2[j];

// copy to output

if (i != lastBlock)

{

// copy transform to output

Buffer.BlockCopy(randBlock1, 0, outputData, i, BLOCK\_SIZE);

}

else

{

// copy last block

int finalSize = (Size % BLOCK\_SIZE) == 0 ? BLOCK\_SIZE : (Size % BLOCK\_SIZE);

Buffer.BlockCopy(randBlock1, 0, outputData, i, finalSize);

}

// increment counters

Increment(Ctr1);

Increment(Ctr2);

}

return outputData;

}

This combination of independent pseudo-random permutations has been explored for some years now, including [a](http://www.iacr.org/archive/eurocrypt2000/1807/18070476-new.pdf) paper by Stefan Lucks; [The sum of PRPs is a secure PRF](http://www.iacr.org/archive/eurocrypt2000/1807/18070476-new.pdf). That paper gives as a security bound for a sum of two independent PRPs q3 /22n−1, where q  is the number of queries and n the block size (i.e. 128 for AES).

This means that this combining of independent pseudo random streams is more secure than using a single PRP, for which the bound is q2 /2n. If you wanted to give an adversary an advantage at most 2−k   then you could use the sum 2(2n−k) /3 times. For e.g. k=64  that's 264, which should be enough (256 Exbibytes). In comparison, with a single PRP you could only use it 232 times (32 Gibibytes).

So aside from the longer key length providing more resistance against brute force attacks, this algorithm also has the advantage of providing a much longer period between necessary rekeying of the stream, which means larger data sets or streams can be safely encrypted with the same key. Another advantage is that because DCS is using an AES configuration, it could be ported to C and made to leverage the AES Instruction set; [AES-NI](http://en.wikipedia.org/wiki/AES_instruction_set).

**Parallel Processing**

The minimum input size that triggers parallel processing is defined as the MinParallelSize property, which is 1024 bytes. Data blocks of this size or greater will be processed in parallel using a [Parallel For](http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.for(v=vs.110).aspx) loop. The input data is sub divided into chunks divisible by the system processer count, with each chunk processed on its own thread inside the loop. The segmented counters are created in a jagged array inside the loop, with each counter member incremented to an offset equal to the chunk size multiplied by the value of the loop iterator. Two distinct counters are offset and passed to the Generate() method which returns the pseudo random output. That random array is then Xor'd with the input bytes at the corresponding offset to create the output.

public void Transform(byte[] Input, byte[] Output)

{

if (Output.Length < 1)

throw new ArgumentOutOfRangeException("Invalid output array! Size can not be less than 1 byte.");

if (Output.Length > Input.Length)

throw new ArgumentOutOfRangeException("Invalid input array! Input array size can not be smaller than output array size.");

int outputSize = Output.Length;

if (!this.IsParallel || outputSize < this.MinParallelSize)

{

// generate random

byte[] random = Generate(outputSize, \_ctrBuffer1, \_ctrBuffer2);

// output is input xor with random

for (int i = 0; i < outputSize; i++)

Output[i] = (byte)(Input[i] ^ random[i]);

}

else

{

// parallel ctr processing //

int count = this.ProcessorCount;

int dimensions = count \* 2;

int alignedSize = outputSize / BLOCK\_SIZE;

int chunkSize = (alignedSize / count) \* BLOCK\_SIZE;

int roundSize = chunkSize \* count;

int subSize = (chunkSize / 16);

// create jagged array of 'sub counters'

byte[][] counters = new byte[dimensions][];

// create random and xor to output in parallel

System.Threading.Tasks.Parallel.For(0, count, i =>

{

// offset first counter by i \* (chunk size / block size)

counters[i] = Increase(\_ctrBuffer1, subSize \* i);

// offset the second counter

counters[count + i] = Increase(\_ctrBuffer2, subSize \* i);

// create random with counter offsets

byte[] random = Generate(chunkSize, counters[i], counters[i + count]);

int offset = i \* chunkSize;

// xor with input at index offset

for (int j = 0; j < chunkSize; j++)

Output[j + offset] = (byte)(Input[j + offset] ^ random[j]);

});

// last block processing

if (roundSize < outputSize)

{

int finalSize = outputSize % roundSize;

byte[] random = Generate(finalSize, counters[count - 1], counters[dimensions - 1]);

for (int i = 0; i < finalSize; i++)

Output[i + roundSize] = (byte)(Input[i + roundSize] ^ random[i]);

}

// copy the last counter positions to class variables

Buffer.BlockCopy(counters[count - 1], 0, \_ctrBuffer1, 0, \_ctrBuffer1.Length);

Buffer.BlockCopy(counters[dimensions - 1], 0, \_ctrBuffer2, 0, \_ctrBuffer2.Length);

}

}

**Example Implementation**

The example implementation is a standalone encryptor. It is compatible with my commercial version, (just with far fewer features).

**Key and Vector Creation**

Keys and vectors are created using a combination of SHA and the .Net RNGCryptoServiceProvider. Using RngCrypto is more than sufficient for creating keying material, but in a document encryptor, the additional time expended creating the key is negligible, and the extra layer of security provided is a reasonable expense.

private static byte[] GetSeed32()

{

byte[] data = new byte[64];

using (RNGCryptoServiceProvider rngRandom = new RNGCryptoServiceProvider())

rngRandom.GetBytes(data);

using (SHA256 sha256Hash = SHA256Managed.Create())

return sha256Hash.ComputeHash(data);

}

**Key and Message Headers**

Both the key and message contain headers that provide some information to the encryptor. The key header contains fields that are used by the encryptor to determine settings; algorithm, cipher mode, padding scheme and block size. It also contains a unique 16 byte key identity field and 16 bytes of p-rand used to encrypt the file extension. The message header contains the identity field of the key used to encrypt the message, a 16 byte field that contains the encrypted file extension, and a 32 byte value that stores the hash of the decrypted plaintext.

**The Transform Class**

The Transform class is a wrapper for the encryption api. The class constructor takes the key file path as an argument, and uses the key header information to initialize the correct algorithm and settings. The class contains two public methods Encrypt() and Decrypt(), both methods take the input and output file paths as arguments.

**PSC (Parallel Segmented Counter)**

PSC is a parallel CTR mode that works in a way similar to DCS, it takes a segmented counter, and creates sub-counters offset at intervals equal to the chunk size \* the Parallel For loop iterator. These chunks which are a division of the input size / processor count, are then processed in parallel.

**Tests**

There are a number of different tests included in the project used to verify the validity of the RDX implementation in standard configurations, test mode validity, padding, and performance comparisons;

* **AESAVS** Known answer tests; the output from a transformation is known, given set parameters of key, iv, or plaintext. The full plaintext and key vectors from the AESAVS set, used for certifying an AES implementation are used, 960 vector tests in total.
* **Block Equality**; where random input is encrypted and the result compared to a known good implementation, in this case RDX with a 32 byte block compared to the .Net RijndaelManaged output.
* **Compare Engine**; three versions of the diffusion algorithm are compared for processing times: Bouncy Castle's FastAesEngine, the Mono engine, and RDX.
* **Fips 197**; the complete set of known answer tests from the AES specification appendix are tested.
* **I/O tests**; Known answer tests are applied to each method access, as well as a series of standard I/O tests.
* **Monte Carlo**; a known answer test where input and output are the same variable, which is encrypted 10 thousand times in succession and compared to a known final value. These are tests based on those written by [Dr Brian Gladman](http://www.gladman.me.uk/AES/), and considered standard in AES implementation testing.
* **Mode tests**; the padding modes PCS7S and x9.23 are tested by comparing outputs with RijndaelManaged.
* **PSC Equality**; this compares the outputs of random data encrypted using the parallel CTR mode with that of the standard SIC mode for equivalence.
* **Rijndael Vector**; known answer tests using test vectors derived from Bouncy Castle RijndaelTest.cs and the [Nessie unverified](https://www.cosic.esat.kuleuven.be/nessie/testvectors/bc/rijndael/Rijndael-256-256.unverified.test-vectors) vectors.
* **SIC Vector**; known answer tests vectors based on NIST Special Publication 800-38A used to test SIC counter mode implementations.
* **Speed Test**; RDX using an AES configuration in CBC mode is compared to RijndaelManaged.

**Conclusion**

In the spring of 1946 work on the ENIAC computer was completed. Newspapers around the globe heralded it as the “Super Brain” and the “Einstein Machine”. It was the most powerful computer ever built; and had more than 17,000 vacuum tubes, 7,200 crystal diodes, 1,500 relays, 70,000 resistors, 10,000 capacitors and around 5 million hand-soldered joints. It weighed 30 tons, took up 1800 square feet, and consumed 150 kW of power. It was capable of calculating an astounding 385 multiplication operations per second.

Imagine that you were one of the designers, out for a few pints with fellow engineers and scientists, and you proposed that in just 25 years, anyone could walk into a Sears store and buy a 10 dollar portable calculator with thousands of times the computational power and speed. I think you would have been greeted with much skepticism; ‘impossible’, ‘infeasible’, ‘transistors and circuit pathways cannot be made that small’.. and you would have been subjected a barrage of scientific theories positing that such a thing could never happen.. at least, not for a hundred years or so..

In a recent article on [wired](http://www.wired.com/2014/09/martinis/), John Martinis, one of the foremost experts on quantum computers, states that one of the objectives of the new Google Quantum AI lab, is to double the number of qubits each year. Recently another breakthrough in how quantum states are measured could prove to be [350 times faster](http://www.livescience.com/47684-calculating-quantum-wave-functions-faster.html) than current methods.. Breakthroughs of this kind are happening ever more frequently as our understanding of quantum processes continues to grow. So, at this ever accelerating rate, how long will it be before computers exist that will be capable of the enormous processing power required to break current encryption technology? It is impossible to say with any certainty, but a major breakthrough could put this in reach, possibly much sooner than expected. So when you hear of the improbability of brute forcing a 256 bit key, remember the ENIAC, and consider how far we have advanced technology in the last 100 years.

There is the argument *against* stronger encryption, often linked to the idea that state agencies are developing a mass surveillance apparatus for our own protection, that facilities like this one in [Utah](http://en.wikipedia.org/wiki/Utah_Data_Center), will be used only to target criminals and terrorists, and that strong encryption hampers their efforts. I think most people understand that this is not strictly the case, that the technology is being forged into a system of people control, and that terrorists either don't use electronic communications at all, or they have access to more sophisticated methods like One Time Pads and steganography so.. not a very honest, or compelling argument.

I think that we have advanced so quickly over the past 100 years because we are living in an age of unparalleled personal freedoms, freedom to express our ideas and communicate them without fear of interference or reprisal. This has been a chief driver in the forward progression of our society, and these freedoms need to be preserved if we are to maintain that forward momentum, and hopefully, create a better society for future generations. Encryption technologies play a pivotal role in that future, and I believe we should be striving towards technologies that protect information for the full span of a human lifetime, that all forms of electronic communication should incorporate strong encryption technology as a matter of standard, and that these technologies should constantly be compared to, and evolved against the projected rate of technological change.