AES Applied In Web Technology

Kyle Russell
AUT University
November 16, 2015

Abstract
This report investigates the design and construction of the Advanced Encryption Standard (AES) algorithm, with a focus on its role in modern web applications. AES provides an efficient, computationally unbreakable solution to the symmetric key encryption problem. Despite efforts to ensure such security, AES is constrained by the inherent nature of symmetric key cryptography and as such, omission of authentication and private key establishment yields the algorithm impractical for modern applications. To address these deficiencies, we explore the use of asymmetric and digital signature algorithms in conjunction with AES, culminating in a complete client side solution.

Introduction
The Advanced Encryption Standard (AES) algorithm otherwise known as Rijndael, was proposed by a pair of Belgian cryptographers, Joan Daemen and Vincent Rijment. Rijndael published in September 1999, where the 128-bit member of the Rijndael family serves as the US standard for data encryption (NIST 2001), ISO/IEC 18033-3 standard, federal govt. standard and to date, no successful or computationally feasible attack on AES exists. It resides as the most popular symmetric-key algorithm in the world with wide use in commercial applications including Skype, TSL/SSL, SSH, router firmware and in particular, networking due to its low latency, high throughput.

The AES specification permits varying key length of 128bit, 192bit and 256bit in which the largest of the three members provides optimal security at the cost of performance and conversely, the smallest key length offers the best performance however, theoretically it observes the worst security of the three despite no computationally feasible brute force related attacks on AES-128 existing. The choice of key length will certainly become more crucial with the advancement of hardware including the introduction of quantum computing attacks and perhaps much sooner than anticipated, with many commercial companies already progressing towards 256-bit length keys.

AES is considered a block cipher, such that we process a fixed length block of input data containing contiguous bits where in the case of AES, the block length is restricted to 128bits (16 bytes). Structurally, AES operates on a substitution-permutation network where the internal components are performed for a fixed number of rounds relative to the key length. More importantly, this structure allows AES to process the entire block at each round instead of processing only a partition of the block in a round as other block cipher structures such as the Fiestel Network observe and this provides a clear performance advantage as there exists fewer rounds and consequently fewer iterations of the internal components.

Background and motivation

The Data Encryption Standard (DES) is the predecessor of AES and as such, it plays an important role in the motivation behind AES and therefore holds a crucial place in the timeline of events prior to the standardization of AES in November 2001.

DES was published in 1975, standardized in 1979 and designed by IBM with NSA involvement. The conception of DES interestingly came from a government call by the National Bureau of Standards (NBS) in 1972 for standardized encryption, which during a time of secrecy in the cold war period was quite unusual. The development process was closed and there was much secrecy in the details of the implementation, particularly in the design of its eight substitution boxes in the underlying Fiestel Cipher. Such secrecy lead to conspiracies in part due to NSA involvement, that DES contained backdoors residing in the substitution boxes. This lead to a differential cryptanalysis attack on the s-boxes where it was discovered that minor changes to the s-boxes significantly weakened the strength of DES. Incidentally, after publication (Coppersmith 1994) of the s-box design, it was revealed IBM's awareness of the attack long before it was brought to the public and additionally, it was found that the design of the s-boxes was to prevent such an attack.

DES is a symmetric block cipher designed on the Fiestel network offering a fixed 56bit key size and 64bit block size with 16 rounds. Due to known theoretical attacks and significantly small key size, DES is no longer secure and has since been replaced by Triple DES (3DES) in 1998 which offers a simple solution to the problem of key size by performing three iterations of DES on a single block. To stress the weakness of the minimal DES key size, a publically collaborated attack on the DES private key was able to be performed in 22 hours and 15 minutes by Distributed.net and Electronic Frontier Foundation in January 1999. This attack although not recent, gives perspective on the weakness of DES and the importance of key size in a symmetric-key cipher and while no more recent, large scale attacks on a DES key have been performed, we would likely observe much more significant results with the use of modern hardware.

In January 1997 the National Institute of Standards and Technology (NIST) would oversee the process of selecting the successor to DES. 3DES offered a solution to DES, however it was yet to be published and although it came to light during the four year selection process of AES, the 3DES solution was particularly inefficient in software implementation as it essentially ran DES three times for a single block where DES was already impractical for software implementation as it was inherently designed for hardware and therefore 3DES was not a candidate. NIST offered a transparent process of selection where submissions of algorithms were open to the public. The choice of a public selection process received very positive responses from the wider community and understandably as public scrutiny of the candidates allowed for rigorous testing in attempts to break the algorithms by the public that would otherwise not be possible in a closed process. The selection process saw three evaluation rounds where initially there were fifteen candidates chosen and narrowed down to five finalists by August 1999. Requirements for submission included the support for three varying key lengths: (128bit, 192bit and 256bit), block cipher with 128bit block length and efficiency in both hardware and software. The five finalists: Mars, RC6, Rijndael, Serpent and Twofish are all considered secure and perfectly acceptable in software implementation. However on October 2nd 2000 NIST announced Rijndael as the choice for AES and on November 26 2001 AES was approved as the US standard.

Structure preliminary I: Finite Fields in AES

Finite Fields otherwise called Galois Fields, are an integral component in AES and their existence can be found in many of the AES underlying internal layer structures.

We denote $GF(p^n)$ for some prime p and integer n, as the set $\{0,..p^n-1\}$ with finite prime cardinality, where p is known as the characteristic. The set is closed under addition, subtraction, multiplication and division where subtraction and division operations can be realized through their additive and multiplicative inverse respectively. Addition of elements in the set p^n is performed modulo p^n, subtraction can be observed as the additive inverse such that $-x = x + (-x) = 0 \ mod \ p^n$. Secondly, multiplication within the set p^n can be performed modulo $P(x)$ which we denote as the irreducible polynomial and similarly, division in the set p^n can be done by computing the multiplicative inverse

In AES we assume $GF(2^8)$ with characteristic 2 as the sum of polynomials $(a^7, a^6, a^5, a^4, a^3, a^2, a^1, a^0)$ which can be represented by 8 bits in software implementation where $\forall a^i \in \{0,1\}$ e.g. $(x^7 + x^2 + x + 1) \cong (10000111)$
For addition in $GF(2^8)$ we observe this as the bitwise exclusive-OR otherwise called the XOR operation denoted by the $\oplus$ gate where we simply perform polynomial addition modulo 2. For simple addition of two bits this can be seen as $1 \oplus 1 = 0$ as we have $1+1 = 0$ mod 2 and similarly, $1 \oplus 0 = 1$ as $1 + 0 = 1$ mod 2. This holds for 8 bit addition for example:
$(x^5 + x^2 + x + 1) + (x^2 + x + 1) = (00100111) \oplus (00000111) = (00100000)$

For multiplication in $GF(2^8)$ we perform multiplication of the polynomials modulo $P(x)$, where AES uses the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. For multiplication of elements <= 0x03 it is quite intuitive where for elements 0x00, and 0x01 this is simply multiplication by 0 and multiplication of the identity. For elements 0x02 and 0x03 we can observe this is multiplication by x and multiplication by $x + 1$. For the case of 0x02 this can be dealt with in implementation by performing a bitwise left shift by one e.g. $(00100111) << 1 = (01001110)$. For multiplication by 0x03 it is necessary to perform a bitwise left shift by one (multiplication by x) and XOR the product by the identity (addition of 1). E.g. $(00100111) << 1 = (01001110) = (01001110) \oplus (00100111) = (10000000)$. We need to also consider if the left-most bit a^7 of the identity is set, in this case the product after performing multiplication in either 0x02 or 0x03, will certainly be carrying bits and needs to be reduced and we do so by reducing by the irreducible polynomial $P(x)$ and in implementation this is done by an XOR of 0x1b on the product. For software implementation the below pseudo code can be used to compute the multiplication of two bytes a, b in $GF(2^8)$ where b is <= 0x03

```
Multiply(a, b)
{
       if(b == 0x00) return 0x00 // multiplication by 0

       if(b == 0x01) return a //multiplication by the identity

       byte product = a << 1 //multiply by x (left-shift by one)

       if(b == 0x03) product = product ⊕ a  //multiply by x + 1

       if(a[7] == 1) product = product ⊕ 0x1b //carrying, reduce

       return product
}
```

For multiplication of elements $>$ 0x03 in $GF(2^8)$, we cannot only perform multiplication by x and $x+1$ using the above methods and it is necessary to consider other possibilities. One solution is offered by the fast multiplication algorithm, otherwise called 'Peasants algorithm' of Russian and Egyptian origin dating back centuries. The algorithm in order to perform the multiplication of two integers a, b essentially iterates for the number of bits in the input, in the case of AES is 8bits where each iteration we check if the right-most bit b^0, of b is set, if so we XOR the product by a. Next we perform a bitwise right-shift by one on b. The following step performs a left shift by one on a and we need to know beforehand if the left-most bit is set as it will be necessary to perform a reduction on a since the shift will result in a carrying a bit. For software implementation we can observe the pseudo code below for computing multiplication of elements $>$ 0x03 in $GF(2^8)$ using a variant of Peasants algorithm suitable for byte input and manipulation.

```
Multiply(a, b)
{
	bool carry
	byte product

	for i in 0..8 do:
		//if right-most bit of b is set: xor product with a
		if(b[0] == 1) product = product ⊕ a

		//shift b one bit right
		b >>= 1

		//identify if left-most bit of a is set
		//following step will cause a to carry a bit
		carry = (a[7] == 1)

		//shift a one bit left
		a <<= 1

		//if carry was true (left-most bit of a is set)
		//then reduce by P(x) (XOR by 0x1b)
		if(carry) a = a ⊕ 0x1b

	return product
}
```

Structure preliminary II: State

The AES specification requires a fixed block with length 128bits or 16 bytes. We can realize the block as a 4x4 matrix A called the State, consisting of 1 byte or 8bit entries $a_0, a_1, .. a_{15}$. Bytes are added column first and operations on the State are done column first where the structure of the State can be visualized by figure 1. For sub-keys of 128bit length we use the same matrix structure shown in figure 1, however for 192bit length there exists an additional column for entries $a_{16}, .. a_{19}$ and for 256-bit length keys we have two additional columns for entries $a_{16}, \dots a_{23}$.

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}
a_2	a_6	a_{10}	a_{14}
a_3	a_7	a_{11}	a_{15}

Figure 1: AES State

We perform the AES internals on a block for a fixed number of rounds relative to the key size as shown in figure 2. The relative increase in the number of rounds is due to the increased number of sub-keys derived from the input private key in the key schedule, such that more rounds are needed to process the increased sized key and its sub-keys.

Key size	No. rounds
128bit	10
192bit	12
256bit	14

Figure 2: No. rounds relative to key size

Structure preliminary III: AES Substitution boxes

Substitution boxes or otherwise called s-boxes, in AES are used in frequently in the AES internal components including the ByteSubstitution layer as well as its inverse layer and additionally in the key schedule's core function. S-boxes existed previously in DES featuring 8 distinct s-boxes, however much of the design of these s-boxes was wrapped in secrecy, lacking any mathematical back-bone and was therefore more of a specification. In AES there exists a single s-box (and its inverse) which enjoys some interesting mathematical properties where values in the s-box are constructed by computing the multiplicative inverse in $GF(2^8)$ followed by an affine mapping.

Pre-computation of the AES s-box can be realized in implementation by a 16x16, 256bit lookup table as shown in figure 3.

```
unsigned char sbox[16][16] =
{
        { 0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76 },
        { 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0 },
        { 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15 },
        { 0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75 },
        { 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84 },
        { 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF },
        { 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8 },
        { 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2 },
        { 0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73 },
        { 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB },
        { 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79 },
        { 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08 },
        { 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A },
        { 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E },
        { 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF },
        { 0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16 }
};
```

Figure 3: AES s-box

Retrieval of the s-box entries is a bijective mapping $S(A) = B$ performed in $GF(2^8)$ for an 8bit input A where the row of the corresponding s-box entry is the decimal value of the left-most bits in A, $(a_7, a_6, a_5, a_4)_{10}$ and similarly, the column of the s-box entry can be seen as the decimal value of the right-most bits in A, $(a_3, a_2, a_1, a_0)_{10}$.

Alternatively, one may choose to compute the s-box constructed values on the fly where there is concern for minimal memory footprint which is a frequent scenario when dealing with embedded systems. Similarly, if one is concerned about low latency, particularly in networking and online services then typically pre-computation of the s-boxes would be desirable. To compute the s-box value B for a given input A, we need to first compute the inverse $B` = A^{-1}$ such that $A^{-1} \cdot A = 1 \; mod \; P(x)$ where the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. The inverse may also be pre-computed by using an additional 256bit lookup table containing the pre-computed inverse values for $GF(2^8)$ however similar consequences of memory footprint exist and careful consideration should take place when choosing the inverse lookup table over the standard s-box which has already applied the affine mapping step. Finally to compute the resulting s-box value B after computing the inverse of A such that $B` = A^{-1}$, we apply the affine mapping on $B`$ shown in figure 4 where additions are performed as bitwise XOR's and multiplication is done in $GF(2^8)$.

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \bmod 2.$$

Figure 4: Affine mapping on B` to compute B

In the case of decryption we need to perform an inverse byte substitution operation and cannot be done using the s-box and affine-mapping solution provided. Instead we need to compute the inverse of the s-box values in $GF(2^8)$. Conveniently, a 256bit inverse s-box table exists for the purpose of pre-computation and is shown in figure 5.

```
unsigned char inverseSbox[16][16] =
{
        { 0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB },
        { 0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB },
        { 0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E },
        { 0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25 },
        { 0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92 },
        { 0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84 },
        { 0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06 },
        { 0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B },
        { 0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73 },
        { 0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E },
        { 0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B },
        { 0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4 },
        { 0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F },
        { 0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF },
        { 0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61 },
        { 0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D }
};
```

Figure 5: AES inverse s-box

Retrieval methods used for the s-box can be similarly applied for the inverse s-box. The clear distinction is in the computation of the values and we can observe that due to the bijection, that enables the inversion of the s-box values such that $A = S^{-1}(B)$ and recall $B = S(A)$. To invert the s-box transformation we first apply the inverse affine mapping shown in figure 6 (where addition and multiplication are performed in $GF(2^8)$) to compute $B`$ and recall $B` = A^{-1}$ therefore we can we compute the inverse $B`^{-1}$ in $GF(2^8)$ to compute $A$ as we have $B`^{-1} = (A^{-1})^{-1} = A$.

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} \equiv \begin{pmatrix} 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \\ 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \mod 2,$$

Figure 6: Inverse affine mapping

Structure preliminary IV: Key schedule

The key addition layer in the AES internal components adds the sub-key at round i for $n_r + 1$. The $n_r + 1$ sub-keys are distinct and produced in the AES key schedule. The key schedule derives n_k sub-keys from the input private key where the number of sub-keys $n_k = n_r + 1$ given the number of rounds n_r relative to the input key size (see figure 2). Note that we produce an additional sub-key despite only n_r rounds as there is an extra key addition layer at round 0 (encryption) or round n_r (decryption). This report assumes the use of a 128bit key and as such the focus is on the key schedule for a 128bit length key as there exists clear distinctions in the key schedules between each of the three versions that require equal attention.

Operations and storage in the key schedule are word (1 word = 32bits or one column in the state) oriented such that we store the produced sub-keys in a 1D word key expansion array W, with length $4n_k$ so for example the bytes of the first sub-key (the input key) are the first 4 entries of the W. This choice of structure is quite inconsistent with the rest of the AES structure as we primarily work with bytes not words, so to preserve consistency a second byte oriented option can be used and will be the assumed structure throughout this report, where the key expansion array W is now a byte array with length $16n_k$ such that a sub-key K_i are added and retrieved in W by their position W[16i] and offset W[16i + 16] with $K_i = W[16i]..W[16i + 16]$. The complete structure of the key expansion process is shown below in figure 7.

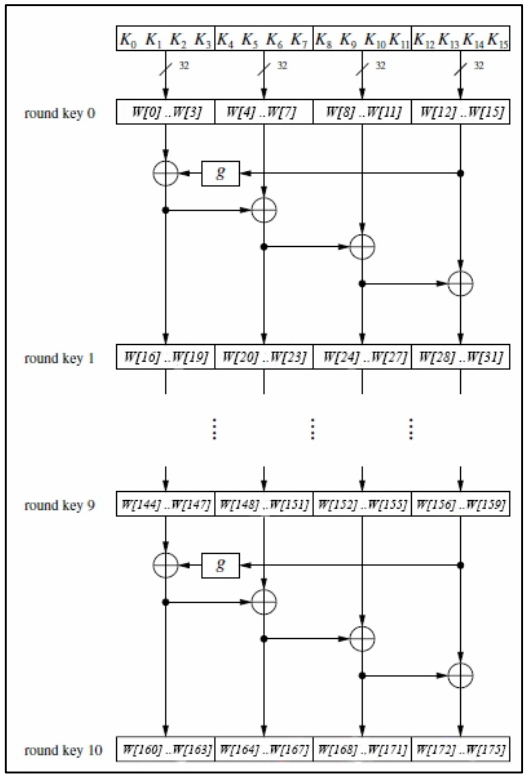

Figure 7: Key expansion structure

The key schedule computes the n_k sub-keys recursively, with the initial sub-key S[0]..S[15] being the input key itself. Subsequent keys are computed by generating 4 bytes of key each iteration until the key expansion array W is full (have n_k keys and size(W) = $16n_k$). In a single iteration we copy the previous 4 bytes in W into a 4 byte array $next$, that we bitwise XOR with the first 4 bytes of the previous sub-key.

Notice the presence of the function g in figure 7. Every 16 bytes (next sub-key) created by the key expansion procedure, we perform the key expansion core function and increment an integer pointer i known as the round coefficient index (rcon index). The core function takes as input a 4 byte array a and an integer i, where we pass the $next$ 4 byte array containing the previous 4 bytes and the rcon index i. The internals of the core function perform a rotate function which circularly left shifts the input a by one byte. Secondly, it performs a byte substitution on a with its corresponding values in the AES substitution box or alternatively assigning

the computed values. Lastly, the function bitwise XOR's the left-most byte $a[0]$, in a with the round coefficient at index i where the round coefficients rcon[i] for $0 \leq i < 10$, can be substituted from the set in figure 8. Up to 256 coefficients exist, however for 128bit key length there are 11 sub-keys where we only perform the core function once for each sub-key produced and so consequently we only need the coefficients rcon[1]..[10] (Note that we do not use the first coefficient, rcon[0]).

rcon = { 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36 }

Figure 8: Round coefficients for rcon[0]..rcon[10]

The rcon values can also be computed on the fly and is essentially exponentiation of 2 in $GF(2^8)$ such that for $1 \leq i < 256$, $rcon(i) = x^{i-1} \; mod \; P(x)$ where the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$. For example $rcon(3) = x^{3-1} = x^2 = (00000100)_2 = 0x04$.

With this intuition the pseudo code for the key expansion can now be observed:

```
key_expansion(Key privatekey)
{
      //input: 4x4 matrix with 128bits (state)
      //round 0 sub-key is the private key itself
      copy all 16 bytes of privatekey into W[0]..W[15]

      int i = 1 //rcon index
      int front = 16 //W index pointer
      Byte next[4] //next 4 bytes of key to be created

      //create up to n_k bytes with 11 sub-keys
      while (front < W.length) do:

            //copy previous 4 bytes into next
            for j in 0..4 && k in (front-4)..front do:
                  next[j] = W[k]

            //perform core on next every 16 bytes
            if(front % 16 == 0) do
                  core(next, i)
                  i++

            //set next 4 bytes in W
            for j in 0..4 do
                  W[front] = W[front – 16] ⊕ next[j]

}
```

Additionally the pseudo code for the core function used in the key expansion is:

```
core(byte[4] a, int i)
{
      //circular left shift a by one byte
      rotate(a)

      //apply s-box on a
      //where getSboxEntry computes row, col of s-box and
      //returns the corresponding s-box entry
      for j in 0..4 do
            a[j] = getSboxEntry (a[j])

      //add the round coefficient at index i to a[0]
      a[0] = a[0] ⊕  rcon[i]
}
```

Encryption structure

AES operates under a substitution permutation network such that we process the entire block at each iteration for a fixed number of rounds relative to the input key size. The encryption process takes as input, the 128bit plain text block and after n_r successive rounds, outputs the encrypted 128bit cipher block. The cipher observes a byte oriented structure and the movement behaviour of the block can be realized by the 4x4 128bit state matrix proposed in Structure Preliminary II.

AES has four internal components that will be investigated in subsequent sections:

- Byte Substitution layer
- Mix Columns layer
- Shift Rows layer
- Key Addition layer

All four components are performed on rounds $1..n_r$, with only a single Key Addition layer on round 0 (recall production of $n_r + 1$ sub-keys in key schedule) and interestingly, omission of the Mix Columns layer on round n_r where it was believed designers could not identify any security implications by omitting the Mix Columns layer in the final round. For a single round we begin with the Byte Substitution layer which substitutes each byte in the state with another byte in the s-box, then use the resulting output bytes as input for the Shift Rows layer where rows 1..3 are circularly shifted followed by performing the transformation in Mix Columns and lastly an addition of the sub-key at round i in the Key Addition layer.

AES provides an excellent source of confusion in its Byte Substitution layer such that a single change in the key followed by passage through the Byte Substitution layer sees a change in every output byte which is then propogated through the rest of the round. Additionally, AES introduces diffusion through its Mix Columns and Shift Rows layer where Mix Columns is the primary source of diffusion in the cipher as each input byte influences four output bytes. The complete encryption structure for rounds $0..n_r$ is shown in figure 9.

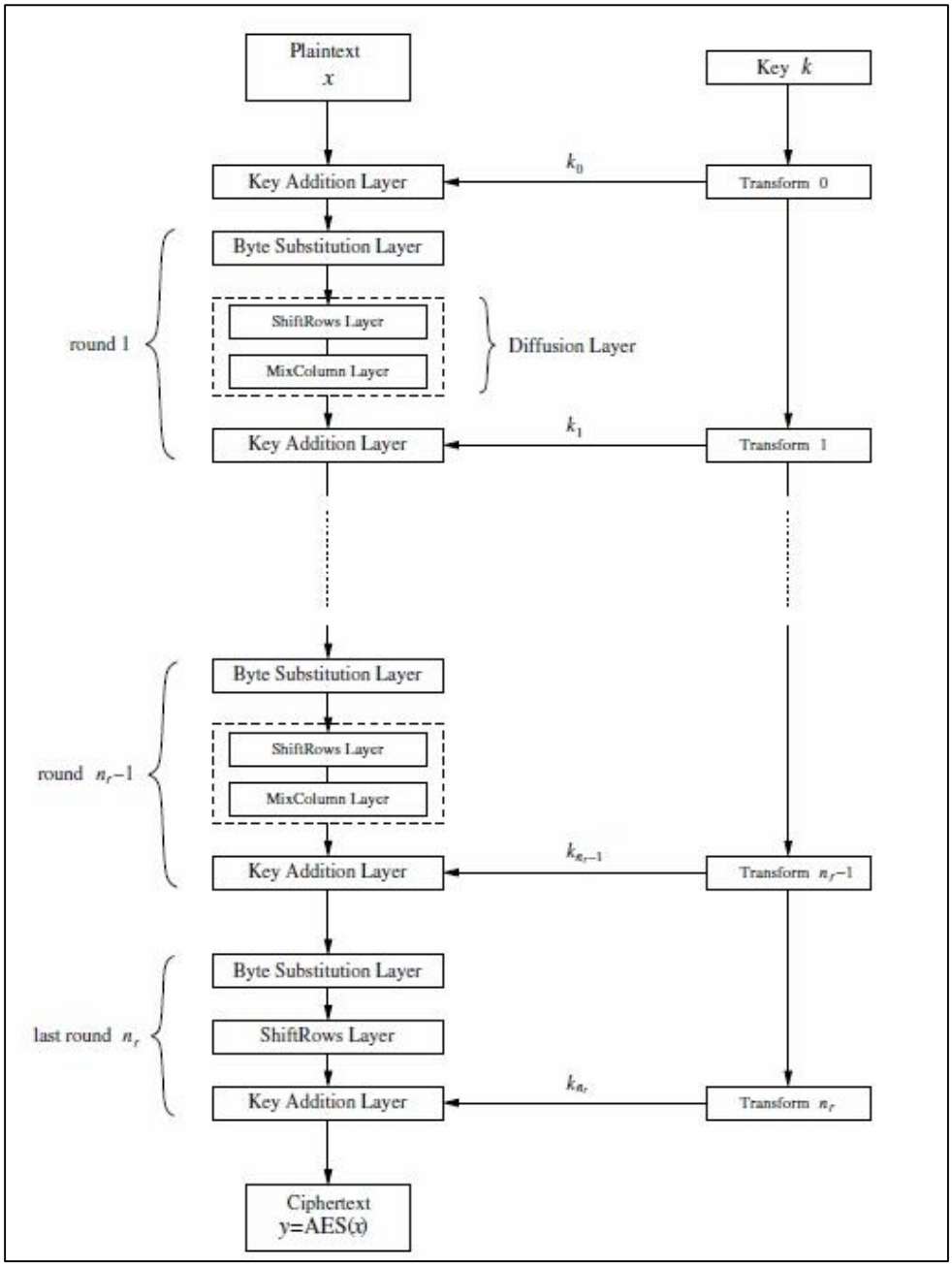

Figure 9: AES encryption structure

Byte Substitution layer

The Byte Substitution layer applies the s-box to each byte in the input State such that each byte in the State a_i is substituted with another byte b_i from the AES substitution boxes where $S(a_i) = b_i$ or otherwise computing the values on the fly using the methods described in Structure Preliminary III. For software implementation assuming pre-computation of the s-boxes, we only need to compute the s-box row and column for a given input a_i and this can be done by computing the decimal value of the left-most bits of a_i and similarly for the column, by computing the decimal value for the right-most bits in a_i. Once the row and column have been computed the remainder of the process involves a simple table lookup and substitution in the state.

Shift Rows layer

The Shift Rows layer is a diffusion component and helps to permutate the input bytes around by performing a circular left-shift on the rows 1..3 of the passed State by 1..3 bytes respectively. Figures 10-11 show the transformation of Shift Rows on the input State.

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}
a_2	a_6	a_{10}	a_{14}
a_3	a_7	a_{11}	a_{15}

Figure 10: Input State

Shift Rows →

	a_0	a_4	a_8	a_{12}
1	a_5	a_9	a_{13}	a_1
2	a_{10}	a_{14}	a_2	a_6
3	a_{15}	a_3	a_7	a_{11}

Figure 11: State with shifted rows 1..3

Mix Columns layer

In the mix columns layer we aim to add further diffusion to the cipher by performing a linear transformation that mixes the columns of the input State. We multiply each 32bit column of the State by the matrix in Figure 11, where additions are bitwise XOR's and multiplications are performed in $GF(2^8)$ where methods of multiplication can be obtained from Structure Preliminary I. The matrix chosen was a design choice with the intention of making the software implementation of the Mix Columns layer easier and with minimal values <= 0x03 we do not have to perform the Peasant's multiplication algorithm proposed in Structure Preliminary I and can use the first multiplication algorithm.

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

Figure 4: Mix column matrix

Key Addition layer

The Key Addition layer otherwise known as Key Whitening, adds the input State and sub-key at round i where the sub-keys are derived from the private key and accessible from the key schedule. Addition is performed in $GF(2^8)$ (bitwise XOR's) and can be observed as standard matrix addition where each entry $a_{i,j}$ of the State after the key addition is the sum of the State entry $a_{i,j}$ and corresponding sub-key entry $b_{i,j}$.

Decryption structure

The AES decryption takes as input, the 128bit cipher block produced by the AES encryption function and outputs the original 128bit plain text block. Since we are operating in a substitution permutation network and not a Fiestel Network, it is necessary to invert the layers performed and in doing so, the order of operations is reversed such that we begin the decryption from n_r and stop at round 0. Additionally the order of internal operations in each round is reversed and the reversed order of the components holds for the structure entirely as we can observe in figure 13, no Mix Columns being performed on the first round of decryption and as did in the last round of encryption as shown in figure 9. Similarly we have an extra key addition layer in the last decryption round to correspond to the round 0 of encryption.

For all but the key addition layer, we perform the internal components inverse such that we have the following layers of interest in decryption:

- Inverse Byte Substitution
- Inverse Mix Columns
- Inverse Shift rows

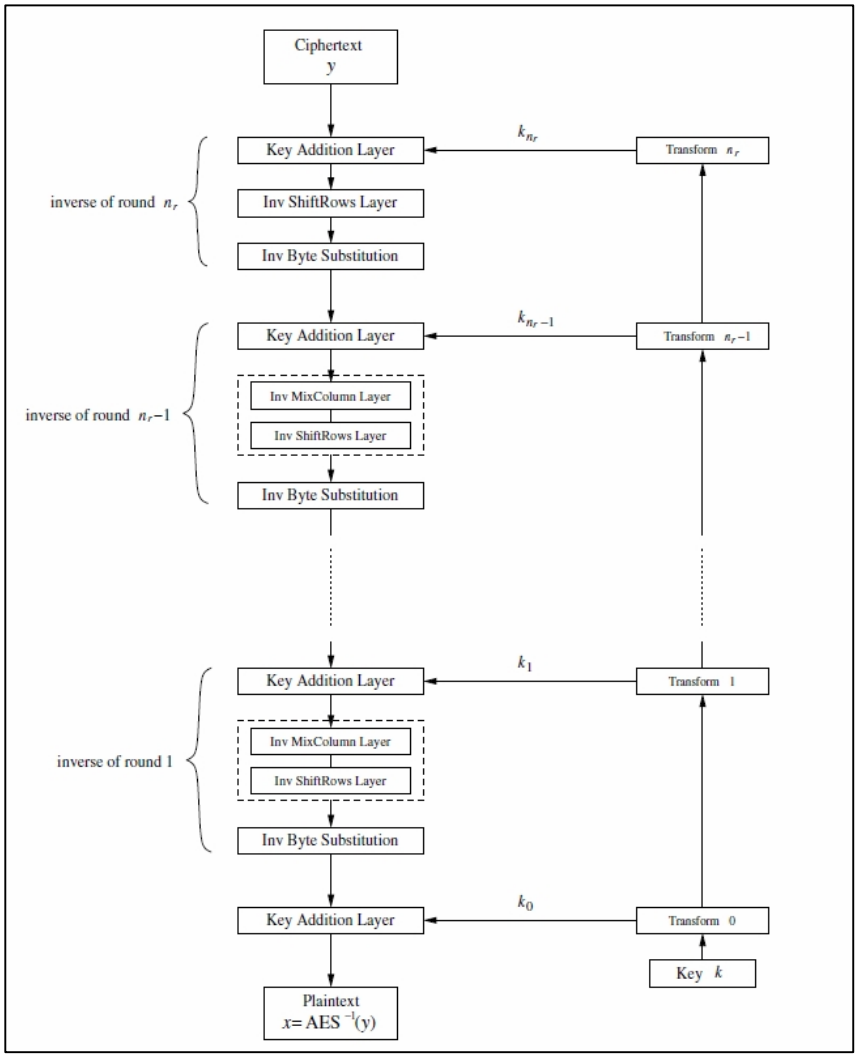

Figure 13: AES decryption structure

Inverse Byte Substitution layer

In order to invert the Byte Substitution layer which applied the AES s-box on each entry in the input State, we need to apply the inverse s-box (shown in figure 5 of Structure Preliminary I) on each entry. The method of substitution using the 256bit lookup tables uses the same operations as in encryption where the only difference is in the pre-computed values we substitute. The computation of the inverse s-box values is discussed in more detail in Structure Preliminary I, but essentially we need to compute the inverse affine mapping for $b`_i$, followed by computing the inverse of $b`_i$ to give us the initial input A.

Inverse Shift Rows layer

The inverse of the Shift Rows layer still performs circular left shifts on the rows 1..3 however the distinction is in the order that we shift the rows such that we shift the rows 1..3 of the input, 3..1 bytes left instead of 1..3 bytes left. Figures 14-15 show the transformation of inverse shift rows on the State.

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}
a_2	a_6	a_{10}	a_{14}
a_3	a_7	a_{11}	a_{15}

Figure 14: State before Inverse Shift Rows

Inverse Shift Rows →

	a_0	a_4	a_8	a_{12}
3	a_{13}	a_1	a_5	a_9
2	a_{10}	a_{14}	a_2	a_6
1	a_7	a_{11}	a_{15}	a_3

Figure 15: State after Inverse Shift Rows

Inverse Mix Columns layer

To invert the Mix Columns layer we need to perform the inverse transformation on the State and this is done by multiplication on the inverse of the matrix used in the initial linear transformation shown in figure 16. Additions and multiplications of entries are still performed in $GF(2^8)$. For multiplication of larger elements (> 0x03) in $GF(2^8)$, it is necessary to use the Peasants algorithm provided in Structure Preliminary I.

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$$

Figure 16: Inverse mix column matrix

Padding and Modes of Operation
AES is constrained to a fixed block size of 128bits which is fine for the case in which input has equal block length however the scenario of the input block containing less bits and conversely, the input block containing more bits and more generally, not knowing the length of the input are inevitabilities that are not accommodated by AES on its own. Consequently, it is necessary to investigate two possible solutions: padding and modes of operation. Padding schemes allow us to encrypt data less than the block size whereas modes of operation permit encryption of data larger than the block size.

Padding
A padding scheme takes a given an input with an insufficient number of bits to be used in a particular block cipher and pads the block with a padding string such that, the input block with padding can now be processed by the block cipher. The padding string is decided by the padding scheme and needs to be distinguishable at decryption time to separate the real block data and the padding. The first padding scheme proposed is to intuitively pad '0' or 0x00. This is perfectly acceptable for standard string encryption as ASCII conversion observes the padding as null. However this does not work for file encryption when dealing with binary data as we cannot distinguish the data from the padding.

The second proposal is to pad bytes up until the last byte where we put the padding length as the last byte and so one could distinguish the padding in the block by checking the last byte and trimming the amount of padding. This is acceptable in both ASCII and binary

Modes of operation
A mode of operation resolves the encryption and decryption of multiple blocks that introduces randomness with each block encrypted such that two blocks with the same message will consequently have different output blocks. A naive approach of encrypting multiple blocks containing the data of a single message would be to encrypt each block individually if they naturally cannot fit into a single block, using the same key. This is an insecure approach called the Electronic Codebook (ECB) mode, it provides no pseudo-randomness and as such there will be input blocks that have the same plain text and consequently the same output block, and additionally leave observable patterns in the output blocks. This can be clearly seen in the bitmap encryption using ECB mode below where in figure 18 where the images pattern/outline is preserved despite the noise.

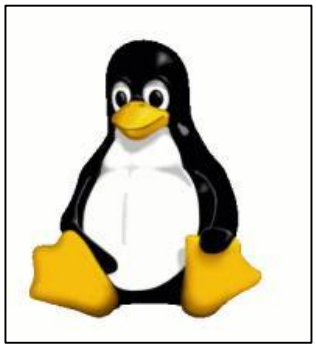

Figure 17: Original image (before)

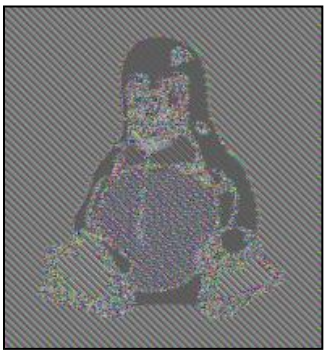

Figure 18: Encrypted image with ECB

The Cipher Block Chaining (CBC) mode developed by Smith, Tuchman, Meyer and Ehrsam in 1976 is well established and is the most widely used mode of operation that introduces randomness in each block and is a good candidate. The structure for CBC is shown in 19.

For encryption, the mode bitwise XOR's the previous output block with the current blocks plain text and then performs the encryption function on the block. In the case of block one, where no previous block exists, we use a 128bit randomized State (in our implementation) called an initial vector, IV. The IV should be randomly generated and disposed of (along with the private key used) when encryption of all blocks in the sequence is completed.

Decryption in CBC mode is done by taking the cipher block as input and performing the decryption function on the input where the output block is XOR'd with the previous cipher block (or initial vector for the first block), to produce the plain text.

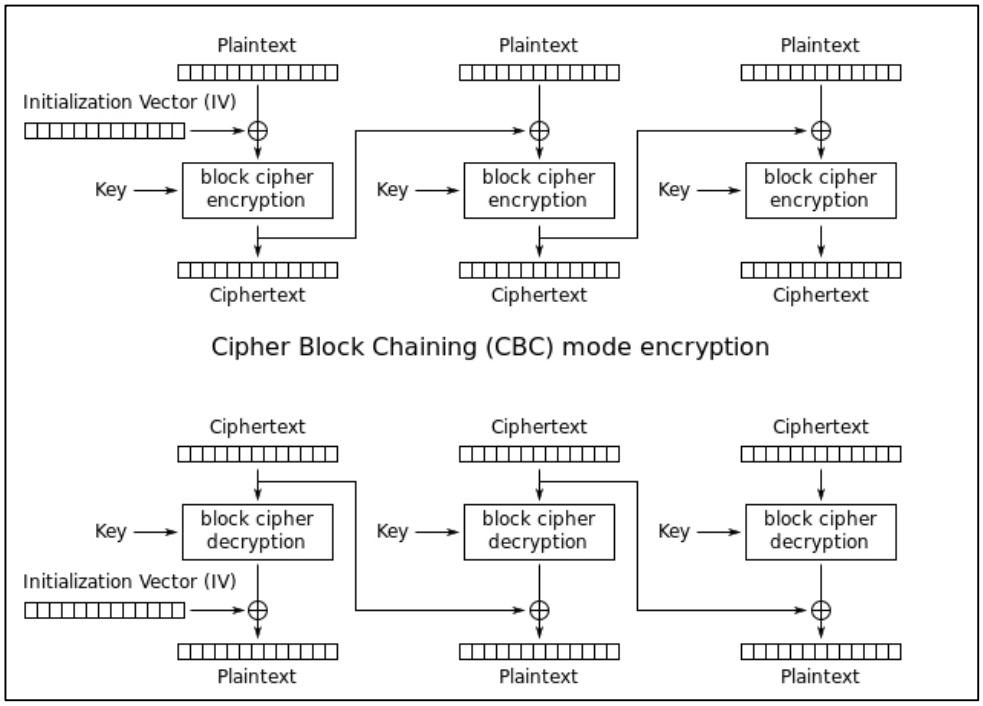

Figure 19: CBC mode of operation structure

AES Computational complexity analysis
AES encrypts and decrypts a single block of a fixed size where all procedures within the AES internal components perform constant operations and each component is performed at most for n_r rounds where the number of rounds n_r is relative to the key size and is at most 14. Therefore we can conclude that both the AES encryption and decryption functions run in constant $O(1)$ time.

With the introduction of modes of operation where there are n blocks, we perform the AES encrypt and decrypt functions n times with n XOR's with the previous block/IV's and can therefore observe linear $O(n)$ time when using AES with a mode of operation.

AES applied in modern web applications
Unfortunately, application of AES on its own is deemed impractical for modern web applications, regardless of being unbreakable and how complex you make or consider the algorithm. There are two very large holes in the inherent design of symmetric-key cryptography that without addressing ensure AES insecure for practical use. The problems we face are key-exchange and authentication.

The problem of key-exchange
A major flaw in the design of symmetric-key ciphers is the presumption of key establishment. The typical assumption is that both parties meet in private and exchange a private key that is later used. Although this is perhaps a desirable reality, it is impossible to assume such a scenario over web communication. Suppose the hypothetical that we use a web-based AES implementation without safe key exchange where users will be sending their private keys to a recipient over a potentially unsafe network where we assume an attacker is capable of intercepting the key during the transaction. Given that the attacker intercepts a private key, we have immediately lost and what we assume is encrypted data is evidently decrypted for the attacker to view. From this scenario created, hopefully one can identify the danger of directly sending private keys over an unsafe network. Therefore the problem can be seen as key-establishment.

We need a way to establish private keys without sending them and to resolve this it is necessary to explore asymmetric or public key algorithms. While ECC's and RSA public key algorithms offer a partial solution to the key-exchange problem, we are particularly interested in the Diffie-Hellman key exchange algorithm, proposed by Whitfield Diffie and Martin Hellman in 1976. Based on the discrete logarithm problem, to establish a private key between two users a, b using DH, we do the following:

(1) a, b each generate a random large integer 'secret'
(2) they agree on OR use a public prime p and generator g
(3) PublicKey(a) = $g^{secret(a)}\ mod\ p$ and PublicKey(b) = $g^{secret(b)}\ mod\ p$
(4) a: send b PublicKey(a), b: send a PublicKey(b)
(5) PrivateKey(a) = $PublicKey(b)^{secret(a)}\ mod\ p$
(6) PrivateKey(b) = $PublicKey(a)^{secret(b)}\ mod\ p$
(7) PrivateKey(a) = PrivateKey(b) and the private key has been established.

Finally to utilize this in our AES implementation we can perform an MD5 hash on the resulting private key which produces 128bits and can now be used as an AES private key.

The problem of authentication

Authentication is increasingly troublesome in web communication and the problem is knowing the origin of a message and whether the message recieved was the intended message sent where it can be captured by asking ourselves the questions who sent the message? And has this message been modified? It becomes particularly problematic when exploited and is essentially the method behind the Man In the Middle Attack, where an attacker sits inside a vulnerable network and intercepts messages, the messages may be encrypted however the attacker can modify the message such that the receiver sees an unintended, modified version of the message. Additionally the attacker may also spoof the identity of a messaging party.

The desirable solution would be to 'sign' a message before sending it so the receiver may know the origin and if the message state is the original. Fortunately such a solution exists and is known as a digital signature. The idea is to create a hash of the message where the hash identifies the initial state the message was in before being sent. The hash is then encrypted using an established private key to prevent attempts of spoofing the hash. The sender then sends the message along with the encrypted hash. On receiving the message, the receiver will decrypt the encrypted hash as well as the encrypted message with their established private key. The receiver then hashes the decrypted message and if the message hash and the decrypted hash match then the message has not been modified and is in the initial state it was sent in.

This provides very strong security for the message being sent as modifications made against any of the encrypted hash or encrypted message will result in a hash miss-match, additionally identities cannot be spoofed if the hash is encrypted as the attacker will require either of the private keys. Using the digital-signature solution accompanied by the DH key-exchange solution above, we can achieve complete client-side authentication and key-establishment with the AES implementation now prepared for deployment in modern web applications.

Conclusions

Throughout this report we explored the history, implementation and underlying design of AES. We delved into the process of encryption and decryption through their respective components. Additionally, the application of different padding schemes was investigated as well as modes of operation where we identified the need for pseudo-randomness when using a mode of operation. Lastly we looked at the role of AES in the web and the problems associated including key-exchange and authentication where we provided a client-side solution to both issues, allowing us to deploy AES in modern applications with confidence.

References

1. J. Daemen, V.Rijmen *The Design of Rijndael* 2002
2. S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani, *Algorithms,* pp. 41-42, 2006
3. J.Daemon, V.Rijmen, *AES Proposal: Rijndael* 1999
4. J.Pelzl, C.Paar, *Understanding Cryptography* pp 102-126, 2009
5. https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
6. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
7. http://www.di-mgt.com.au/cryptopad.html