Added Advanced Encryption Standard (A-Aes): With 512 Bits Data Block And 512, 768 And 1024 Bits Encryption Key

Mahendra Kumar Shrivas
Lecturer – Information Technology
Sikkim Manipal University, Kumasi, Ghana
mahendra@smughana.com

Satya Vir Singh
Campus Head
Sikkim Manipal University, Kumasi, Ghana
satya@smughana.com

ABSTRACT - Recent data security attacks have certainly played with the trust of the Computer and Internet users. They are panic to know about surveillance programs of some governments and security agencies. Secure systems are being compromised and encrypted communication channels are being intercepted by attacker and security agencies. Security systems need to be updated and algorithms need to be revised time to time. In November 26, 2001 National Institute of Standards and Technology (NIST) approved Advance Encryption Standard (AES) [1], which specifies a (Federal Information Processing Standards) FIPS approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. Researchers and attackers have done cryptanalysis and successfully recovered the secrete key after attack. 13 years old standard is still in use which is not be advisable to use. Some of the known attacks on AES are Biclique Cryptanalysis [2], Related-Key Cryptanalysis [3], and Improved Related-Key Impossible Differential Attacks [4], Cache-timing attacks on AES.[5], AES power attack[6], etc. In this research paper we are proposing Added Advance Encryption Standard (A-AES) algorithm which is capable of using cryptographic symmetric keys of 512, 768 and 1024 bits to encrypt and decrypt data in blocks of 512 bits.

KEYWORDS: AES, Encryption, Decryption, Symmetric Key, AES-512, AES-768, AES-1024, A-AES

I. INTRODUCTION

Encryption is the technique where the "plain text" i.e., the data to be secured is converted into "cipher text" which cannot be easily identified by unauthorized users. It is powerful tool in providing privacy, authenticity, integrity, and limited access to data. For the reason that networks often involve even greater risks, data is often secured with encryption, plausibly in combination with other controls.

The most important type of the encryption type is the symmetric key encryption. In the symmetric key encryption (Fig.1) both for the encryption and decryption process the same key is used. Hence the secrecy of the key is maintained and it is kept private.

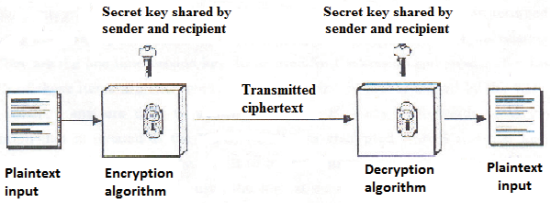

Fig 1 : Symmetric key cryptography

Symmetric algorithms have the advantage of not consuming too much of computing power and it works with high speed in encryption. A block cipher is taken as the input, a key and input, and then the output block will be same in size in the symmetric key encryption.

Though DES, Triple DES, AES and Blowfish are symmetric key cryptographic algorithm, and they have the ability to secure data.

AES is most widely and commonly used symmetric key encryption technique which is approved by National Institute of Standards and Technology (NIST) and specifies a (Federal Information Processing Standards) FIPS

approved cryptographic algorithm that can be used to protect electronic data.

A. ALGORITHM SPECIFICATION

For the AES algorithm, the length of the input block, the output block and the State is 128 bits. This is represented by Nb = 4, which reflects the number of 32-bit words (number of columns) in the State.

For the AES algorithm, the length of the Cipher Key, K, is 128, 192, or 256 bits. The key length is represented by Nk = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key.

For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr, where Nr = 10 when Nk = 4, Nr = 12 when Nk = 6, and Nr = 14 when Nk = 8.

The only Key-Block-Round combinations that conform to this standard are given below :-

	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Fig 2. Relation between key length, block size and number of rounds

B. ATTACHS ON AES

13 years old standard is still in use which is not be advisable to use. Some of the known attacks on AES are Biclique Cryptanalysis[2], Related-Key Cryptanalysis[3], and Improved Related-Key Impossible Differential Attacks [4],Cache-timing attacks on AES [5], AES power attack [6].

For AES-128, the key can be recovered with a computational complexity of $2^{126.1}$ using the biclique attack [2]. For biclique attacks on AES-192 and AES-256, the computational complexities of $2^{189.7}$ and $2^{254.4}$ respectively apply. Related-key attacks [3] can break AES-192 and AES-256 with complexities 2^{176} and $2^{99.5}$, respectively.

On July 1, 2009, Bruce Schneier blogged[7] about a related-key attack on the 192-bit and 256-bit versions of AES, discovered by Alex Biryukov and Dmitry Khovratovich,[8] which exploits AES's somewhat simple key schedule and has a complexity of 2^{119}. In December 2009 it was improved to $2^{99.5}$. This is a follow-up to an attack discovered earlier in 2009 by Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolić, with a complexity of 2^{96} for one out of every 2^{35} keys [9].

In November 2010 Endre Bangerter, David Gullasch and Stephan Krenn published a paper which described a practical approach to a "near real time" recovery of secret keys from AES-128 without the need for either cipher text or plaintext. The approach also works on AES-128 implementations that use compression tables, such as OpenSSL [10]. Like some earlier attacks this one requires the ability to run unprivileged code on the system performing the AES encryption, which may be achieved by malware infection far more easily than commandeering the root account[11].

II. PROPOSED ALGORITHM SPECIFICATION

For proposed ADDED ADVANCED ENCRYPTION STANDARD (A-AES) algorithm, the length of the input block, the output block and the State is 512 bits. This is represented by Nb = 8, which reflects the number of 64-bit words (number of columns) in the State.

For the A-AES algorithm, the length of the Cipher Key, K, is 512, 768, or 1024 bits. The key length is represented by Nk = 8, 12, or 16, which reflects the number of 64-bit words (number of columns) in the Cipher Key.
For the A-AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr, where Nr = 18 when Nk = 8, Nr = 22 when Nk = 12, and Nr = 26 when Nk = 16.

The only Key-Block-Round combinations that conform to this standard are given below :-

	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
A-AES-512	8	8	18
A-AES-768	12	8	22
A-AES-1024	16	8	26

Fig 3. Relation between key length, block size and number of rounds

A-AES algorithm uses a round function that is composed of four different byte-oriented transformations :-

- Byte substitution using a substitution table (S-box),
- Shifting rows of the State array by different offsets,
- Mixing the data within each column of the State array
- Adding a Round Key to the State.

A. CIPHER

At the start of the Cipher, the input is copied to the State array. After an initial Round Key addition, the State array is transformed by implementing a round function 18, 22, or 26 times (depending on the key length), with the final round differing slightly from the first ***Nr*** -1 rounds. The final State is then copied to the output.

The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the Key Expansion routine described in Sec. 3.2.

The Cipher is described in the pseudo code in Fig. 4. The individual transformations - **SubBytes()**, **ShiftRows()**, **MixColumns()**, and **AddRoundKey()** – process the State and are described in the following subsections. In Fig. 4, the array **w[]** contains the key schedule, which is described in Sec. 3.2.

As shown in Fig. 4, all ***Nr*** rounds are identical with the exception of the final round, which does not include the **MixColumns()**transformation.

```
Cipher(byte in[8*Nb], byte out[8*Nb], word w[Nb*(Nr+1)])
begin
byte    state[8,Nb]
        state = in
        AddRoundKey(state, w[0, Nb-1])
        for round = 1 step 1 to Nr–1 SubBytes(state)
            ShiftRows(state)
            MixColumns(state)
            AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
        end for
        SubBytes(state)
        ShiftRows(state)
        AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
        out = state
end
```

Fig 4. Pseudo Code for the Cipher.

- **SubBytes() Transformation:** The **SubBytes()** transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box (Fig. 5), which is invertible, is constructed by composing two transformations:

- Take the multiplicative inverse in the finite field GF(2^8)
- Apply the following affine transformation (over

$$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

GF(2^8)) (1)

for $0 \le i < 8$, where bi is the i^{th} bit of the byte, and ci is the i^{th} bit of a byte c with the value {63} or {01100011}. Here and elsewhere, a prime on a variable (e.g., b`) indicates that the variable is to be updated with the value on the right.
The S-box used in the **SubBytes()** transformation is presented in hexadecimal form in Fig. 5.

For example, if $s_{1,1}$ ={53}, then the substitution value would be determined by the intersection of the row with index ‘5’ and the column with index ‘3’ in Fig. 5. This would result in $s'_{1,1}$ having a value of {ed}.

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig 5. S-box: substitution values for the byte xy (in hexadecimal format)

- **ShiftRows() Transformation:** In the **ShiftRows()** transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted. Specifically, the ShiftRows() transformation proceeds as follows:

$$s' = s\,r,cr,(c+ shift\ (r, Nb)) \bmod Nb \text{ for } 0 < r < 8 \text{ and } 0 \le c < \boldsymbol{Nb}, \quad (2)$$

where the shift value *shift*(*r*,*Nb*) depends on the row number, *r*, as follows (recall that ***Nb*** = 8):

$$shift(1,8)=1; shift(2,8)=2; shift(3,8)=3; shift(4,8)=4; \quad shift(5,8)=5; shift(6,8)=6; shift(7,8)=7 . \quad (3)$$

This has the effect of moving bytes to "lower" positions in the row (i.e., lower values of *c* in a given row), while the "lowest" bytes wrap around into the "top" of the row (i.e., higher values of *c* in a given row).

- **MixColumns() Transformation :** The MixColumns() transformation operates on the State column-by-column, treating each column as a eight-term polynomial. The columns are considered as polynomials over GF(2^8) and multiplied modulo $x^8 + 1$ with a fixed polynomial a(x), given by

$$a(x) = \{07\}x^7 + \{06\}x^6 + \{05\}x^5 + \{04\}x^4 + \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \quad (4)$$

- **AddRoundKey() Transformation :** In the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of ***Nb*** words from the key schedule (described in Sec. 3.2). Those ***Nb*** words are each added into the columns of the State, such that

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}, s'_{4,c}, s'_{5,c}, s'_{6,c}, s'_{7,c}] = [s_{0,c}, s_{1,c}, s^{2,c}, s_{3,c}, s_{4,c}, s_{5,c}, s_{6,c}, s_{7,c}] \oplus [W_{round*Nb+c}] \quad \text{for } 0<=c<Nb \quad (5)$$

where [wi] are the key schedule and round is a value in the range $0 \le round \le Nr$. In the Cipher, the initial Round Key addition occurs when round = 0, prior to the first application of the round function (see Fig. 4). The application of the AddRoundKey() transformation to the Nr rounds of the Cipher occurs when $1 \le round \le Nr$.

B. KEY EXPANSION

The A-AES algorithm takes the Cipher Key, ***K***, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of ***Nb*** (***Nr*** + 1) words: the algorithm requires an initial set of ***Nb*** words, and each of the ***Nr*** rounds requires ***Nb*** words of key data. The resulting key schedule consists of a linear array of 8-byte words, denoted [*wi*], with *i* in the range $0 \le i < \boldsymbol{Nb}(\boldsymbol{Nr} + 1)$.

The expansion of the input key into the key schedule proceeds according to the pseudo code in Fig. 6.

SubWord() is a function that takes a eight-byte input word and applies the S-box (Fig. 5) to each of the eight bytes to produce an output word.

The function RotWord() takes a word [a0,a1,a2,a3,a4,a5,a6,a7] as input, performs a cyclic permutation, and returns the word [a1,a2,a3,a4,a5,a6,a7,a0]. The round constant word array, Rcon[i], contains the values given by [xi-1,{00},{00},{00}], with x i-1 being powers of x (x is denoted as {02}) in the field GF(28), It is important to note that the Key Expansion routine for 1024-bit Cipher Keys (Nk = 16) is slightly different than for 512- and 768-bit Cipher Keys. If Nk = 16 and i-8 is a multiple of Nk, then SubWord()is applied to w[i-1]prior to the XOR.

```
KeyExpansion(byte key[8*Nk], word w[Nb*(Nr+1)],
Nk)
begin
word temp
i = 0
while (i < Nk)
        w[i] = word(key[8*i], key[8*i+1],
        key[8*i+2], key[8*i+3] ], key[8*i+4] ],
        key[8*i+5] ], key[8*i+6] ], key[8*i+7])
        i = i+1
end while
i = Nk
while (i < Nb * (Nr+1))
         temp = w[i-1]
        if (i mod Nk = 0)
                temp = SubWord(RotWord(temp))
                xor Rcon[i/Nk]
        else if (Nk > 12 and i mod Nk = 8)
                temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
   end while
end
```

Fig. 6. Pseudo Code for Key Expansion.

C. INVERSE CIPHER

The Cipher transformations in Sec. 3.1 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the A-AES algorithm. The individual transformations used in the Inverse Cipher - InvShiftRows(), InvSubBytes(),InvMixColumns(), and AddRoundKey() – process the State and are described in the following subsections. The Inverse Cipher is described in the pseudo code in Fig. 7. In Fig. 7, the array w[] contains the key schedule, which was described previously in Sec. 3.2.

```
InvCipher(byte in[8*Nb], byte out[8*Nb], word
w[Nb*(Nr+1)])
begin
byte state[8,Nb]
        state = in
        AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-
        1])
        for round = Nr-1 step -1 downto 1
                InvShiftRows(state)
                InvSubBytes(state)
                AddRoundKey(state, w[round*Nb,
                (round+1)*Nb-1])
                InvMixColumns(state)
        end for
        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w[0, Nb-1])
out = state
end
```

Fig. 7 Pseudo Code for the Inverse Cipher

- **InvShiftRows() Transformation:** InvShiftRows() is the inverse of the ShiftRows() transformation. The bytes in the last serven rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted. The bottom three rows are cyclically shifted by Nb -*shift*(r, Nb) bytes, where the shift value *shift(r,Nb)* depends on the row number, and is given in equation (3) (see Sec. 3.1.2).

Specifically, the InvShiftRows() transformation proceeds as follows:

$$s\ r,(c+\ shift\ (r,\ Nb))\ \mathrm{mod}\ Nb\ = s\ r,c \text{ for } 0 < r < 4 \text{ and } 0 \le c < \boldsymbol{Nb} \qquad (6)$$

- **InvSubBytes() Transformation :** InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (3.1) followed by taking the multiplicative inverse in GF(2^8).

The inverse S-box used in the InvSubBytes() transformation is presented in Fig. 8:

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Fig. 8 Inverse S-box: substitution values for the byte xy (in hexadecimal format).

- **InvMixColumns() Transformation:** InvMixColumns() is the inverse of the MixColumns() transformation. InvMixColumns() operates on the State column-by-column, treating each column as a eight-term polynomial. The columns are considered as polynomials over GF(2^8) and multiplied modulo x^8 + 1 with a fixed polynomial.

- **Inverse of the AddRoundKey() Transformation:** AddRoundKey(), which was described in Sec. 3.1.4, is its own inverse, since it only involves an application of the XOR operation.

- **Equivalent Inverse Cipher:** In the straightforward Inverse Cipher presented in Sec. 3.3 and Fig.7, the sequence of the transformations differs from that of the Cipher, while the form of the key schedules for encryption and decryption remains the same. However, several properties of the AES algorithm allow for an Equivalent Inverse Cipher that has the same sequence of transformations as the Cipher (with the transformations replaced by their inverses). This is accomplished with a change in the key schedule.

The two properties that allow for this Equivalent Inverse Cipher are as follows:

- The SubBytes() and ShiftRows() transformations commute; that is, a SubBytes() transformation immediately followed by a ShiftRows() transformation is equivalent to a ShiftRows() transformation immediately followed by a SubBytes() transformation. The same is true for their inverses, InvSubBytes() and InvShiftRows.

- The column mixing operation - MixColumns() and InvMixColumns() - are linear with respect to the column input, which means InvMixColumns(state XOR Round Key) = InvMixColumns(state) XOR InvMixColumns(Round Key).

These properties allow the order of InvSubBytes() and InvShiftRows() transformations to be reversed. The order of the AddRoundKey() and InvMixColumns() transformations can also be reversed, provided that the columns (words) of the decryption key schedule are modified using the InvMixColumns()transformation.

The equivalent inverse cipher is defined by reversing the order of the InvSubBytes() and InvShiftRows() transformations shown in Fig. 7, and by reversing the order of the AddRoundKey() and InvMixColumns() transformations used in the "round loop" after first modifying the decryption key schedule for round = 1 to Nr-1 using the InvMixColumns() transformation. The first and last Nb words of the decryption key schedule shall *not* be modified in this manner. Given these changes, the resulting Equivalent Inverse Cipher offers a more efficient structure than the Inverse Cipher described in Sec. 3.3 and Fig. 7. Pseudo code for the Equivalent Inverse Cipher appears in Fig. 9. (The word array **dw[]** contains the modified decryption key schedule. The modification to the Key Expansion routine is also provided in Fig. 9.)

```
EqInvCipher(byte in[8*Nb], byte out[8*Nb], word dw[Nb*(Nr+1)])
begin
byte state[8,Nb]
state = in
AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])
for round = Nr-1 step -1 downto 1
        InvSubBytes(state)
        InvShiftRows(state)
        InvMixColumns(state)
AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
end for
InvSubBytes(state)
InvShiftRows(state)
AddRoundKey(state, dw[0, Nb-1])
out = state
end
For the Equivalent Inverse Cipher, the following pseudo code is added at the end of the Key Expansion routine (Sec. 3.2):
for i = 0 step 1 to (Nr+1)*Nb-1
        dw[i] = w[i]
end for
for round = 1 step 1 to Nr-1
        InvMixColumns(dw[round*Nb, (round+1)*Nb-1]) // note change of type
end for
```

Fig. 9 Pseudo Code for the Equivalent Inverse Cipher

III. CONCLUSION

AES is being used in various Archive and compression tools(7z, RAR, WinZip, UltraISO), Encrypting File System

in Windows, Disk encryption tools (DiskCryptor, BitLocker, TrueCrypt, Private Disk),Security for communications in Local Area Networks(IEEE 802.11i, IEEE 802.11), IPsec, OpenSSL, CyaSSL, Intel and AMD processors include the AES instruction set. On IBM zSeries mainframes, AES is implemented as the KM series of assembler opcodes when various Message Security Assist facilities are installed.

As mentioned AES is not 100% secure and there is a need of more secure standard. This proposal may fulfill the need as A-AES uses 512 bit data block with 512 bit, 768 bit or 1024 bit key for encryption and decryption and can be easily implemented using any programming language for any platform.

IV. FUTURE WORKS

A-AES need to be implemented by programmers and need to be tested against well known attacks.

V. ACKNOWLEDGMENTS

Proposed algorithm is enhancement of standard AES. Authors do not hold any rights on original standards version of AES. Software mentioned in this work are just for reference purpose and intellectual properties of respected organization(s).

VI. REFERENCES

[1] US National Institute of Standards and Technology Advanced Encryption Standard, Federal Information Processing Standards Publications No. 197, 2001.

[2] Andrey Bogdanov, Dmitry Khovratovich and Christian Rechberge Biclique Cryptanalysis of the Full AES 16 Aug 2011

[3] Alex Biryukov and Dmitry Khovratovich, Related-Key Cryptanalysis of the Full AES-192 and AES-256, Advances in Cryptography, proceedings of ASIACRYPT2009, Lecture Notes in Computer Science 5912, pp. 1–18, Springer, 2009.

[4] Key Impossible Differential Attacks on Reduced-Round AES-192, Proceedings of Selected Areas in Cryptography 2006, Lecture Notes in Computer Science 4356,pp. 15–27, Springer, 2007.

[5] Daniel J. Bernstein. Cache-timing attacks on AES.April2005.http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[6] Guido Bertoni, Luca Breveglieri, Matteo Monchiero, Gianluca Palermo, and Vittorio Zaccaria, AES power attack based on induced cache miss and countermeasure. ITCC (1), 2005.

[7] Bruce Schneier (2009-07-01). "New Attack on AES". Schneier on Security, A blog covering security and security technology. Archived from the original on 8 February 2010. Retrieved 2010-03-11.

[8] Biryukov, Alex; Khovratovich, Dmitry (2009-12-04). "Related-key Cryptanalysis of the Full AES-192 and AES-256". Retrieved 2010-03-11.

[9] Nikolić, Ivica (2009). "Distinguisher and Related-Key Attack on the Full AES-256". Advances in Cryptology – CRYPTO 2009. Springer Berlin / Heidelberg. pp. 231–249. doi:10.1007/978-3-642-03356-8_14. ISBN 978-3-642-03355-1.

[10] Endre Bangerter, David Gullasch and Stephan Krenn (2010). "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice".

[11] "Breaking AES-128 in realtime, no ciphertext required | Hacker News". News.ycombinator.com. Retrieved 2012-12-23.