Formal Verification of AES Using the Mizar Proof Checker

Hiroyuki Okazaki[1], **Kenichi Arai**[2], **and Yasunari Shidama**[1]
[1]Shinshu University, 4-17-1 Wakasato Nagano-city, Nagano 380-8553, Japan
[2]Tokyo University of Science, 2641 Yamazaki Noda-City, Chiba 278-8510, Japan

Abstract— *In this paper, we introduce our formalization of the Advanced Encryption Standard (AES) algorithm. AES, which is the most widely used symmetric cryptosystem in the world, is a block cipher that was selected by the National Institute of Standards and Technology (NIST) as an official Federal Information Processing Standard for the United States in 2001. We prove the correctness of our formalization using the Mizar proof checking system as a formal verification tool. Mizar is a project that formalizes mathematics with a computer-aided proving technique and is a universally accepted proof checking system. The main objective of this work is to prove the security of cryptographic systems using the Mizar proof checker.*

Keywords: Formal Verification, Mizar, Cryptology, Advanced Encryption Standard (AES)

1. Introduction

Mizar[1] is a project that formalizes mathematics with a computer-aided proving technique. The objective of this study is to prove the security of cryptographic systems using the Mizar proof checker. To achieve this study, we intend to formalize several topics concerning cryptology. As a part of this effort, we introduced our formalization of the Data Encryption Standard (DES)[2] at the FCS'11[3].

In this paper, we introduce our formalization of the Advanced Encryption Standard (AES). AES, which is the most widely used symmetric cryptosystem in the world, is a block cipher that was selected by the National Institute of Standards and Technology (NIST) as an official Federal Information Processing Standard for the United States in 2001[4]. AES is the successor to DES, which was formerly the most widely used symmetric cryptosystem in the world. However, DES is now considered insecure and has been replaced by AES[5]. We formalized the AES algorithm as shown in FIPS 197[4] in the Mizar language. We then verified the correctness of the formalized algorithm that the ciphertext encoded by the AES algorithm can be decoded uniquely by the same key using the Mizar proof checker.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the Mizar project. In Section 3, we briefly introduce the Advanced Encryption Standard (AES). In Section 4, we discuss our strategy for formalizing AES in Mizar. In Sections 5 and 6, we propose a formalization of AES. We conclude our discussion in Section 7. The definitions and theorems in this paper have been verified for correctness using the Mizar proof checker.

2. Mizar

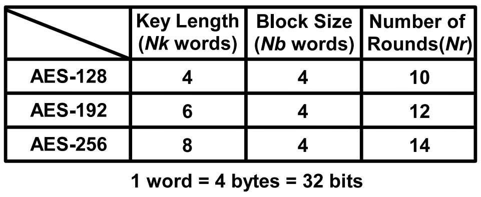
Mizar[1] is an advanced project of the Mizar Society led by A.Trybulec which formalizes mathematics with a computer-aided proving technique. The Mizar project describes mathematical proofs in the Mizar language, which is created to formally describe mathematics. The Mizar proof checker operates in both Windows and UNIX environments, and registers the proven definitions and theorems in the Mizar Mathematical Library (MML).

What formalizes the proof of mathematics by Mizar and describes it is called "article". When an article is newly described, it is possible to advance it by referring to articles registered in the MML that have already been inspected as proof. Although the Mizar language is based on the description method for general mathematical proofs, the reader should consult the references for its grammatical details, because Mizar uses a specific, unique notation[6], [7], [8], [9].

3. Advanced Encryption Standard

In this section, we review the outline of the AES algorithm, which is a variant of Rijndael algorithm[10]. The AES algorithm can process 128–bit data blocks using secret keys of lengths 128, 192, or 256 bits. Decryption must be performed using the same key as that used for encryption. However, the decryption algorithm is different from the encryption algorithm. Depending on the key lengths, AES is referred to as AES–128, AES–192, or AES–256.

AES is a type of iterated block cipher that has a Substitution Permutation Network (SPN) structure. The SPN structure alternately performs substitution and permutation operations. The encryption and decryption of the SPN structure involve different processes. The AES algorithm is composed of the SPN structure and a key scheduling. In the SPN structure of AES, there are 10, 12, and 14 rounds of processing iterations. The number of rounds to be performed during the execution of the AES algorithm is dependent on the key lengths. In AES algorithm, the key length, block size, and number of rounds are represented by Nk, Nb, and Nr, respectively. The Nk–Nb–Nr combinations are shown in Figure 1.

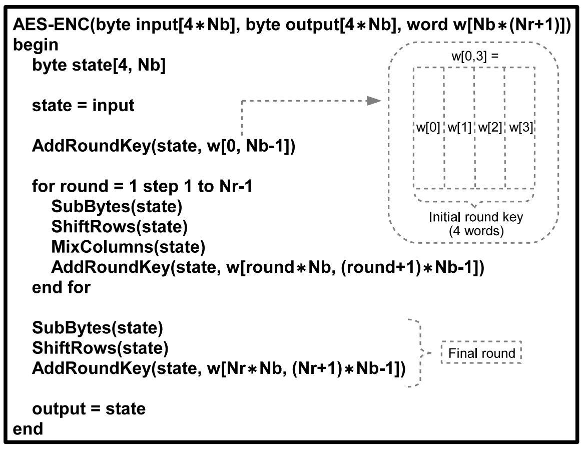


	Key Length (*Nk* words)	Block Size (*Nb* words)	Number of Rounds(*Nr*)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

1 word = 4 bytes = 32 bits

Figure 1: Nk–Nb–Nr combinations

The AES algorithm is performed on a two-dimensional array of bytes called the "State". Before the main iterations, the plaintext block is copied into the State array. After an initial round key addition, the State array is transformed by performing a round process Nr times. However, only the final round is different. Finally, the final State is copied to the ciphertext block. The round process is composed of the "SubBytes", "ShiftRows", "MixColumns", and "AddRoundKey" transformations. The final round does not include the MixColumns transformation. The round key is yielded by the key scheduling from the given secret key and is added to the State array using the AddRoundKey transformation. Figure 2 shows the pseudo code for the encryption algorithm of AES.



```
AES-ENC(byte input[4∗Nb], byte output[4∗Nb], word w[Nb∗(Nr+1)])
begin
  byte state[4, Nb]

  state = input

  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])

  output = state
end
```

w[0,3] =
w[0]
w[1]
w[2]
w[3]
Initial round key (4 words)
Final round

Figure 2: Pseudo code for the encryption algorithm

In decryption, the round process is composed of the "InvSubBytes", "InvShiftRows", "InvMixColumns", and AddRoundKey transformations. The InvSubBytes, InvShiftRows, and InvMixColumns are the inverse of the SubBytes, ShiftRows, and MixColumns transformations, respectively. In decryption, each transformation is performed in the reverse order of encryption and the round keys are used in the reverse order of encryption.

4. Strategy for Formalizing AES

In Mizar, there are two ways to define computational routines in an algorithmic sense. One way is by defining a routine as a "Function". A Function is a map from the space of the input onto that of the output. We can handle a Function as an element of the set of Functions.

The other way is by defining a routine as a "functor". A functor is a relation between the input and output of a routine in Mizar. It is easy to write and understand the formalization of a routine as a functor, because the format of a functor in Mizar is similar to that of a function in certain programming languages. Note that both functor and Function can take a Function as their substitutable subroutines.

In Section 5, we will formalize the subroutines, that is, the primitives of AES, according to FIPS 197[4]. In Section 6, we first formalize the algorithm of generalized AES as a functor that takes substitutional subroutines. This generalized definition of AES is easily reusable for the formalization of different key lengths of AES. We will then formalize the AES algorithm using the formalization of the primitives in Section 5 and the generalized definition in Section 6.1.

5. Formalization of AES Primitives

In this section, we formalize the AES primitives according to FIPS 197[4].

5.1 State array

Figure 3 shows a sketch of the State array.

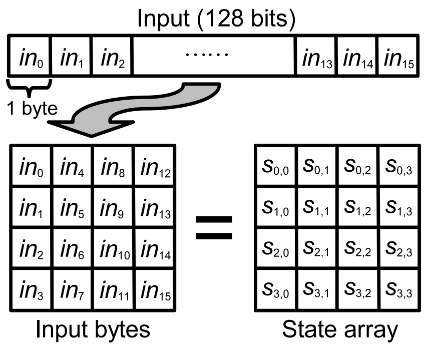

Input (128 bits)
in_0 in_1 in_2 ...... in_{13} in_{14} in_{15}
1 byte
in_0 in_4 in_8 in_{12}
in_1 in_5 in_9 in_{13}
in_2 in_6 in_{10} in_{14}
in_3 in_7 in_{11} in_{15}
=
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
Input bytes
State array

Figure 3: State array

The State array consists of 4 rows of bytes, each of which contains Nb bytes.

We formalize the State array as the following functor in the Mizar language:

Definition 5.1: (State array)

```
func AES-Statearray ->
  Function of 128-tuples_on BOOLEAN,
  4-tuples_on (4-tuples_on (8-tuples_on
  BOOLEAN))
means
for input be Element of 128-tuples_on
  BOOLEAN
for i,j be Nat st i in Seg 4 & j in Seg 4
holds
((it.input).i).j = mid(input,1+(i-'1)*8+
  (j-'1)*32,1+(i-'1)*8+(j-'1)*32+7);
```

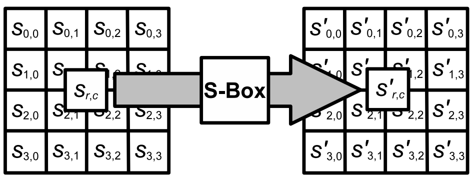
□

Here, mid is a function that extracts a subsequence (finite sequence) and * is multiplication. For example, mid(input,9,16) is a finite sequence (input.9,......,input.16) of length 8. Note that the index of the finite sequence starts from 1 in the Mizar language.

We similarly defined the functor of the inverse of the State array as AES-Statearray".

5.2 SubBytes

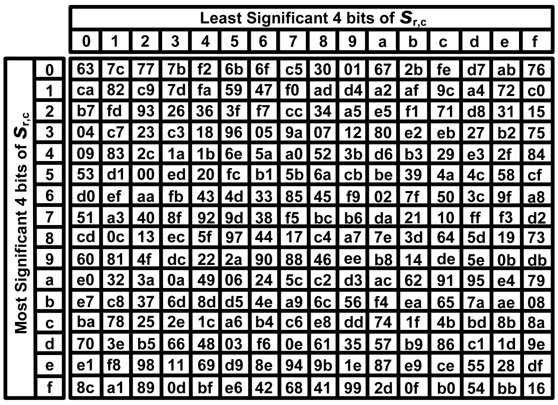
Figure 4 shows a sketch of the SubBytes transformation.



$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,3}$
$S_{r,c}$
S-Box
$S'_{r,c}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S'_{0,0}$ $S'_{0,1}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,3}$
$S'_{2,0}$ $S'_{2,1}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$ $S'_{3,1}$ $S'_{3,2}$ $S'_{3,3}$

Figure 4: SubBytes

The SubBytes transformation is a nonlinear byte substitution that independently operates on each byte of the State array using a 1-byte substitution table. The substitution table is called the “S-Box”. The S-Box, which is invertible, is constructed by composing two transformations. Two transformations are composed of the calculation of the multiplicative inverse in the finite field GF(2^8) and the affine transformation over GF(2). Figure 5 shows a sketch of the S-Box.



Most Significant 4 bits of $S_{r,c}$	Least Significant 4 bits of $S_{r,c}$															
	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

Figure 5: S-Box (in hexadecimal form)

We formalize the SubBytes transformation as the following functor in the Mizar language:

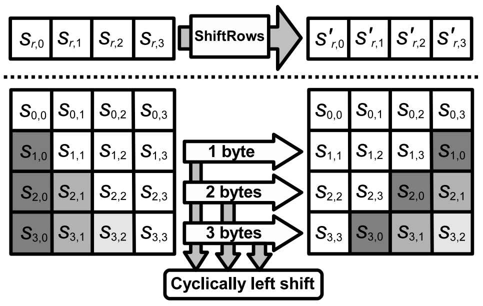
Definition 5.2: (SubBytes)
```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
func SubBytes(SBT) ->
  Function of 4-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN)),4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
means
for input be Element of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
holds
(for i,j be Nat st i in Seg 4 & j in Seg 4
holds
ex inputij be Element of 8-tuples_on
  BOOLEAN
st inputij = (input.i).j &
((it.input).i).j = SBT.(inputij));
```
□

Please note the following points about this formalization. This functor can specify an arbitrary Permutation of 8-tuples_on BOOLEAN because it takes SBT as an argument. In this formalization, so as not to lose generality, we describe this functor as the SubBytes transformation. The actual SubBytes transformation uses the S-Box, as shown in Figure 5. However, the formal description of this S-Box is not significant. Therefore, we described this functor as the SubBytes transformation.

We similarly defined the functor of the InvSubBytes transformation (Definition A. 1).

5.3 ShiftRows

Figure 6 shows a sketch of the ShiftRows transformation.



$S_{r,0}$ $S_{r,1}$ $S_{r,2}$ $S_{r,3}$
ShiftRows
$S'_{r,0}$ $S'_{r,1}$ $S'_{r,2}$ $S'_{r,3}$
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
1 byte
2 bytes
3 bytes
Cyclically left shift
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,0}$
$S_{2,2}$ $S_{2,3}$ $S_{2,0}$ $S_{2,1}$
$S_{3,3}$ $S_{3,0}$ $S_{3,1}$ $S_{3,2}$

Figure 6: ShiftRows

The ShiftRows transformation operates the State array by cyclically shifting the last three rows of the State array by different offsets (numbers of bytes). Note that the first row is not shifted.

We formalize the ShiftRows transformation as the following functor in the Mizar language:

Definition 5.3: (ShiftRows)
```
func ShiftRows ->
  Function of 4-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN)),4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
means
for input be Element of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
holds
(for i be Nat st i in Seg 4
holds
ex xi be Element of 4-tuples_on
  (8-tuples_on BOOLEAN)
st xi = input.i &
(it.input).i = Op-Shift(xi,5-i));
```
□

Here, Op-Shift is a cyclically left shift function.

We similarly defined the functor of the InvShiftRows transformation (Definition A. 2).

5.4 MixColumns

Figure 7 shows a sketch of the MixColumns transformation.

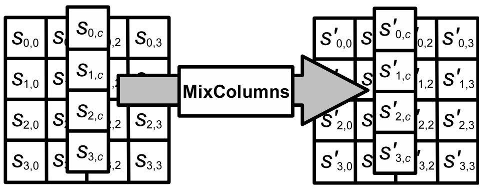


Figure 7: MixColumns

In the MixColumns transformation, the 4 bytes of each column of the State array are mixed using an invertible linear transformation. The columns are considered as polynomials over GF(2^8) and multiplied modulo x^4+1 with a fixed polynomial $a(x)$, given by $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$. As a result of the above mentioned multiplication, the 4 bytes in a column are replaced by the following:

$$\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}, \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}, \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}), \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).
\end{aligned}$$

Note that {01}, {02}, and {03} are hexadecimal notations and • is a multiplication in GF(2^8).

We formalize the MixColumns transformation as the following functor in the Mizar language:

Definition 5.4: (MixColumns)
```
func MixColumns ->
  Function of 4-tuples_on(4-tuples_on
  (8-tuples_on BOOLEAN)),4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
means
for input be Element of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
holds
ex x,y being Element of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
st x = input & y = it.input &
for i be Element of NAT st i in Seg 4
holds
ex x1,x2,x3,x4 be Element of
  8-tuples_on BOOLEAN
st x1 = (x.i).1 & x2 = (x.i).2 &
x3 = (x.i).3 & x4 = (x.i).4 &
(y.1).i = Op-XOR(Op-XOR(Op-XOR(2 'gf' x1,
  3 'gf' x2),1 'gf' x3),1 'gf' x4) &
(y.2).i = Op-XOR(Op-XOR(Op-XOR(1 'gf' x1,
  2 'gf' x2),3 'gf' x3),1 'gf' x4) &
(y.3).i = Op-XOR(Op-XOR(Op-XOR(1 'gf' x1,
  1 'gf' x2),2 'gf' x3),3 'gf' x4) &
(y.4).i = Op-XOR(Op-XOR(Op-XOR(3 'gf' x1,
  1 'gf' x2),1 'gf' x3),2 'gf' x4);
```
□

Here, Op-XOR is a bitwise XOR (exclusive OR) function and 'gf' is a multiplication in GF(2^8).

We similarly defined the functor of the InvMixColumns transformation (Definition A. 3).

5.5 AddRoundKey

Figure 8 shows a sketch of the AddRoundKey transformation. Here, w_{l+c} are the key scheduling words and $round$

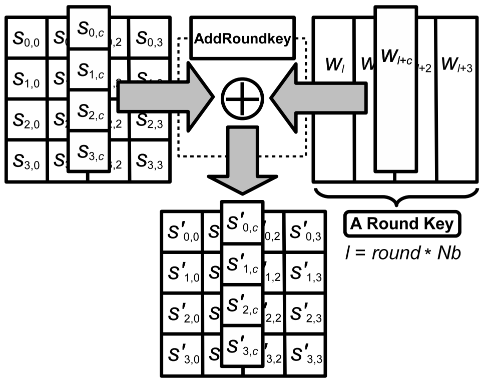


Figure 8: AddRoundKey

is a value in the range $0 \leq round \leq Nr$.

The AddRoundKey transformation adds a Round Key and the State array using a bitwise XOR.

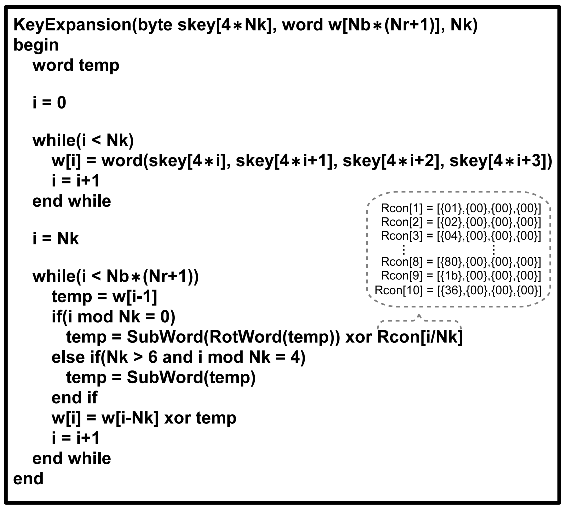
We formalize the AddRoundKey transformation as the following functor in the Mizar language:

Definition 5.5: (AddRoundKey)
```
func AddRoundKey ->
  Function of [:4-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN)),4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN)):],
  4-tuples_on (4-tuples_on (8-tuples_on
  BOOLEAN))
means
for text,key be Element of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
holds
for i,j be Nat st i in Seg 4 & j in Seg 4
holds
ex textij,keyij be Element of
  8-tuples_on BOOLEAN
st textij = (text.i).j & keyij = (key.i).j
& ((it.(text,key)).i).j =
  Op-XOR(textij,keyij);
```
□

5.6 Key Expansion

The AES algorithm takes the secret key and performs a Key Expansion process to generate the key scheduling. The resulting key scheduling consists of a linear array of 4-byte words, denoted as w_i, with i being in the range $0 \leq i \leq$

$Nb(Nr+1)$. Figure 9 shows the pseudo code for the Key Expansion.



```
KeyExpansion(byte skey[4∗Nk], word w[Nb∗(Nr+1)], Nk)
begin
  word temp

  i = 0

  while(i < Nk)
    w[i] = word(skey[4∗i], skey[4∗i+1], skey[4∗i+2], skey[4∗i+3])
    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 > 6 and i mod Nk = 4)
      temp = SubWord(temp)
    end if
    w[i] = w[i-Nk] xor temp
    i = i+1
  end while
end
```

```
 Rcon[1] = [{01},{00},{00},{00}]
 Rcon[2] = [{02},{00},{00},{00}]
 Rcon[3] = [{04},{00},{00},{00}]
      ⋮                 ⋮
 Rcon[8] = [{80},{00},{00},{00}]
 Rcon[9] = [{1b},{00},{00},{00}]
Rcon[10] = [{36},{00},{00},{00}]
```

Figure 9: Pseudo code for the Key Expansion

SubWord is a function that takes an input word of 4 bytes and applies the S-Box to each of the 4 bytes to produce an output word.

We formalize SubWord as the following functor in the Mizar language:

Definition 5.6: (SubWord)

```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
let x be Element of 4-tuples_on
  (8-tuples_on BOOLEAN);
func SubWord(SBT,x) ->
  Element of 4-tuples_on (8-tuples_on
  BOOLEAN)
means
for i be Element of Seg 4
holds
it.i = SBT.(x.i);
```

□

RotWord is a function that takes a word $[b_0, b_1, b_2, b_3]$ as the input, performs a cyclically left shift, and returns the word $[b_1, b_2, b_3, b_0]$.

We formalize RotWord as the following functor:

Definition 5.7: (RotWord)

```
let x be Element of 4-tuples_on
  (8-tuples_on BOOLEAN);
func RotWord(x) ->
  Element of 4-tuples_on (8-tuples_on
  BOOLEAN)
equals
Op-LeftShift(x);
```

□

Here, Op-LeftShift is a cyclically 1 byte left shift function.

The round constant word array, Rcon, is a constant that is different for each round. Rcon[i] is defined as $\left[x^{i-1}, \{00\}, \{00\}, \{00\}\right]$. Here, x^{i-1} are powers of $x(=\{02\})$ in the field GF(2^8). Note that i starts from 1.

We formalize Rcon as the following functor:

Definition 5.8: (Rcon)

```
func Rcon ->
  Element of 10-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN))
means
it.1 = <* <*0,0,0,0*>^<*0,0,0,1*>,
  <*0,0,0,0*>^<*0,0,0,0*>,
  <*0,0,0,0*>^<*0,0,0,0*>,
  <*0,0,0,0*>^<*0,0,0,0*> *> &
                  :
            (omitted)
                  :
it.10 = <* <*0,0,1,1*>^<*0,1,1,0*>,
  <*0,0,0,0*>^<*0,0,0,0*>,
  <*0,0,0,0*>^<*0,0,0,0*>,
  <*0,0,0,0*>^<*0,0,0,0*> *>;
```

□

Here, ^ is concatenation. For example, $\langle *0,0,1,1* \rangle \hat{} \langle *0,1,1,0* \rangle = \{36\}$.

Next, to formalize the Key Expansion, we formalize the KeyExTemp and KeyExMain as the following functors.

Definition 5.9: (KeyExTemp)

```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
let m,i be Nat,
w be Element of (4-tuples_on (8-tuples_on
  BOOLEAN));
assume (m = 4 or m = 6 or m = 8) &
i < 4*(7+m) & m <= i;
func KeyExTemp(SBT,m,i,w) ->
  Element of (4-tuples_on (8-tuples_on
  BOOLEAN))
means
(ex T3 be Element of (4-tuples_on
  (8-tuples_on BOOLEAN))
st T3 = Rcon.(i/m) &
it = Op-WXOR(SubWord(SBT,RotWord(w)),T3))
if ((i mod m) = 0),(it = SubWord(SBT,w))
if (m = 8 & (i mod 8) = 4) otherwise
  it = w;
```

□

Definition 5.10: (KeyExMain)

```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
let m be Nat;
assume m = 4 or m = 6 or m = 8;
func KeyExMain(SBT,m) ->
  Function of m-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN)),(4*(7+m))-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
means
for Key be Element of m-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
holds
(for i be Element of NAT st i < m
  holds (it.Key).(i+1) = Key.(i+1)) &
(for i be Element of NAT st m <= i &
  i < 4*(7+m)
holds ex P be Element of (4-tuples_on
  (8-tuples_on BOOLEAN)),Q be Element of
  4-tuples_on (8-tuples_on BOOLEAN)
```

```
    st P = (it.Key).((i-m)+1) &
    Q = (it.Key).i & (it.Key).(i+1) =
      Op-WXOR(P,KeyExTemp(SBT,m,i,Q)));
```

□

Finally, we formalize the Key Expansion as the following functor.

Definition 5.11: (Key Expansion)

```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
let m be Nat;
assume m = 4 or m = 6 or m = 8;
func KeyExpansion(SBT,m) ->
  Function of m-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN)),(7+m)-tuples_on
  (4-tuples_on (4-tuples_on (8-tuples_on
  BOOLEAN)))
means
for Key be Element of m-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
holds
ex w be Element of (4*(7+m))-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN))
st w = (KeyExMain(SBT,m)).Key &
for i be Nat st i < 7+m
holds
(it.Key).(i+1) = <*w.(4*i+1),w.(4*i+2),
  w.(4*i+3),w.(4*i+4)*>;
```

□

6. Formalization of AES

In this section, we formalize the AES algorithm according to FIPS 197[4] in the Mizar language. First, we formalize and prove the correctness of the generalized AES algorithm. Next, we formalize and prove the correctness of the AES algorithm.

6.1 Formalization of Generalized AES

The generalized AES algorithm is easily reusable for the formalization of different key lengths of AES.

We formalize the encryption algorithm of generalized AES as a functor in the Mizar language as follows:

Definition 6.1: (Generalized AES encryption algorithm)

```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
let MCFunc be Permutation of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN));
let m be Nat;
let text be Element of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN));
let Key be Element of m-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN));
assume 4 <= m;
func AES-ENC(SBT,MCFunc,text,Key) ->
  Element of 4-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN))
means
ex seq be FinSequence of (4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN)))
st len seq = 7+m-1 &
(ex Keyi1 be Element of 4-tuples_on
    (4-tuples_on (8-tuples_on BOOLEAN))
    st Keyi1 = ((KeyExpansion(SBT,m)).(Key)).1
    & seq.1 = AddRoundKey.(text,Keyi1)) &
  (for i be Nat st 1 <= i & i < 7+m-1
    holds
    ex Keyi be Element of 4-tuples_on
      (4-tuples_on (8-tuples_on BOOLEAN))
    st Keyi = ((KeyExpansion(SBT,m)).(Key)).
      (i+1) & seq.(i+1) = AddRoundKey.
      ((MCFunc*ShiftRows*SubBytes(SBT)).
      (seq.i),Keyi)) &
  ex KeyNr be Element of 4-tuples_on
    (4-tuples_on (8-tuples_on BOOLEAN))
  st KeyNr = ((KeyExpansion(SBT,m)).(Key)).
    (7+m) & it = AddRoundKey.((ShiftRows*
    SubBytes(SBT)).(seq.(7+m-1)),KeyNr);
```

□

Here, * is composition. Note that the composition of the function is described in the reverse order of the actual processing.

We similarly defined the functor of the decryption algorithm of generalized AES as AES-DEC (Definition A. 4).

We then prove the following theorem:

Theorem 6.1: (Correctness of generalized AES)

```
for SBT be Permutation of (8-tuples_on
  BOOLEAN),
MCFunc be Permutation of 4-tuples_on
  (4-tuples_on (8-tuples_on BOOLEAN)),
m be Nat,
text be Element of 4-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN)),
Key be Element of m-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN))
st 4 <= m
holds
AES-DEC(SBT,MCFunc,AES-ENC(SBT,MCFunc,text,
  Key),Key) = text
```

□

Thus, we proved in the Mizar system that the ciphertext encoded by the generalized AES algorithm can be decoded uniquely with the same secret key that was used in encryption.

6.2 AES Algorithm

In this section, we formalize the AES algorithm in the Mizar language using our formalization of the AES primitives in Section 5 and the generalized AES algorithm in Section 6.1.

First, we formalize the encryption algorithm of AES–128 as a functor in the Mizar language as follows:

Definition 6.2: (AES–128 encryption algorithm)

```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
let message be Element of 128-tuples_on
  BOOLEAN;
let Key be Element of 128-tuples_on
  BOOLEAN;
func AES128-ENC(SBT,message,Key) ->
  Element of 128-tuples_on BOOLEAN
```

```
equals
(AES-Statearray").(AES-ENC(SBT,MixColumns,
  AES-Statearray.message,AES-Statearray.
  Key));
```

□

Here, AES-Statearray" is the inverse of AES-Statearray.

Next, we formalize the decryption algorithm of AES–128 as a functor in the Mizar language as follows:

Definition 6.3: (AES–128 decryption algorithm)

```
let SBT be Permutation of (8-tuples_on
  BOOLEAN);
let cipher be Element of 128-tuples_on
  BOOLEAN;
let Key be Element of 128-tuples_on
  BOOLEAN;
func AES128-DEC(SBT,cipher,Key) ->
  Element of 128-tuples_on BOOLEAN
equals
(AES-Statearray").(AES-DEC(SBT,MixColumns,
  AES-Statearray.cipher,AES-Statearray.
  Key));
```

□

Finally, we then prove the following theorem:

Theorem 6.2: (Correctness of AES–128)

```
for SBT be Permutation of (8-tuples_on
  BOOLEAN),
message,Key be Element of 128-tuples_on
  BOOLEAN
holds
AES128-DEC(SBT,AES128-ENC(SBT,message,Key),
  Key) = message
```

□

We similarly formalized AES–192 and AES–256 using our formalization of the AES primitives and the generalized AES algorithm.

Thus, we proved using the Mizar system that the ciphertext encoded by the AES algorithm can be decoded uniquely with the same secret key that was used in encryption.

7. Conclusion

In this paper, we introduced our formalization of the AES algorithm in Mizar. We also proved the correctness of the AES algorithm using the Mizar proof checking system as a formal verification tool. Currently, we are analyzing the cryptographic systems using our formalization in order to achieve the security proof of cryptographic systems.

Acknowledgments

This work was supported by JSPS KAKENHI 21240001 and 22300285. We sincerely thank Prof. Kaneko at the Tokyo University of Science for his helpful discussions about AES.

References

[1] *Mizar Proof Checker*. *Available at* http://mizar.org/.

[2] U.S. Department of Commerce/National Institute of Standards and Technology, *FIPS PUB 46-3, DATA ENCRYPTION STANDARD (DES)*, Federal Information Processing Standars Publication, 1999. *Available at* http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[3] H.Okazaki, K.Arai, and Y.Shidama, *Formal Verification of DES Using the Mizar Proof Checker*, Proceedings of the 2011 International Conference on Foundations of Computer Science (FCS'11), pp.63–68, 2011.

[4] U.S. Department of Commerce/National Institute of Standards and Technology, *FIPS PUB 197, Advanced Encryption Standard (AES)*, Federal Information Processing Standars Publication, 2001. *Available at* http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[5] NIST Special Publication 800-67 Version 1.1, *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*, National Institute of Standards and Technology, 2008. *Available at* http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf.

[6] E.Bonarska, *An Introduction to PC Mizar*, Mizar Users Group, Fondation Philippe le Hodey, Brussels, 1990.

[7] M.Muzalewski, *An Outline of PC Mizar*, Fondation Philippe le Hodey, Brussels, 1993.

[8] Y.Nakamura, T.Watanabe, Y.Tanaka, and P.Kawamoto, *Mizar Lecture Notes (4th Edition)*, Shinshu University, Nagano, 2001. *Available at* http://markun.cs.shinshu-u.ac.jp/kiso/projects/proofchecker/mizar/index-e.html.

[9] A.Grabowski, A.Kornilowicz, and A.Naumowicz, *Mizar in a Nutshell*, Journal of Formalized Reasoning 3(2), pp.153–245, 2010.

[10] J.Daemen and V. Rijmen, *The block cipher Rijndael*, Smart Card research and Applications, LNCS 1820, Springer-Verlag, pp.288–296, 2000.

Appendix

Definition A. 1: (InvSubBytes)

```
let SBT be Permutation of (8-tuples_on BOOLEAN);
func InvSubBytes(SBT) ->
  Function of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN)),
  4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
means
for input be Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
holds
for i,j be Nat st i in Seg 4 & j in Seg 4
holds
ex inputij be Element of 8-tuples_on BOOLEAN
st inputij = (input.i).j & ((it.input).i).j = (SBT").(inputij);
```

□

Definition A. 2: (InvShiftRows)

```
func InvShiftRows ->
  Function of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN)),
  4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
means
for input be Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
holds
(for i be Nat st i in Seg 4 holds ex xi be Element of 4-tuples_on
  (8-tuples_on BOOLEAN) st xi = input.i & (it.input).i = Op-Shift(xi,i-1));
```

□

Definition A. 3: (InvMixColumns)

```
func InvMixColumns ->
  Function of 4-tuples_on(4-tuples_on (8-tuples_on BOOLEAN)),
  4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
means
for input be Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
holds
ex x,y being Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
st x = input & y = it.input & for i be Element of NAT st i in Seg 4
holds
ex x1,x2,x3,x4 be Element of 8-tuples_on BOOLEAN
st x1 = (x.i).1 & x2 = (x.i).2 & x3 = (x.i).3 & x4 = (x.i).4 &
(y.1).i = Op-XOR(Op-XOR(Op-XOR(14 'gf' x1,11 'gf' x2),13 'gf' x3),9 'gf' x4) &
(y.2).i = Op-XOR(Op-XOR(Op-XOR(9 'gf' x1,14 'gf' x2),11 'gf' x3),13 'gf' x4) &
(y.3).i = Op-XOR(Op-XOR(Op-XOR(13 'gf' x1,9 'gf' x2),14 'gf' x3),11 'gf' x4) &
(y.4).i = Op-XOR(Op-XOR(Op-XOR(11 'gf' x1,13 'gf' x2),9 'gf' x3),14 'gf' x4);
```

□

Definition A. 4: (AES-DEC)

```
let SBT be Permutation of (8-tuples_on BOOLEAN);
let MCFunc be Permutation of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN));
let m be Nat;
let text be Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN));
let Key be Element of m-tuples_on (4-tuples_on (8-tuples_on BOOLEAN));
assume 4 <= m;
func AES-DEC(SBT,MCFunc,text,Key) ->
  Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
means
ex seq be FinSequence of (4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN)))
st len seq = 7+m-1 & (ex Keyi1 be Element of 4-tuples_on (4-tuples_on
  (8-tuples_on BOOLEAN)) st Keyi1 = (Rev((KeyExpansion(SBT,m)).(Key))).1 &
  seq.1 = (InvSubBytes(SBT)*InvShiftRows).(AddRoundKey.(text,Keyi1))) &
(for i be Nat st 1 <= i & i < 7+m-1 holds
  ex Keyi be Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
  st Keyi = (Rev((KeyExpansion(SBT,m)).(Key))).(i+1) &
  seq.(i+1) = (InvSubBytes(SBT)*InvShiftRows*(MCFunc")).(AddRoundKey.
  (seq.i,Keyi))) &
ex KeyNr be Element of 4-tuples_on (4-tuples_on (8-tuples_on BOOLEAN))
st KeyNr = (Rev((KeyExpansion(SBT,m)).(Key))).(7+m) &
it = AddRoundKey.(seq.(7+m-1),KeyNr);
```

□