Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ONE-TO-MANY REVERSIBLE MAPPING FOR IPv6 ADDRESS GENERATION: SIMULATION SOFTWARE DEVELOPMENT

[1]NASHRUL HAKIEM, [2]MOHAMMAD UMAR SIDDIQI
Department of Electrical and Computer Engineering, Faculty of Engineering, IIUM
E-mail: [1]hakiem@yahoo.com, [2]umarsiddiqi@iium.edu.my

ABSTRACT

This paper presents the development of a one-to-many reversible mapping mechanism simulation for IPv6 address generation. The aim of this mechanism is to improve IPv6 addresses generation in terms of privacy and security in an enterprise local area network (LAN). Each time a user accesses a network, a dynamic IPv6 address is assigned via the DHCPv6 server. The dynamic address (one-to-many mapping) is to protect a user from unwanted behavior analysis attempting to exploit IPv6 addresses, thus protecting user privacy. However, the dynamic address can be uniquely linked to the user (many-to-one mapping) if the need arises. The one-to-many reversible mapping is generated dynamically using Cipher Feedback (CFB) mode of operation of the Advanced Encryption Standard (AES). Software simulation is developed using the software engineering waterfall model and a Unified Modeling Language (UML) class diagram as a notation. The results show that the mechanism simulates well for IPv6 address generation and IPv6 address owner identification. The one-to-many mapping may be incorporated into DHCPv6 software and many-to-one mapping may be implemented as a complement of local area network monitoring software.

Keywords: *IPv6 address, Advanced Encryption Standard, Cipher Feedback, Software Engineering, Simulation.*

1. INTRODUCTION

A one-to-many reversible mapping mechanism [1] is developed which can be embedded into the DHCPv6 (Dynamic Host Configuration Protocol for IPv6) [2] servers in the stateful mode. The aim of this mechanism is to improve IPv6 address generation in terms of privacy and security via DHCPv6 in an enterprise local area network (LAN).

Each time a user accesses the network, a dynamic IPv6 address is assigned via a DHCPv6 [3] server in the stateful mode. The dynamic address (one-to-many mapping) is to protect a user from unwanted behavior analysis attempting to exploit IPv6 addresses, thus protecting user privacy. However, the dynamic address can be uniquely linked to the user (many-to-one mapping) if the need arises to improve network visibility, thus protecting network security.

The requirement of a one-to-many reversible mapping for IPv6 address generation is that it should manage the 64-bit Interface ID part of an IPv6 address. It is assumed that the mechanism has to be able to manage up to 2^{18} registered user IDs within an enterprise local area network.

The first criterion of one-to-many reversible mapping is that the processing speed for generating an address and identifying an address must be practical. Secondly, the collision probability [4] of the Interface ID part of an IPv6 address must be very small. The proposed mechanism should integrate well with the existing mechanism (DHCPv6). Eventually, the mechanism should be able to perform validation for generating and identifying IPv6 addresses.

In this paper, the simulation of a one-to-many mapping for stateful IPv6 address generation and identification proposed in [1] is developed using waterfall model and Unified Modeling Language (UML) class diagram as a notation.

The remainder of this paper is organized as follows. Section 2 describes related works to this research and Section 3 briefly reviews one-to-many reversible mapping. Section 4 gives the simulation development. Section 5 provides the conclusion of this paper.

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

2. RELATED WORKS

2.1. IPv6 Address Generation

IPv6 has 128 bits to specify the address of a node which is represented in hexadecimal format with colon notation [5].

An example of an IPv6 address is shown in Figure 1. The first 48 bits are allocated for the network address and the following 16 bits are allocated for a subnet prefix within the network. The remaining 64 bits are allocated for the Interface ID.

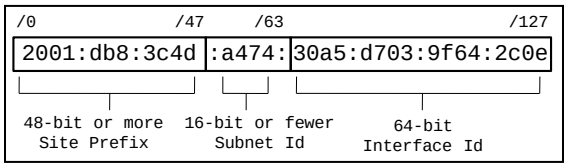

Fig. 1. IPv6 Address Example

IPv6 supports different mechanisms for assigning IP addresses to nodes [6]. This is illustrated in Table 1.

Table 1. IPv6 Address Generation Mechanism

Mechanism	Advantages	Disadvantages	Target App.	Remarks
Auto config	No DHCP server required	No control over IPv6 address usage	Unmanaged network, ad hoc net, sensor net, etc.	RFC 4941, January 2007 [7]
DHCPv6	Control over IPv6 address usage	Requires DHCP server; planning	Managed Enterprise net, home net,	RFC 3315, July 2003 [2]
CGA	Verification of address owner	Requires asymmetric key cryptography	Mobile network	RFC 3972 Oct 2005 [8], RFC 4982 Jul 2007 [9]
Multi-key CGA	Verification of address owner, enhanced mobility	Requires asymmetric key cryptography; proxy server	Mobile network	US Patent 7925027 B2, 12 April 2011 [10]
Random	Unique Local Unicast Address	Temporary address	Local Unicast	RFC 4193 October 2005 [7],[11]

2.2. Cipher Feedback

To generate a one-to-many reversible IPv6 address mapping one may use the Advanced Encryption Standard (AES) which is resistant against all known attacks [12]. There are some modes of operations for encryption; one of them is Cipher Feedback (CFB). It allows a variable size of input and output.

2.3. Software Engineering

The simulation development uses the waterfall model which has had a profound effect and influence on software development prevalent today [13]. The waterfall model is the most efficient way for creating software that provides back-end functionality [13]. It has the following steps: System Requirement and Analysis; System Design; System Implementation (coding); System Testing, and System Operation and Maintenance [14].

The development tool uses object oriented technology, therefore the notation uses a Unified Modeling Language (UML) class diagram [15]. UML has become the standard tool for object oriented modeling [13].

3. ONE-TO-MANY REVERSIBLE MAPPING

The various aspects of the development of one-to-many reversible mapping for IPv6 address generation in enterprise local area networks are:

1) IPv6 address owner identification is important for improving network visibility in order to improve the security of the enterprise local area network.
2) Changing the interface identifier, and the global scope addresses generated from it, over time makes it more difficult for eavesdroppers and other information collectors to identify the node when different addresses are used for different transactions that actually correspond to the same node [7].

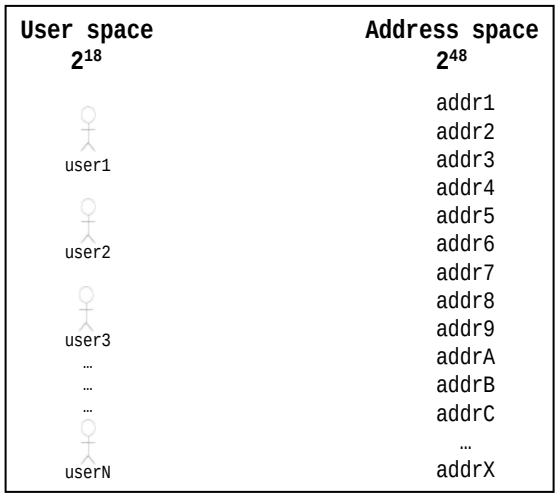

Fig. 2. One-to-many Reversible Mapping

The main requirement of a generated dynamic IPv6 address is that it should manage the 64-bit Interface ID part of the IPv6 address. It is assumed that there are a maximum of 2^{18} registered user IDs within an enterprise local area network.

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Thus, the objectives of one-to-many reversible mapping in an enterprise local area network are:

1) To identify an IPv6 address owner (user) from a particular IPv6 address in order to improve network visibility.
2) To generate dynamic IPv6 addresses for a particular user in order to protect user privacy.

4. RESULTS AND DISCUSSION

4.1. System Environment

The system simulation runs under Microsoft Windows XP Professional version 2002 Service Pack 2. The simulation was developed, compiled, and launched using the Java™ Standard Edition Runtime Environment version 1.6.0. The Netbeans IDE 6.0.1 was used as the development tool.

The processor uses AMD TurionTM X2 dual-core mobile technology RM-70 (1 MB L2 cache, 2.00 GHz, DDR2 800 MHz) CPU, supporting AMD HyperTransport 3.0 technology with RAM 1 GB DDR2 RAM.

4.2. System Requirements

The mechanism that manages the 64-bit Interface ID part of an IPv6 address uses an address format as shown in Figure 3. Another criterion of a generated dynamic IPv6 address is that the network administrator should be able to easily identify an IPv6 address owner. However, this mechanism should generate IPv6 addresses dynamically (difficult to identify) for a particular user in order to protect user privacy. The one-to-many reversible mapping has been developed using the Advanced Encryption Standard with a Cipher Feedback mode of operation (CFB-AES) [1].

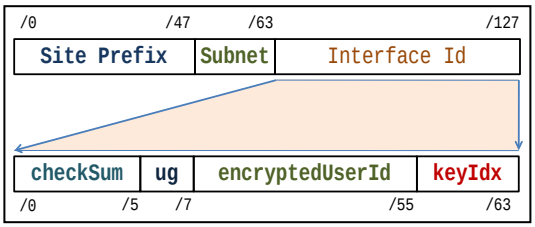

Fig. 3. Interface ID Format

4.3. System Analysis

4.3.1.Interface ID generation

The Interface ID generation is illustrated in Figure 4. Further, Figure 5 illustrates the `encryptedUserId` process. The one-to-many mapping between the 18-bit user ID and the 48-bit encrypted user ID can be represented as:

$$P \leftarrow R \mid p \qquad (1)$$

where the 48-bit user ID P is a concatenation of a 30-bit R (random number) and the 18-bit p (user ID). From (1), it can be seen that same p can have a number of P (one-to-many mapping) because of the additional R bits. However such P are clearly visible and thus conflicts with one of the objectives to protect user privacy. Therefore encryption is performed using CFB-AES which has a high avalanche effect such that any change of a bit in P may affect many bits of C significantly to produce a pseudorandom effect that actually corresponds to the same user ID p.

$$C = E(K, P) \qquad (2)$$

where E denotes the encryption of P under key K and C is the encrypted user ID which will be embedded in the Interface ID.

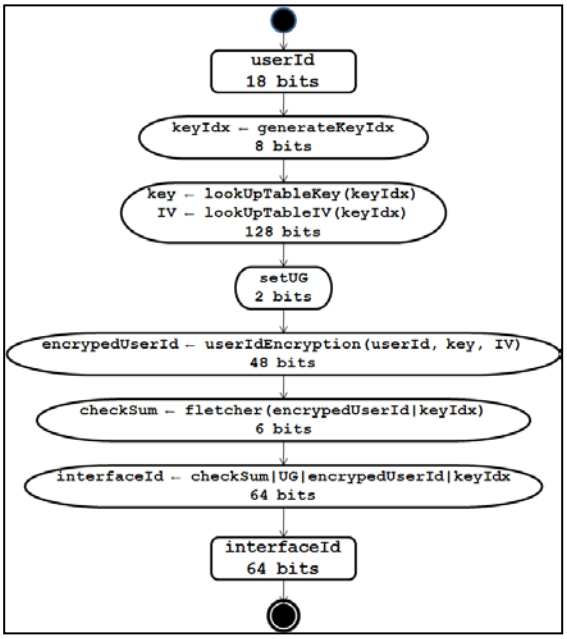

Fig. 4. Interface ID Generation

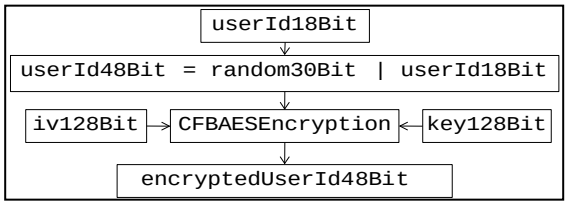

Fig. 5. User ID Encryption

The details of the CFB-AES encryption (2) are as follows:

$$C_k = P_k \oplus S_s[E(K, C_{k-1})] \qquad (3)$$

where k is the second block to the end block, while

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

the first block encryption also depends on an IV (Initialization Vector) as follows:

$$C_1 = P_1 \oplus S_s[E(K, IV)] \quad (4)$$

4.3.2.User ID identification

To obtain p to identify an 18-bit user ID from a member of C which is part of the Interface ID, we first have to perform validation first as depicted in Figure 6. The `userIdDecryption` process is illustrated in Figure 7 and can be represented as:

$$P = D(K, C) \quad (5)$$

where D denotes the decryption of C under the key K to produce a 48-bit user ID. Subsequently, simply eliminate the first 30 bits (R) from P.

$$p \leftarrow P - R \quad (6)$$

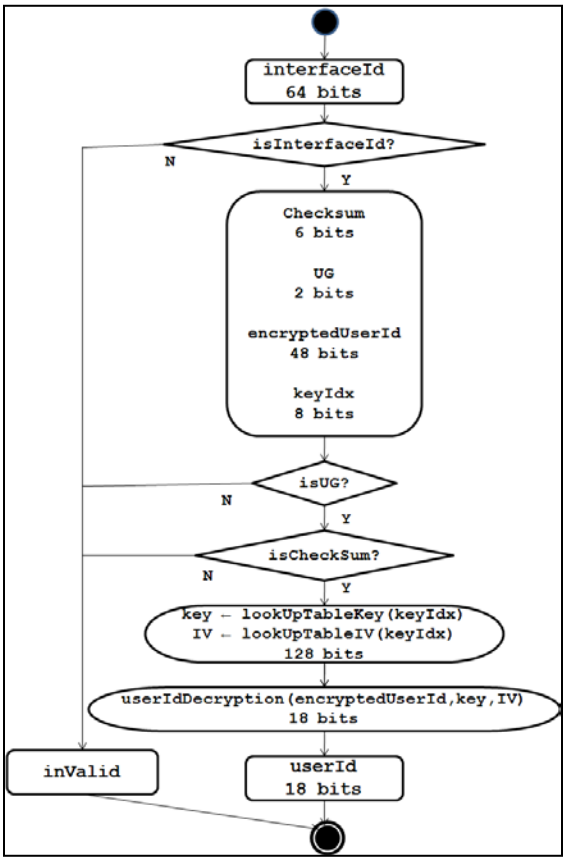

Fig. 6. User ID Identification

This produces a user ID (p) from some P (many-to-one mapping).

For the identification process, we have to obtain P from C (5). This encrypts both the first block and the rest of the blocks which can be seen in (7) and (8).

$$P_1 = C_1 \oplus S_s[E(K, IV)] \quad (7)$$

$$P_k = C_k \oplus S_s[E(K, C_{k-1})] \quad (8)$$

where k is the second block to the end of the blocks and s is the segment of unit of bits.

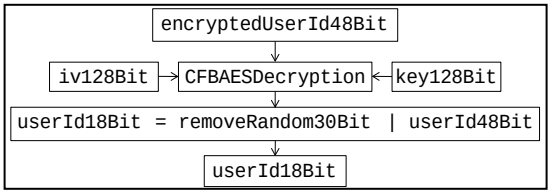

Fig. 7. User ID Decryption

4.4. System Design

4.4.1.Unified modeling language

Unified Modeling Language (UML) for object oriented technology is used for this system design [15]. Figure 8 shows the UML class diagram structure.

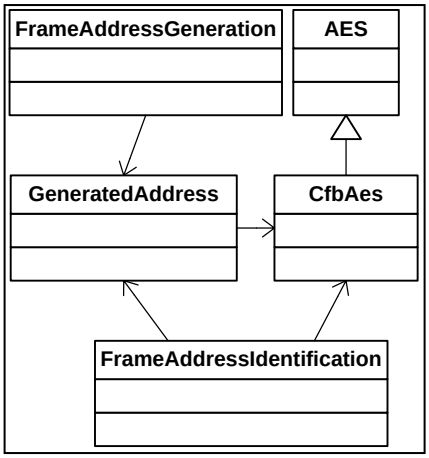

Fig. 8. UML Class Diagram Structure

Both the `FrameAddressGeneration` and the `FrameAddressIDentification` class call the `GeneratedAddress` class. The `FrameAddressIdentification` and the `GeneratedAddress` class send messages to the `CfbAes` class which is inherited from the `AES` class. The details of each class are defined in Figure 9 to Figure 12 inclusive.

Figure 9 shows the three methods of the `FrameAddressGeneration` class as being: `initialization()`, `buttonGenerate()`, and `buttonClose()`.

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

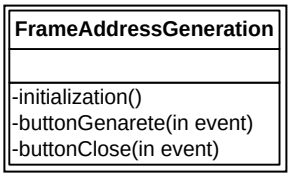

Fig. 9. FrameAddressGeneration Class Diagram

Figure 10 given below illustrates the `FrameAddressIdentification` class which also consists of three main methods: `initialization()`, `buttonIdentify()`, and `buttonClose()`.

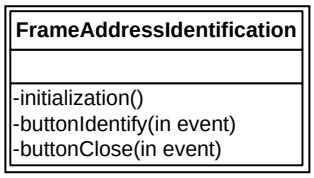

Fig. 10. FrameAddressIdentification Class Diagram

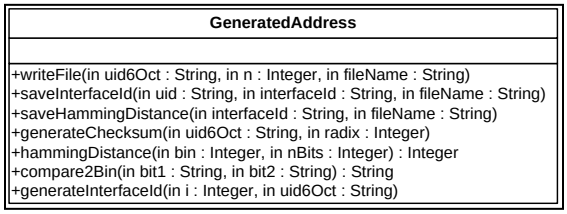

Fig. 11. GeneratedAddress Class Diagram

The `GeneratedAddress` class is the major class of this simulation development as depicted in Figure 11. There are several public methods which are directly related to Interface ID generation.

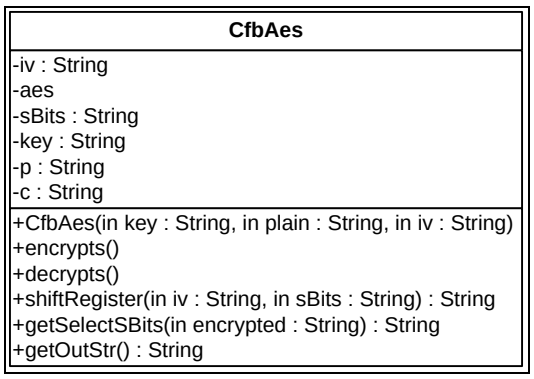

Fig. 12. CfbAes Class Diagram

The `CfbAes` class as portrayed in Figure 12 is an inheritance class of the `AES` class. It performs encryption and decryption using the Cipher Feedback mode of operation.

4.4.2.User interface

There are two user interfaces which are an IPv6 address generator as depicted in Figure 13 and an IPv6 address identification interface as depicted in Figure 14.

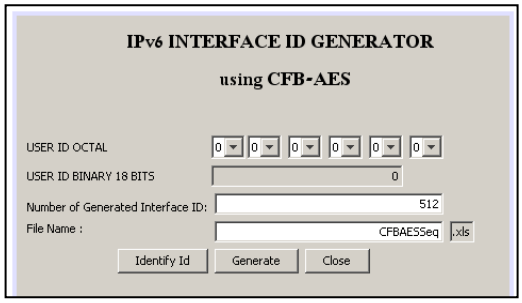

Fig. 13. IPv6 Address Generator User Interface

There are six combo buttons as input fields representing the user ID in the IPv6 address generator user interface as depicted in Figure 13. The other two input text fields represent the number of IPv6 address sequences and the file name respectively. The button `Generate` creates a sequence of Interface IDs along with other parameters and stores it in a file.

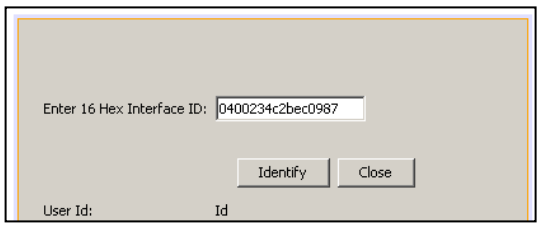

Fig. 14. IPv6 Address Identification User Interface

The IPv6 Address Identification User Interface as depicted in Figure 14 is displayed once the button `Identify Id` on the IPv6 Address Generator User Interface (Figure 13) is clicked. The IPv6 address owner message is displayed in the `User Id` text field or a specific error message is displayed if the Interface ID which has been entered does not match with the IPv6 address generation mechanism.

4.5. System Implementation

4.5.1.Address generation

Function `writeFile()` as shown in Figure 15 can be considered as the initialization function of the `GeneratedAddress` class. It has three input parameters of which is `uid6Oct` and `n` are used within the function `generateInterfaceId()`. Another parameter is `fileName` which is used in the function `saveInterfaceId()` and also `saveHammingDistance()`.

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Function `saveInterfaceId()` as depicted in Figure 16 stores a sequence of Interface IDs in the file along with their Hamming weight and their chi Square for further analysis.

```
function writeFile(uid6Oct: String, n:
int, fileName: String)
{
  fileNameHD : String = fileName.concat(
"HD" ).concat( xls );
  fileNameHeader : String = fileName.
concat( xls );
  interfaceId : array of String[n];
  for ( int i = 0; i < n; i++ )
  {
    interfaceId[i] = generateInterfaceId(
i, uid6Oct)
  }
  saveInterfaceId(uid6Oct, interfaceId,
fileNameHeader )
  saveHammingDistance( interfaceId,
fileNameHD )
} // writeFile
```

Fig. 15. Function writeFile() Pseudocode

```
function saveInterfaceId( uid6Oct:String,
interfaceId: array of String,
fileName:String )
{
  length : int = interfaceIdBin.length
  myFile : File( fileName )
  out : PrintWriter(FileWriter( myFile))
  out.printHeader()
  n1, sumN1, sumSn, sn, i : int = 0
  phi, chiSquare, sChiSquare : double = 0
  for ( i = 0; i < length; i++ )
  {
    out.print(interfaceId[i] )
    n1=countN1( interfaceId[i])
    sumN1 += n1
    sn = 2 * n1 - 62
    sumSn += sn
    phi = (double) n1 / 62
    chiSquare = ( phi - 0.5 ) ^ 2
    sChiSquare += chiSquare
    out.print( n1 + "\t" + sn  )
    out.println(phi+"\t"+chiSquare)
  }
  out.print(sChiSquare)
  out.close()
  }
}
```

Fig. 16. Function saveInterfaceId() Pseudocode

The Function `saveHammingDistance()` as depicted in Figure 17 stores the Hamming distance value of the sequence of Interface IDs. It also saves the chi square value of the Hamming distance for further analysis.

Function `compare2Binaries()` compares between two Interface ID and this result is used as a parameter input in the function `hammingDistance()` as depicted in Figure 19.

The function `hammingDistance()`calculates the difference of each bit between two Interface IDs.

```
function saveHammingDistance( interfaceId:
array of String, fileName:String )
{
  compare : String
  hammingDist, hammingDistTotal : int
  hammingDistaverage : double
  length : int
  length = interfaceId.length
  n, sn, sSn : int
  phi, chiSquare, sChiSquare : double
  myFile : File( fileName )
  out = PrintWriter(FileWriter(myFile))
  out.printHeaderHD()
  for ( int i = 0; i < length; i++ )
  {
    for (int j = i + 1; j < length; j++ )
    {
      n++;
      compare =
compare2Binaries(interfaceId[i],
interfaceId[j])
      hammingDist = hammingDistance(
compare, 64 )
      hammingDistTotal += hammingDist
      out.print((i + 1) + "|" + (j + 1))
      out.print( hammingDist)
      sn = 2 * hammingDist - 62
      out.print( sn )
      phi = (double) hammingDist / 62;
      out.print( phi )
      chiSquare = ( phi - 0.5 ) ^2
      sChiSquare += chiSquare
      out.println( chiSquare )
      sSn += sn
    }
  }
  hammingDistaverage = hammingDistTotal/n
  out.println( "n = " + ( n * 62 ) +
hammingDistTotal + sSn + sChiSquare )
  out.println( "Average = " +
hammingDistaverage )
    out.close()
  }
} // saveHammingDistance
```

Fig. 17. Function saveHammingDistance() Pseudocode

```
function compare2Binaries(bin1: String,
bin2: String ) → String
{
  bin : String
  l1 : long = bin64ToLong( bin1 )
  l2 : long = bin64ToLong( bin2 )
  bin : int2Bin ( l1 ⊕ l2 )
  ← bin
} // compare2Binaries
```

Fig. 18. Function compare2Binaries() Pseudocode

The function `generateInterfaceId()` returns a string representing of the Interface ID. The function as depicted in Figure 20 implements

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Interface ID generation diagram as described in Figure 4.

```
function hammingDistance( String bin, int
nBits ) → int
{
  c : char
  nOne : int = 0
  for ( int i = 0; i < nBits; i++ )
  {
    c = bin.charAt( i )
    if ( c == "1" )
    {
      nOne++
    }
  }
  ← nOne
} // hammingDistance
```

Fig. 19. Function hammingDistance() Pseudocode

```
function generateInterfaceId(i: int,
uid6Oct: String) → String
{
  randIn : int = randomInt()
  keyId8Bit : String = int2Bin( randIn)
  keyIndex2Hex : String = binToHex(
keyId8Bit)
  keyIdxInt:int=parseInt( keyId8Bit, 2)
  seq8Bit : String = int2Bin( i )
  key32Hex : String =
Key.CFB_KEY[keyIdxInt]
  iv32Hex : String = Key.CFB_IV[keyIdxInt]
  uid18Bit : String =octToBin( uid6Oct )
  String random30Bit = int2Bin( randIn, 30
)
  plain48Bit : String = random30Bit +
uid18Bit
  plain12Hex : String  =
binToHex(plain48Bit,12)
  cfbAes : CfbAes( key32Hex, plain12Hex,
iv32Hex )
  cfbAes.encrypts();
  encrypted12Hex : String =
cfbAes.getOutStr()
  checkSum6Bit : String
  ug2Bit : String = "00"
  String encryptedUserId48Bit = hexToBin(
encrypted12Hex, 48 );
  interfaceId : String
  checkSumOct2 : String =
generateCheckSum( encrypted12Hex +
keyIndex2Hex, 16 )
  checkSum6Bit = octToBin( checkSumOct2)
  interfaceId = checkSum6Bit + ug2Bit +
encryptedUserId48Bit + keyId8Bit
  ← interfaceId;
} // generateInterfaceId
```

Fig. 20. Function generateInterfaceId() Pseudocode

The function `generateChecksum()` as depicted in Figure 21 returns a string value. It has two parameters which are a string and an integer data type. This function is generated using the Fletcher checksum which is defined in detail in [4]. This checksum is used for both the address generation and the IPv6 address identification.

4.5.2.Address identification

The function `buttonIdentify()` in the `FrameAddressIdentification` class as depicted in Figure 22 identifies the IPv6 address owner. The identification is implemented according to the steps in the user ID identification flowchart as described in Figure 6.

```
function generateChecksum( uid:String,
radix:int ) → String
{
  c, s, y, z : String
  cInt, yInt, zInt : int
  sumY, sumZ, i  : int
  i = 0
  while ( i < uid.length() )
  {
    c = uid.substring( i, i + 1 )
    cInt = parseInt( c, radix )
    yInt = Constant * cInt
    sumY += yInt
    sumZ += sumY
    i++
  }
  sumY = sumY % 8
  sumZ = sumZ % 8
  y = toOctalString( sumY )
  z = toOctalString( sumZ )
  s = y + z
  ← s
}
```

Fig. 21. Function generateChecksum() Pseudocode

4.5.3.CFB AES

The functions below are within the `CfbAes` class. It is initialized by the function `setArrayP()` as depicted in Figure 23 which splits the plain text into an array. The function `encrypts()` as depicted in Figure 24 instantiates the `AES` class with two input parameters which are the key and the IV (initialization vector). It performs AES encryption and selects certain bits as depicted in Figure 27 (function `getSelectSBits`()). It then performs an XOR between the block of plain and the selected bits.

Function `decrypts()` is almost the same as function `encrypts()` within the `CfbAes` class. The difference is in the input during the XOR and the input within the function `shiftRegister`().

Function `shiftRegister()` as depicted in Figure 26 is used to shift certain bits of the initialization vector (IV). It has two parameters as input which are the IV or the previous phase output, and the selected bits.

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
function buttonIdentify (ActionEvent evt)
{
  interfaceId = textInterfaceId.getText()
  userId : String
  length : int = interfaceId.length()
  isInterfaceId : boolean = true
  if ( length > 16 )
  {
    interfaceId = "Interface Id's length
must be <= 16 HEXAdecimal."
    isInterfaceId = false
  }
  else if ( !g.isHex( interfaceId ) )
  {
    interfaceId = "Interface Id must be in
HEXAdecimal."
    isInterfaceId = false
  }
  isUG = isUGunction(interfaceId)
  intefaceIdOct6 : String = interfaceId
  if ( isInterfaceId )
  {
    if (isUG)
    {
      key2Hex : String =
interfaceId.substring( 14, 16 );
      checkSum8Bin : String =
Common.hexsToBin( checkSumHex, 8 );
      checkSum2Oct : String =
Common.binToOct( checkSum8Bin.substring(
0, 6 ), 2 );
      encrypted12Hex : String =
cfbAes.getOutStr();
      encrypted16Oct : String = hexToOct(
encrypted12Hex, 16 );
      generatedCheckSumOct2 : String =
g.generateCheckSum ( encryptedUserId12Hex
+ key2Hex, 16 );
      if ( checkSum2Oct.equalsIgnoreCase(
generatedCheckSumOct2 ) )
      {
        intefaceIdOct6 = plainUserId6Oct
      }
      else
      {
        intefaceIdOct6 = "Checksum does
not match."
      }
    }
    else
    {
      intefaceIdOct6 = "Interface Id is
not Universal nor Group."
    }
  }
  userId = intefaceIdOct6
  jLabelUserId.setText( userId )
  jLabelRandom.setText( plainRandom30Bit )
}
```

Fig. 22. Function buttonIdentify() Pseudocode

Function `getSelectSBits()` which is depicted in Figure 27 has an `encrypted` parameter as a string data type. It selects certain leftmost bits of the `encrypted` parameter and discards the others.

```
function setArrayP(plain: String )
{
  for ( int i = 0; i < 6; i++ )
  {
    this.p[i] = plain.substring( 2 * i, 2
* i + 2 );
  }
}
```

Fig. 23. Function CFB-AES setArrayP() Pseudocode

```
function encrypts()
{
  int cInt;
  for ( int i = 0; i < 6; i++ )
  {
    aes[i] = AES(key,iv[i]  )
    aes[i].encrypts()
    sBits[i] = getSelectSBits(
aes[i].getOutStr() )
    cInt = parseInt( p[i], 16 ) ⊕
parseInt( sBits[i], 16 )
    c[i] = intToHex( cInt, 2 )
    if ( i < 5 )
    {
      iv[i+1] = shiftRegister(iv[i], c[i])
    }
  }
}
```

Fig. 24. Function CFB-AES encrypts() Pseudocode

```
function decrypts()
{
  cInt : int
  for ( int i = 0; i < 6; i++ )
  {
    aes[i] = AES(key,iv[i]  )
    aes[i].encrypts()
    sBits[i] = getSelectSBits(
aes[i].getOutStr() )
    cInt = parseInt( p[i], 16 ) ⊕
parseInt( sBits[i], 16 )
    c[i] = intToHex( cInt, 2 )
    if ( i < 5 )
    {
      iv[i+1] = shiftRegister(iv[i], p[i])
    }
  }
}
```

Fig. 25. Function CFB-AES decrypts() Pseudocode

```
function shiftRegister(iv: String, sBits:
String ) → String
{
  ← iv.substring( 2, 32 ) + sBits;
}
```

Fig. 26. Function CFB-AES shiftRegister() Pseudocode

```
function getSelectSBits(encrypted :
String) → String
{
  ← encrypted.substring( 0, 2 )
}
```

Fig. 27. Function CFB-AES getSelectSBits()

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Pseudocode

Function `getOutStr()` as depicted in Figure 28 concatenates the array of ciphertext. It returns a string parameter representing ciphertext.

```
function getOutStr() → String
{
  out : String
  for ( int i = 0; i < 6; i++ )
  {
    out += this.c[i]
  }
  ← out
}
```

Fig. 28. Function CFB-AES getOutStr() Pseudocode

4.6. System Testing

The last step before system operation and maintenance during the waterfall model in the software engineering methodology is system testing. Black box testing was used which mainly refers to functional testing [16].

User interface testing checks the functionality of each component which should work properly including the combo boxes, the input text fields, and the buttons.

Button `Generate` in Figure 29 calls function `writeFile()` in the `GeneratedAddress` class. It generates a sequence of Interface IDs as part of the IPv6 address based on user identity. It then stores them in the file along with other supplementary parameters for further analysis.

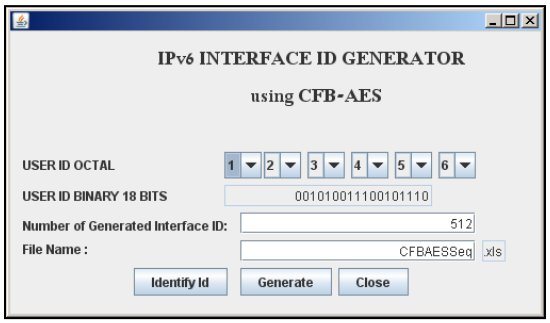

Fig. 29. IPv6 Address Generation Frame

Button `Identify Id` in Figure 29 enables the address identification frame as depicted in Figure 30. This frame has a text field for the Interface ID input. The button `Identify` within this frame calls the function `buttonIdentify()` as depicted in Figure 22. The Interface ID owner or an error message is displayed in the user ID text field. Firstly, it checks the Interface ID format, then it checks the `ug` bits, and finally it compares the checksum value.

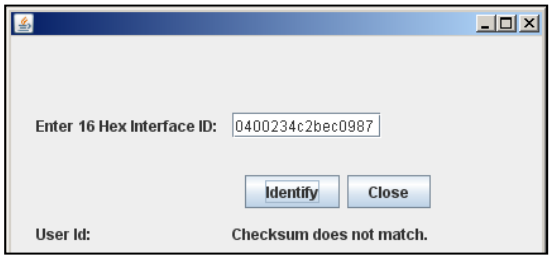

Fig. 30. IPv6 Address Identification Frame

5. CONCLUSION

The simulation of a one-to-many reversible mapping for IPv6 address generation and identification has been developed using the waterfall model in the software engineering methodology. It shows that IPv6 address generation and identification can be well simulated and the data saved to a file for further analysis. The IPv6 address generation may be incorporated into DHCPv6 and the IPv6 address owner identification may be implemented as a complement of local area network monitoring software.

REFERENCES:

[1] N. Hakiem, A. U. Priantoro, M. U. Siddiqi, and T. H. Hasan, "Generation of IPv6 Addresses Based on One-to-Many Reversible Mapping Using AES," in *Recent Progress in Data Engineering and Internet Technology*, Lecture Notes in Electrical Engineering. vol. 157, F. L. Gaol, Ed., ed Heidelberg: Springer-Verlag Berlin, 2012, pp. 183-189.

[2] RFC 3315, *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*, Standards Track, IETF, Network Working Group, 2003

[3] N. Hakiem, *et al.*, "Implementation of IPv6 address generation mechanism for enterprise wireless local area network in open source DHCPv6," in *2010 International Conference on Computer and Communication Engineering (ICCCE)*, Kuala Lumpur, Malaysia, 2010, pp. 1-5.

[4] N. Hakiem, M. U. Siddiqi, and S. P. W. Jarot, "Collision probability of one-to-many reversible mapping for IPv6 address generation," in *Computer and Communication Engineering (ICCCE), 2012 International Conference on*, Kuala Lumpur Malaysia, 2012, pp. 599-602.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[5] RFC 4291, *IP Version 6 Addressing Architecture*, Standards Track, IETF, Network Working Group, 2006

[6] N. Hakiem, M. U. Siddiqi, and S. P. W. Jarot, "Secure IPv6 Address Generation," in *Topics in Coding, Cryptography and Information Security*, M. U. Siddiqi, *et al.*, Eds., First ed Kuala Lumpur Malaysia: IIUM Press, 2011, pp. 183-189.

[7] RFC 4941, *Privacy Extensions for Stateless Address Autoconfiguration in IPv6*, Standards Track, IETF, Network Working Group, 2007

[8] RFC 3972, *Cryptographically Generated Addresses (CGA)*, Standards Track, IETF, Network Working Group, 2005

[9] RFC 4982, *Support for Multiple Hash Algorithms in Cryptographically Generated Addresses (CGAs)*, Standards Track, IETF, Network Working Group, 2007

[10] J. Kempf and C. B. Gentry, "Secure address proxying using multi-key cryptographically generated addresses," US Patent US 7925027 B2, 12 April 2011, 2011.

[11] RFC 4193, *Unique Local IPv6 Unicast Addresses*, Standards Track, IETF, Network Working Group, 2005

[12] W. Stallings, *Cryptography and Network Security, Principles and Practices*, 4 ed.: Pearson Prentice Hall, 2006.

[13] N. B. Ruparelia, "Software development lifecycle models," *SIGSOFT Softw. Eng. Notes*, vol. 35, pp. 8-13, 2010.

[14] W. W. Royce, "Managing the Development of Large Software Systems," in *IEEE WESCON*, 1970, pp. 1-9.

[15] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 6th printing ed. Massachusetts: Addison Wesley, 2000.

[16] J. Pan, *Software Testing*: Carnegie Mellon University, 1999.