

단대단 비동기 암호통신에서 효율적인 전송을 위한 데이터 변환방법

정 현 철†

요 약

비동기 통신에서는 특정 영역의 데이터가 각종 제어문자로 사용된다. 그러므로 데이터를 송신할 때 이 영역의 데이터는 제어문자로의 오인을 방지하기 위하여 다른 문자로 변환하고 이를 제어접두문자와 함께 전송하여야 한다. 본 논문에서는 비동기 프로토콜 상으로 데이터를 암호화하여 전송할 때 발생하는 모든 제어문자에 대해 제어 접두문자를 추가하지 않고 일정한 변환만 하여 송신함으로써 데이터의 길어짐을 방지하고 전체 통신 속도를 높이는 문자 변환 방법을 몇 가지 제시하였다. 이러한 변환을 위해 전송 데이터의 유효 범위를 가정하고 이 범위를 벗어나지 않도록 하였으며 실험을 통하여 이 방법이 기존의 방법에 비해 통신속도가 향상됨을 보이고 암호화된 데이터의 임의성을 확인하므로써 암호화에 문제가 없음을 보였다.

Data Conversion Schemes for Efficient Transmission on End-to-End Asynchronous Secure Communication

Hyon Cheol Chung†

ABSTRACT

In asynchronous communication data of specific area are used as all kinds of control characters. Therefore, data of this area must be converted to other character and transmitted followed by control prefix to prevent the misconception as control characters. This paper presents several methods for character conversion that prevent the lengthening of data and enhance the overall efficiency of communication by transmitting with a certain conversion and without control prefixes on control-like characters occurring when data are transmitted with ciphering onto asynchronous communication path. For such conversion, the scope of transmitted data was supposed and efforts were made not to exceed that scope. Experiments showed that this method is better in communication speed than the existing ones and that ciphering has no problem by confirming the randomness of ciphered data.

1. 서 론

최근 하드웨어 및 소프트웨어의 발달과 개인용 컴

퓨터의 보급으로 전화선을 이용한 전자 게시판 등의 이용이 급격히 늘고 있다. 대부분의 전자 게시판에서는 주로 비동기 프로토콜을 사용하여 통신하기 때문에 비동기 프로토콜의 사용은 계속 증가하는 추세이다. 이와 같은 비동기 프로토콜 사용의 증가와 함께 그에 대한 정보보호도 시급한 실정이다. 본 논문에서

† 정 회 원: 한국전자통신연구소

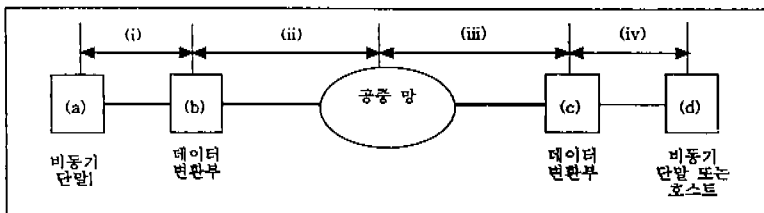
논문접수: 1996년 10월 1일, 심사완료: 1996년 11월 22일

는 비동기 데이터의 정보보호시 제어문자와 같은 ASCII 값을 가지면서 실제 제어문자가 아닌 데이터에 대해 제어접두문자(Control Prefix)를 사용하므로써 발생하는 데이터의 길어짐과 그에 따른 통신속도의 저하를 방지하는 데이터의 변환 방법을 제안한다.

일반적으로 비동기 통신에서 데이터의 전송은 문자 단위로 이루어지며 한 문자는 7비트 또는 8비트로 이루어진다. 하위 7비트가 00h(16진수 00)부터 1Fh 사이인 문자와 7Fh인 문자는 기능제어 또는 흐름제어를 위해 사용될 수 있으며 비동기 통신에서는 이들 중 일부 또는 전부를 제어문자로 사용하고 있다. 그런데 이진데이터 중에 이런 코드 값들을 갖는 문자가 나타날 수 있다.^{[1][2][3]} 이렇게 제어문자가 아니면서 제어문자와 같은 값을 갖는 문자를 모의 제어문자(Control-like Character)라고 정의한다. 모의 제어문자를 그대로 전송하면 제어문자로 인식되어 통신 요소로 하여금 제어동작을 하도록 하므로써 한 바이트의 데이터를 잃어버리게 된다. 특히 암호통신 또는 압축 데이터의 송신 등 데이터에 일정한 변환을 하는 경우는 통신중에 잃은 한 바이트의 데이터로 인해 수신측에서 데이터를 원래의 데이터로 재변환할 때 많은 데이터를 잃어버리는 경우가 발생하게 된다. 왜냐하면 모의 제어문자는 제어문자가 아닌 데이터이며 이것을 그대로 송신하면 데이터(모의 제어문자)를 제어문자로 인식하여 그에 해당하는 제어동작을 행하며 결국 이는 데이터로 처리되지 않아 한 바이트를 잃어버리게 되기 때문이며, 암호화된 데이터나 압축된 데이터에서 이런 모의 제어문자가 발생하여 한바이트를 잃어버리게 되면 원래의 데이터로 재변환할 때 여러 바이트를 잃어 버릴 수 있기 때문이다. 이런 문제를 해결하기 위해서는 이 모의 제어문자를 다른 형태로

변환해야 하는데 단지 변환만이 아니라 수신측에서 일정한 방법의 재변환으로 원래 데이터를 얻을 수 있어야 한다. 일반적으로는 Kermit 프로토콜에서 제공하는 '#'(23h) 추가 후 40h과 배타적 논리합(XOR)을 취하는 방법이 많이 사용된다. 물론 '#' 대신 제어문자가 아닌 어떤 문자를 사용해도 무방하며 이것을 제어접두문자라고 한다.^[1] 그러나 이 방법은 최악의 경우 '#'이 너무 많이 추가되어 데이터에 변환을 행하는 중간 노드((그림 1)의 (b) 또는 (c) 입장에서 볼 때 수신하는 데이터에 비해 전송해야할 데이터가 현격히 늘어나는 단점이 있다. 즉, 텍스트 데이터 자체는 모의 제어문자를 갖지 않지만 정보를 보호하기 위해 데이터를 변환하는 경우 텍스트 데이터도 난수에 가까운 이진 데이터가 되기 때문에 모의 제어문자가 많이 발생할 수 밖에 없다. 또 이진 데이터의 경우도 단말에서 이미 '#'을 추가하여 제어문자 처리를 하였더라도 데이터 변환부에서 암호화를 행하면 다시 이진 데이터가 되므로 이중으로 처리 해야하며 데이터의 길이 또한 더 길어진다.

최근에 프로세서의 속도가 빨라지면서 암호화 등 데이터 송신 전의 처리는 통신의 전체적인 속도 또는 전송률에 큰 지장을 주지 않게 되었다. 그러나 예를 들어 통신로 상의 한 노드인 (그림 1)의 (b)가 일정 전송률로 데이터를 받아서 그 데이터에 어떤 데이터를 더 붙여서 보낸다면 프로세서의 속도에 관계없이 (그림 1)의 (b)나 구간 (ii)와 같은 특정 노드 및 구간에 많은 부하가 걸릴 것이며, 이로 인해 발생하는 흐름제어의 횡수도 늘어나 통신 속도는 더 떨어질 것이다. 이는 그 노드의 프로세서가 아무리 빨리 데이터의 암호화를 처리한다고 하더라도 다음 노드로 전송하는 전송률이 일정하기 때문에 통신 속도의 저하는 피



(그림 1) 전형적인 비동기 통신망
(Fig. 1) Typical Network of Asynchronous Communication

할 수가 없기 때문이다. 이 문제의 해결을 위해 본 논문에서는 비동기 통신에서 데이터를 암호화하여 송신할 때 모의 제어문자가 발생하지 않도록 하므로써 특정 구간에서 일어나는 병목현상을 없애고 전체적인 전송효율을 높이는 방법을 제시한다.

논문의 효율적인 전개를 위해 본 논문의 2절에서는 Kermit 프로토콜에서의 모의 제어문자 처리방법을 소개하고 암호통신시 발생하는 문제점을 기술하였으며, 3절에서는 니블(Nibble, 4비트) 단위로 배타적 논리합을 취하여 문자의 추가가 없이 모의 제어문자를 처리하는 방법과을 제시하고 배타적 논리합이 아닌 합과 모듈로 연산을 이용한 방법을 제시한다. 또, 4절에서는 각 방법에 대한 실험 및 결과를 제시하고 분석하며 5절에서 결론을 맺는다.

2. Kermit 프로토콜에서의 모의 제어문자 처리 방법

Kermit 프로토콜은 화일을 전송하기 위해 설계된 비동기 프로토콜이다. 이 프로토콜은 XMODEM과

비슷한 절차로 수행된다.^[2] 이 프로토콜은 ASCII 뿐만 아니라 이진 화일까지 처리 가능한데 본 절에서는 이진 화일 송수신시 모의 제어문자를 처리하는 방법에 대해 언급한다.

Kermit에서는 화일 송신 중 모의 제어문자가 나타나면 이 문자에 대해 40h 값과 배타적 논리합을 취한 후에 '#'을 앞에 삽입하여 송신하는 방법을 사용한다. 일반적으로 제어문자는 하위 7비트의 값이 00h에서 1Fh 사이이거나 7Fh인 문자(00h~1Fh, 7Fh, 80h~9Fh, FFh)로 정의되는데 40h과 배타적 논리합을 취하면 3Fh에서 5Fh 사이의 값으로 변환되므로 모의 제어문자의 발생은 막을 수 있다. 그러나 이 값은 원래 3Fh에서 5Fh 사이의 값을 갖는 데이터와 중복되어 수신측에서 구분을 할 수 없으므로 이 변환된 값을 송신하기 전에 '#'과 같은 제어점두문자를 먼저 송신한 후에 해당 문자를 송신하게 된다. 이렇게 하면 수신측에서는 데이터를 수신하면서 만나는 '#'을 제거한 후 바로 다음 문자를 40h과 배타적 논리합을 취하여 원래의 데이터를 얻게 된다. 이 '#' 또한 원래의 '#'과 구분하여야만 수신측에서 '#'을 제거하더라도 데이터를 잃

| | |
|---|---|
| <pre>while(1) { Getch(byte); ch = byte & 0x7f; if (ch<0x20 ch==0x7f) { byte = byte ^ 0x40; Send_char("#"); } else if (ch == '#') Send_char("#"); Send_char(byte); } </pre> <p style="text-align: center;">(a) 송신시의 처리과정</p> | <pre>while(1) { Getch(byte); if (byte == '#') { Getch(byte); ch = byte & 0x7f; if (ch != '#') byte = byte ^ 0x40; } Accept(byte); } </pre> <p style="text-align: center;">(b) 수신시의 처리과정</p> |
|---|---|

(그림 2) Kermit 프로토콜에서의 모의 제어문자 처리과정
(Fig. 2) A method of processing control-like characters on kermit protocol

| | |
|--|---|
| <pre>A1 D7 23 F5 CA 07 59 3F 3D 90 1F 2D 6A 48 BD 39 06 40 2A 35 4C 6B 79 DE 08 EB B1 31 74 6C 1C 20 3F 54 96 8B A3 9D 30 0A</pre> <p style="text-align: center;">(a) 원시 데이터</p> | <pre>A1 D7 23 23 F5 CA 23 47 59 3F 3D 90 23 5F 2D 6A 48 BD 39 23 46 40 2A 35 4C 6B 79 DE 23 48 EB B1 31 74 6C 23 5C 20 3F 54 96 23 CB A3 23. 9D 30 23 4A</pre> <p style="text-align: center;">(b) 변환된 데이터</p> |
|--|---|

(그림 3) 원시 데이터 및 Kermit에 의해 변환된 데이터
(Fig. 3) Source and converted with kermit protocol

어버리는 일이 없을 것이다. 이를 위해 송신측에서는 원래의 '#'에 대해서도 '#'을 하나 더 붙여서 송신하며 수신측에서는 '#'이 연속해서 나타날 경우 짝수번째의 '#'만을 취한다. (그림 2)에 이진 데이터에 대한 Kermit의 모의 제어문자 처리과정¹¹⁾을 나타내었으며 (그림 3)에 일련의 이진 데이터에 대해 모의 제어문자 처리가 된 결과를 나타내었다.

그러나 전송하고자 하는 화일이 암호화된 데이터와 같이 거의 난수에 가까운 데이터이면 추가되는 '#'으로 인해 송수신해야 할 데이터가 너무 늘어나는 단점이 있다. 즉 kermit의 경우 하위 7비트가 00h에서 1Fh 사이이거나 7Fh인 데이터와 '#'에 대해 '#'을 하나씩 추가하여야 하며 이는 확률적으로 34/128의 데이터 증가를 유발한다. 이 수치는 원래 데이터 이외에 1/4이상의 데이터를 더 송수신 해야하는 부담이 되는 것이다. 이 통신이 암호화를 하지 않는 일반 통신일 때는 송신부에서 모의 제어문자 처리를 거친 후 송신되므로 병목 현상은 일어나지 않으나 암호화 송수신의 경우는 문제가 달라진다. (그림 1)과 같은 구조에서 송신부의 통신 단말(a)는 모의 제어문자 처리를 한 후의 데이터를 데이터 변환부(b)로 송신할 것이다. 데이터 변환부에서는 이 데이터를 암호화 하고나서 다시 난수가 된 데이터에 대해 같은 방법으로 모의 제어문자 처리를 해야한다. 이 때 통신 단말에서는 일정한 데이터 전송률로 송신하는데 데이터 변환부에서는 이 데이터를 34/128 만큼 늘려서 전송하면 병목현상이 일어나지 않을 수 없을 것이다. 이렇게 되면 송신측의 데이터 변환부(b)에서는 데이터를 잃어버리지 않기 위해 지속적인 흐름제어를 해야할 것이고 이 때문에 전체적인 통신 속도는 더 많이 저하된다. 실질적인 통신속도의 비교 결과는 4절에 나타내었다.

3. 모의 제어문자를 제거하기 위한 데이터 변환 방법

본 절에서는 모의 제어문자의 발생을 근본적으로 제거하여 제어점두문자를 송신하지 않게 하므로써 추가 데이터를 송신하지 않도록하는 데이터 변환 방법을 제시한다. 제시된 방법은 크게 두가지이며 각각 니블 단위의 배타적 논리합을 이용한 방법과 합과 모

둘로 연산을 이용한 방법이다.

3.1 니블 단위의 배타적 논리합을 이용한 방법

본 절에서는 단말에서 모의 제어문자 처리가 되어 데이터 변환부에 도달한 평문 데이터는 00h부터 1Fh까지의 값과 7Fh 값을 갖지 않는다는 점을 이용하여 암호화시에 원래의 데이터 값과 조화시켜서 전송하는 방법을 제시한다. 이 방법은 스트림 암호화를 이용한 데이터 암호화 및 복호화시에 주로 많이 사용되는 키 수열과 데이터를 배타적 논리합하는 방법^{12)에}과 거의 흡사하나 바이트 단위의 연산이 아닌 니블 단위의 연산을 수행하고 그 결과에 따라 다른 방법의 연산을 행한다.

3.1.1 니블 단위로 배타적 논리합을 취하는 방법

(그림 1)의 데이터 변환부는 송신부로부터 받은 데이터를 (b)의 암호화부에서 생성한 키 수열과 바이트 단위로 배타적 논리합을 수행하여 이 값이 모의 제어문자이면 상위 니블의 하위 3비트는 입력 데이터를 그대로 취하고 상위 니블의 최상위 비트 및 하위 니블은 배타적 논리합으로 연산된 값을 취하여 이 바이트를 전송한다. 제어문자는 상위 니블의 하위 3비트 값이 '0' 또는 '1'이며 하위 니블의 값은 0h~Fh 중 어느 값이 될 수도 있다. 즉, 모의 제어문자가 되지 않기 위해서는 상위 니블의 하위 3비트만 '1' 보다 큰 값이 되면 된다. 입력된 문자는 상위 니블의 하위 3비트 값이 2h~7h 이므로 데이터 변환시 상위 니블을 그대로 취하면 변환된 데이터는 모의 제어문자가 아니며 변환된 데이터의 하위 니블은 배타적 논리합에 의해 변환된 값이므로 이 바이트 자체는 변환되기 전의 값과는 다르다. 복호화시에도 암호화시와 같은 방법으로 수행하는데, 먼저 키와 데이터에 대해 배타적 논리합 연산을 수행한 후 복호화 된 데이터의 상위 니블이 '0', '1', '8', 또는 '9'이면 복호화 하기 전의 데이터의 상위 니블을 취한다. 왜냐하면 원시 데이터는 상위 니블이 '0', '1', '8', 또는 '9'가 될 수 없기 때문이다. 이 방법을 사용할 경우 텍스트 데이터를 이진 데이터로 변환하여 송신할 때 데이터의 길이가 늘지 않았기 때문에 송신부에서의 흐름제어도 현격히 줄어 전체적인 전송률은 크게 상승하였다. (그림 4)에 니블 단위의 배타적 논리합을 이용한 모의 제어문자 처리 방법

```

while(1) {
    Receive_char(byte);
    Getkey(key);
    ch = byte ^ key;
    temp = ch & 0x7f;
    if (temp < 0x20) {
        ch = byte ^ (key & 0x8f);
        if ((ch & 0x7f) == 0x7f) ch = byte ^ (key & 80);
    }
    else if (temp == 0x7f) ch = byte ^ (key & 0x80);
    Send_char(ch);
}
    
```

(그림 4) 배타적 논리합을 한 번 취하여 모의 제어문자를 처리하는 방법
 (Fig. 4) Processing control-like characters with single XORing

의 간략한 알고리즘을 나타내었다. 송신시와 수신시 모두 같은 알고리즘을 적용하면 된다.

이상에서 언급한 방법은 통신속도 면에서는 문제가 없으나 니블 단위의 처리가 아닌 정상적으로 암호화된 데이터에 비해 상위 니블이 원래의 데이터 값과 일치할 확률이 크다. 정상적인 방법으로 암호화 한 경우 상위 니블이 원래의 데이터의 상위 니블과 일치할 확률은(키 수열의 상위 니블이 '0'인 경우) 1/16인데 반해 이 방법을 이용하면 암호화된 데이터의 상위 니블이 '0' 또는 '1'인 경우까지 포함되어 1/16 + 1/8 이 된다. 이 때문에 제3자로부터의 공격이 쉽게 이루어질 것 같으나 실제로 그런지 검증하기는 어렵다. 암호화시 모의 제어문자로 나타난 데이터의 상위 니블이 항상 일정한 값으로 매핑되는 것이 아니라 원시 데이터 값에 따라 다르게 나타나기 때문이다. 다만 원시 데이터가 모의 제어문자를 처리한 이진 데이터 라면 원시 데이터 자체가 임의성이 있으므로 문제가 없으나 텍스트 데이터의 경우는 일정 범위의 데이터가 자주 나타나 임의성을 잃어버릴 수가 있다. 이 문제는 다음 소절에 제시한 방법으로 해결할 수 있다.

3.1.2 니블 단위의 배타적 논리합을 두 번 취하는 방법

앞 소절에서 제시한 방법으로 니블 단위의 배타적 논리합을 한 번 취하게 되면 상위 니블이 원시 데이터와 일치할 확률은 1/16 + 1/8이다. 그런데 텍스트 데이터의 경우 자주 나타나는 문자가 일정하므로 이

방법을 이용하면 암호화된 데이터에 원시 데이터와 상위 니블이 일치하는 데이터가 일정 영역(예를 들면 41h~7Ah)에서 많이 나타날 수 있다. 이렇게 되면 암호화된 데이터 수열이 임의성을 잃어버릴 수 있다.

이 문제를 해결하기 위해서 같은 변환 방법을 한번 더 취하는 방법 즉, 암호화된 데이터에 대해 또 다른 키 수열과 배타적 논리합을 취한 후 이 값이 모의 제어문자이면 상위 니블을 그대로 취하고 하위 니블은 암호화된 값을 취하는 방법을 사용할 수 있다. 이 방법은 키 수열을 두배로 생성해야 한다는 점과 배타적 논리합 연산도 두배로 해야한다는 점이 오버헤드로 작용하지만 19200bps 정도의 비동기 통신에서는 요즘의 프로세서 속도가 빠르므로 큰 영향을 미치지 않는다. 복호화시에도 3.1절의 복호화 과정을 두 번 하면된다. 이 방법을 사용할 경우 상위 니블이 원시 데이터의 상위 니블과 일치할 확률은 $(1/16 + 1/8)^2 = 9/256$ 으로 줄어 정상적인 배타적 논리합을 이용한 방법의 1/16 보다 낮은 확률을 가지기 때문에 3.1절에서 언급한 방법에서 나타나는 문제점이 해결되었다. 또, 한 번 암호화 처리된 데이터가 원시 데이터에 비해서는 임의성을 가지므로 두 번의 배타적 논리합을 이용한 암호화 방법은 임의성 면에서 배타적 논리합을 한 번 취한 방법에 비해 확실히 나아진다고 할 수 있다. (그림 5)에 이 방법의 알고리즘을 나타내었다. 니블 단위의 배타적 논리합을 한 번 취한 방법에서는 송신과 수신시의 과정이 같으나 두 번 취하는 방법에서는 두 개의 키 중 두 번째 키를 먼저 사용하

| | |
|--|--|
| <pre> while(1) { Receive_char(byte); Getkey(key); ch = byte ^ key; temp = ch & 0x7f; if (temp < 0x20) { ch = byte ^ (key & 0x8f); if((ch & 0x7f) == 0x7f) ch = byte ^ (key & 80); } else if (temp == 0x7f) ch = byte ^ (key & 0x80); byte = ch; Getkey(key); ch = byte ^ key; temp = ch & 0x7f; if (temp < 0x20) { ch = byte ^ (key & 0x8f); if((ch & 0x7f) == 0x7f) ch = byte ^ (key & 80); } else if (temp == 0x7f) ch = byte ^ (key & 0x80); Send_char(ch); } </pre> <p>(a) 송신시의 변환방법</p> | <pre> while(1) { Receive_char(byte); Getkey(key_1); Getkey(key_2); ch = byte ^ key_2; temp = ch & 0x7f; if (temp < 0x20) { ch = byte ^ (key_2 & 0x8f); if((ch & 0x7f) == 0x7f) ch = byte ^ (key_2 & 80); } else if (temp == 0x7f) ch = byte ^ (key_2 & 0x80); byte = ch; ch = byte ^ key_1; temp = ch & 0x7f; if (temp < 0x20) { ch = byte ^ (key_1 & 0x8f); if((ch & 0x7f) == 0x7f) ch = byte ^ (key_1 & 80); } else if (temp == 0x7f) ch = byte ^ (key_1 & 0x80); Send_char(ch); } </pre> <p>(b) 수신시의 변환방법</p> |
|--|--|

(그림 5) 배타적 논리합을 두 번 취하여 모의 제어문자를 처리하는 방법
(Fig. 5) Processing control-like characters with double XORing

여야 한다. 왜냐하면 이 방법이 단순한 배타적 논리합을 취한 것이 아니라 니블 단위로 서로 다른 과정을 거칠 수 있기 때문이다. 예를 들어 'key_1'과 'key_2'가 각각 55h, 66h이고 'ch'가 44h일 때 이 방법에 의해 암호화하면 37h이다. 이를 'key_1'을 먼저 사용하여 복호화를 하면 74h가 되어 원하는 값을 얻을 수가 없고 'key_2'를 먼저 사용해야만 44h를 얻을 수 있다.

3.2 합과 모듈로 연산을 이용한 방법

앞에서 언급한 방법들은 일반적으로 암호화시 많이 사용되는 키 수열과 데이터의 배타적 논리합을 이용한 것이다.^{[5][6]} 암호화시 배타적 논리합을 많이 사용하는 이유는 복호화가 빠르고 프로그램하기가 편하기 때문이다. 그러나 이 방법은 제어문자를 고려한 비동기 통신시에는 배번 데이터의 범위를 비교하여야 하고 확률적으로 1/8의 데이터에 대해 배타적 논리합을 두 번 해야 한다. 게다가 텍스트 데이터에 대해 배타적 논리합을 한 번 행한 방법은 데이터가 길어질 경우 암호화된 데이터의 임의성에 문제가 있으

며, 두 번 행한 방법은 최악의 경우 한바이트에 대해 4번의 배타적 논리합과 2번의 비교를 행하게 되어 수행 속도면에서 약간의 오버헤드(4% 이내의 전송시간 증가)를 가질 수 있다. 본 절에서 언급하고자 하는 덧셈과 모듈로 연산을 이용한 암호화는 모의 제어문자의 발생을 방지하므로써 배타적 논리합을 이용하는 방법에 비해 연산속도면에서 오버헤드가 없으며 데이터의 임의성에도 문제가 없다.

3.2.1 키 수열의 도메인을 256개의 원소로 구성하는 방법

이 방법은 데이터를 바이트 단위로 암호화하기 위해 00h~FFh의 데이터를 키 수열로 생성하여 사용하는 방법인데 암호복호화 방법을 간략히 설명하면 다음과 같다.

먼저 암호화시에는 (그림 6)의 (a)에 나타낸 바와 같이 데이터에서 20h 또는 41h를 빼서 데이터를 00h~BDh로 매핑시킨 후 매핑된 데이터와 키 값을 더한 후 190으로 모듈로 연산을 한다. 190으로 모듈로 연

산을 행하는 이유는 나타날 수 있는 문자의 종류가 20h~7Eh 및 A0h~FEh로 190가지이기 때문이다. 모듈로 연산을 수행하고 나면 이 값은 00h~BDh 사이의 값을 가지는데 이 값에 20h 또는 41h를 더하면 20h~7Eh와 A0h~FEh의 값으로 변화되어 모의 제어 문자가 없는 데이터가 된다. 복호화시에는 (그림 6)의 (b)에 나타낸바와 같이 암호화와 반대의 과정을 행하면 된다. 암호화된 값에서 20h 또는 41h를 빼고 이 값과 키값을 비교하여 암호화된 값이 작으면 190을 더한다. 이 값에서 키값을 빼서 다시 20h 또는 41h를 더하면 원시 데이터가 구해진다. 여기서 20h을 빼고 더하는 것은 암호복화가 명백하게 이루어짐을 보이기 위해 편의상 삽입한 것이며 이것이 오버헤드로 작용할 경우는 더 간단한 방법을 사용할 수도 있다. 이 방법을 이용하여 암호복화가 명백하게 이루어짐을 보이기 위해서 다음 식을 참조한다.

$$\text{Encrypted_Data} = \text{remap}((\text{map}(\text{Source_Data}) + \text{Key}) \bmod 190) \quad (1)$$

$$\text{Source_Data} = \text{remap}(\text{map}(\text{Encrypted_Data}) + (\text{Count} * 190) - \text{Key}) \quad (2)$$

각 식에서 함수 'map'은 (그림 6)에 나타낸 바와 같이 각 입력 문자를 비교하여 20h 또는 41h를 빼주는 기능을 수행하며, 함수 'remap'은 20h 또는 41h를 더해주는 기능을 수행한다. 식 (2)에서 'Encrypted_Data'가 'Key'보다 작으면 'Count'를 증가시키면서 비교한

후 계산하며 이 때 'Count'는 0, 1 또는 2이다. 식 (1)에서 구해진 'Encrypted_Data'는 일정한 'Source_Data'와 'Key'에 대해 유일하며 식 (2)에서 190개의 값 중 하나를 갖는 'Source_Data'는 정해진 'Encrypted_Data'와 'Key'에 대해 유일하다. 그러므로 이 식을 이용한 암호화 및 복호화는 정상적으로 이루어진다.

3.2.2 키 수열의 도메인을 190개의 원소로 구성하는 방법

앞 소절에서 언급한 방법은 데이터의 안전성면에 서 좋은 방법이라 할 수 있으며 3절에서 소개한 방법에 비해 계산 복잡도도 좋으며 암호화된 데이터의 임의성도 3.2절의 방법과 비슷하다. 다만 암호화된 데이터가 키수열 만큼의 임의성을 갖지는 않는다. 본 소절에서는 키 수열의 도메인을 00h~DFh로 하여 생성된 키와 4.1절과 비슷한 절차를 이용하여 암호화를 수행하므로써 암호화된 데이터의 임의성을 극대화하는 방법을 소개한다.

암호화시에는 4.1절의 방법과 같으며 복호화시 식 (2)의 Count가 0 또는 1이라는 점만 다르다. 확률적으로 계산해보면 식 (1)과 식 (2)의 Source_Data 및 Key가 0~189이므로 두 데이터의 합은 0~378이므로 이를 190으로 모듈로 연산을 취하면 0~189에 해당되는 값이 모두 두 개씩이다. 즉, 0, 190은 0로 1, 191은 1로 그리고 189, 378은 189로 매핑되므로 암호화된 데이터는 키만큼의 임의성을 갖는다고 볼 수 있다. (그림 7)에 0~189 사이의 값을 갖는 키를 이용한 암호화 및

| | |
|--|---|
| <pre> while(1) { Receive_char(ch); if (ch < 0x80) ch = ch - 0x20; else ch = ch - 0x41; Getkey(key); /* 0x00 ≤ key ≤ 0xff */ ch = (ch + key) mod 190 /* 0xbe */; if (ch < 0x5f) ch = ch + 0x20; else ch = ch + 0x41; Send_char(ch); } </pre> <p>(a) 송신시의 변환방법</p> | <pre> while(1) { Receive_char(ch); if (ch < 0x80) ch = ch - 0x20; else ch = ch - 0x41; Getkey(key); /* 0x00 ≤ key ≤ 0xff */ while(ch < key) ch = ch + 190; ch = ch - key; if (ch < 0x5f) ch = ch + 0x20; else ch = ch + 0x41; Send_char(ch); } </pre> <p>(b) 수신시의 변환방법</p> |
|--|---|

(그림 6) 합과 모듈로 연산을 이용하여 모의 제어문자를 처리하는 방법-(i)
(Fig. 6) Processing control-like characters with modular addition-(i)

| | |
|--|--|
| <pre> while(1) { Receive_char(ch); if (ch < 0x80) ch = ch - 0x20; else ch = ch - 0x41; do { Getkey(key); } while(key >= 190); ch = (ch + key) mod 190; if (ch < 0x5f) ch = ch + 0x20; else ch = ch + 0x41; Send_char(ch); } </pre> <p>(a) 송신시의 변환방법</p> | <pre> while(1) { Receive_char(ch); if (ch < 0x80) ch = ch - 0x20; else ch = ch - 0x41; do { Getkey(key); } while(key >= 190); if (ch < key) ch = ch + 190; ch = ch - key; if (ch < 0x5f) ch = ch + 0x20; else ch = ch + 0x41; ... Send_char(ch); } </pre> <p>(b) 수신시의 변환방법</p> |
|--|--|

(그림 7) 합과 모듈로 연산을 이용하여 모의 제어문자를 처리하는 방법-(ii)
 (Fig. 7) Processing control-like characters with modular addition-(ii)

복호화 방법을 나타내었다. 키 생성시에 키값의 범위를 정할 수도 있으나 0~255의 범위를 갖는 키생성 알고리즘을 사용한 경우로 가정하고 기술하였다. 0~255의 범위를 갖는 키 수열이 충분히 임의성을 갖는다면 이 수열에서 190 이상의 값을 제외한 수열도 충분히 임의성을 갖기 때문이다.

4. 실험 및 검증

이상 3절에서 모의 제어문자가 발생하지 않도록 암호화하는 방법을 살펴보았다. 본 절에서는 앞에서 언급한 방법들을 이용한 암호 통신시 데이터의 전송속도를 측정하여 제시하고 이런 방법들을 이용하여 암호화된 데이터가 약해지지 않았음을 보이기 위해 암호화된 데이터의 임의성을 측정하여 나타내었다. 각 실험에서 사용한 텍스트는 1Mbyte의 C 프로그램 소스이다.

4.1 각 방법을 이용한 경우의 통신 속도

(그림 1)의 환경에서 80Kbyte의 텍스트 데이터를 9600bps 및 19200bps에서 송신시험을 하였다. 이 시험에서 양쪽 단말인 (그림 1)의 (a)와 (d)는 486DX2급을 사용하였으며 통신 프로그램은 'procomm'을 사용하였다. 통신 프로토콜은 kermite를 사용하였으며 8비트 데이터를 사용하였다. 암호화는 스트림 암호화 방법을 이용하였으며, 데이터 변환부는 256byte 크기의 8개의 버퍼를 사용하였고 버퍼가 차거나(bufferfull) 데이터 전진문자(data forwarding character)가 입력되었을 때 암호화하여 송신하는 방법을 사용하였다. 또, 데이터 변환부의 주 프로세서는 MC68000을 사용하였다. <표 1>에 나타낸 바와 같이 Kermit의 방법은 엄청난 속도저하를 초래했다. 이론적으로 데이터 변환부에서 처리없이 통과시킨 경우가 45초였다면 kermite 방법을 사용하면 1/4 정도가 늘어난 56초 정도에 전송이 완료되어야 하나 송신측의 데이터 변환부와 망

<표 1> 모의 제어문자 처리방법에 따른 통신속도 비교
 (Table 1) Comparison of communication speeds depending on each method

| 암호문/평문 전송률 | 암 호 문 | | | | 평문 (처리 없이 통과) |
|---------------|---------------|---------------|------------------|-------------------|------------------|
| | XOR 한 번 취합 | XOR 두 번 취합 | 합과 모듈로 연산을 이용 | Kermit을 이용한 경우 | |
| 19200bps | 45s | 48s | 45s | 62s | 45s |
| 9600bps | 85s | 86s | 85s | 114s | 85s |

사이인 (그림 1)의 구간 (ii)에서 병목현상이 일어나 많은 흐름제어를 야기하였고 이 때문에 통신속도는 더 떨어졌다. 본 논문에서 제시한 방법들은 통신속도 면에서 암호화를 행하지 않는 방법과 비슷하였다. 단, 이 실험은 서로 다른 키 수열로 같은 데이터를 암호화하여 30회 시험하였으며, 시간 측정은 수동으로 하였기 때문에 소수점 이하는 반올림하여 초 단위로만 기록하였다.

4.2 키 수열 및 키 수열을 이용하여 암호화된 데이터의 임의성

이 실험에서 사용된 키 수열이 충분히 임의적이라는 사실을 보이고 키 수열과 데이터를 단순히 배타적 논리합을 취하여 얻은 결과의 임의성을 보이기 위해 일반적으로 많이 사용되는 평가방법인 frequency test, serial test, poker-8 test 등을 이용하였으며 각 실험의 유의수준은 5%로 하였다.¹⁷⁾¹⁸⁾¹⁹⁾ 이 중 frequency test는 각 비트별로 '0'과 '1'이 얼마나 고르게 나타났는지에 대한 실험이며, serial test는 모든 연속한 두 비트에 대해 가능한 값이 고르게 분포되었는지를 실험하는 것이다. poker-8 test는 8비트 단위로 데이터를 잘라서 가능한 값(0~255)이 고르게 분포되었는 지를 실험하는 것이다. 즉, poker-8 test는 자유도가 255인 chi-square test와 같다. 각 모의 제어문자 처리방법에 의해 생성된 수열의 임의성을 키 수열의 임의성과 비교하기 위해 먼저 키수열에 대해 충분히 임의적이라는 것을 실험한 후, 키 수열과 1메가 바이트의 텍스트 데이터에 대해 배타적 논리합을 취한 값이 충분히 임의적이라는 것을 보였다. 단 임의성 시험 및 비교를 위하여 모의 제어문자 처리는 하지 않았다. 모의 제어문자 처리를 할 경우 20h~7Eh 및 A0h~FEh의 범위에

서만 데이터가 나타나므로 비트나 바이트 단위의 임의성 실험으로는 타당한 값을 얻을 수 없기 때문이다. <표 2>에 각 실험방법을 이용한 키 수열의 임의성 및 모의 제어문자 처리를 하지 않은 암호데이터의 임의성 실험 결과를 나타내었다. 이 실험은 1메가 바이트의 키 수열 및 암호화된 데이터를 100바이트 단위 및 1000바이트 단위로 잘라서 각각 10000회 및 1000회의 실험을 하였고 1메가 바이트 길이의 데이터에 대해 한 번의 실험을 수행하였다. 이 실험에서 유의수준이 5%였으므로 95% 정도가 성공하여야 임의성이 좋은 수열이라 할 수 있으며 일반적으로 90%이상의 성공확률을 보이면 임의성이 좋은 수열이라 할 수 있다.¹⁶⁾¹⁷⁾¹⁸⁾ 이 결과로 알 수 있듯이 키 수열과 암호화된 데이터는 충분히 임의성을 갖고 있다.

4.3 니블 단위의 배타적 논리합을 이용하여 암호화된 데이터의 임의성

본 소절에서는 3절에서 언급한 암호화 방법을 이용하여 생성된 데이터의 임의성을 실험하고 그 결과를 제시한다. 3절에서 언급한 방법으로 생성된 데이터는 20h~7Eh 및 A0h~FEh 사이의 값이므로 비트 또는 바이트 단위의 임의성 실험은 의미가 없다. 5.2절의 실험 방법 대신 자유도(degrees of freedom)를 189로 하여 chi-square 실험을 하므로써 임의성을 측정하였다.¹⁸⁾ 식 (3)¹⁸⁾에 의하여 chi-square V 값이 구해지는데 식 (3)에서 k는 샘플의 종류이며 Y_s는 s 번째 샘플이 나타난 횟수이고, np_s는 s 번째 샘플이 나타날 기대치이다. 이 실험에서 k는 190이 된다.(실제로는 s=0 to 189 이다.)

$$V = \sum_{s=1}^k \frac{(Y_s - np_s)^2}{np_s} \tag{3}$$

<표 2> 키와 암호화된 데이터에 대한 임의성 시험
(Table 2) Randomness test for key and cipher data

| 시험 방법 | Frequency Test | | | Serial Test | | | Poker-8 Test | | |
|-------------|-----------------------|-----------------------|--------------------|-----------------------|-----------------------|--------------------|-----------------------|-----------------------|--------------------|
| | 100바이트 × 10000회 | 1000바이트 × 1000회 | 1메가 바이트 × 1회 | 100바이트 × 10000회 | 1000바이트 × 1000회 | 1메가 바이트 × 1회 | 100바이트 × 10000회 | 1000바이트 × 1000회 | 1메가 바이트 × 1회 |
| 키 수열 | 95.4% | 96.5% | 성공 | 95.2% | 94.0% | 성공 | 94.4% | 94.5% | 성공 |
| 암호화된 데이터 | 94.6% | 95.1% | 성공 | 94.0% | 91.7% | 성공 | 93.0% | 90.6% | 성공 |

식 (3)에서 구해진 값이 어느 정도이면 합당한지는 자유도 v 를 이용하여 chi-square 분포표에서 자유도 v 를 이용하여 기준치를 구하므로써 알 수 있으며 자유도 v 는 $k-1$ 로 정해진다. 이 실험에서 유의수준은 5%로 하였고 $v=189$ 로 하였다. <표 3>에 나타낸 바와 같이 니블 단위의 배타적 논리합을 한 번 취한 방법은 100바이트 단위로 임의성 검증을 한 경우에만 통과하였고 긴 수열에 대해서는 임의성이 부족한 것으로 나타났다. 반면 니블 단위의 배타적 논리합을 두 번 취한 방법에서는 100바이트는 물론 1000바이트 단위의 검증에서도 임의성이 충분한 것으로 나타났다.

4.4 합과 모듈로 연산을 이용하여 암호화된 데이터의 임의성

본 소절에서는 4절에서 언급한 암호화 방법을 이용하여 생성된 데이터의 임의성을 실험하고 그 결과를 제시한다. 4절에서 제시한 방법으로 생성된 데이터 역시 20h~7Fh 및 A0h~FEh 사이의 값이므로 비트 또는 바이트 단위의 임의성 실험은 의미가 없다. 그래서 니블 단위의 배타적 논리합을 이용한 방법과 마찬가지로 자유도를 189로 하여 chi-square 실험을 하므로써 임의성을 측정하였다. <표 4>에 나타낸 바와 같이 키의 범위가 0~255인 경우는 100바이트 단위의 검증에서는 충분히 임의성을 갖는다고 볼 수 있으나 1000바이트 및 1Mbyte의 수열에 대한 임의성 검증에서는 실패하였다. 키의 범위가 0~189인 경우는 모든 임의성 검증을 통과하여 키 수열과 임의성 면에서는

대등함을 알 수 있다.

이상의 실험 결과를 살펴보면 <표 2>와 <표 3>에서 알 수 있듯이 각 변환 방법 중 니블 단위의 배타적 논리합을 한 번 이용한 방법과 합과 모듈로 연산을 이용한 방법 중 키 수열의 도메인을 256개로 구성한 경우는 100바이트 단위의 실험에서만 성공률이 90%를 상회하였고 1000바이트 및 그 이상의 길이의 실험에서는 성공률이 낮았다. 즉, 이 두가지 방법을 이용하면 데이터를 100바이트 이하의 단위로 전송하는 경우가 대부분인 대화형 통신에는 적합하나 규모가 큰 화일 전송 등에는 적합하지 않다고 할 수 있다.

반면 니블 단위의 배타적 논리합을 두 번 이용한 방법과 합과 모듈로 연산을 이용한 방법 중 키 수열의 도메인을 190개로 구성한 경우는 100바이트 및 1000바이트 단위의 실험은 물론 1메가 바이트의 데이터에 대해서도 95%에 가까운 성공률을 보여 앞의 두 방법에 비해 안전하다고 할 수 있다. 특히 이 두가지 방법 중에서도 키 도메인을 190개로 하는 합과 모듈로 연산을 이용한 방법이 시간적인 오버헤드가 없어 가장 우수한 방법이라 하겠다.

5. 결론

비동기 통신시 전송되는 이진 데이터가 제어문자와 같은 코드값을 가질 수 있는데 이를 모의 제어문자라 한다. 프로토콜이 제대로 수행되기 위해서는 이런 모의 제어문자를 다른 문자로 변환하여 송신하여

<표 3> 배타적 논리합을 이용하여 생성된 데이터의 임의성 시험
<Table 3> Randomness test for data produced by XORing Method

| 변환 방법 | 시험 횟수 | 100바이트 × 10000회 | 1000바이트 × 1000회 | 1메가 바이트 × 1회 |
|-------------|-------|-----------------|-----------------|--------------|
| XOR를 한 번 취함 | | 94.6% | 50.5% | 실패 |
| XOR를 두 번 취함 | | 94.2% | 92.0% | 실패 |

<표 4> 합과 모듈로 연산을 이용하여 생성된 데이터에 대한 임의성 시험
<Table 4> Randomness test for data produced by Modular addition

| 키의 도메인 | 시험 횟수 | 100바이트 × 10000회 | 1000바이트 × 1000회 | 1메가 바이트 × 1회 |
|--------|-------|-----------------|-----------------|--------------|
| 0~255 | | 90.5% | 48.7% | 실패 |
| 0~189 | | 94.2% | 95.5% | 성공 |

야 한다. 일반적으로 모의 제어문자 및 '#'에 대해 40h과 배타적 논리합을 취한 다음 '#'과 함께 송신하는 방법을 많이 사용한다. 이 방법은 Kermit 프로토콜에 정의된 방법인데 데이터가 암호화되어 송신될 경우 난수에 가까워져 추가되는 '#'의 갯수도 많아지며 전송하는데 시간이 많이 걸릴 뿐만 아니라 그에 따른 흐름제어 횟수도 많아지게 되어 통신 효율을 더욱 떨어뜨리게 된다. 본 논문에서는 Kermit에서 제공하는 모의 제어문자 처리 방법을 소개하고 분석하였으며 암호통신시 발생하는 문제점을 분석하였다. 이에 대한 해결책으로 키 수열과 데이터의 합성 방법을 크게 두가지 제시하여 추가되는 문자 없이 모의 제어문자를 처리하여 암호통신시 송신효율을 높이는 방안을 제시하였다. 이 방법들은 단말에서 데이터 변환부로 보내지는 데이터는 모의 제어문자를 갖지 않는다는 특성을 이용하였다.

첫 번째 방법은 니블 단위로 키 수열과 배타적 논리합 연산을 하는 방법으로 배타적 논리합 연산을 행한 값이 모의 제어문자이면 상위니블은 원시 데이터 값을 취하는 방법이다. 이 방법은 Kermit에 의한 방법 보다 몇가지 절차를 더 수행하나 프로세서의 속도가 충분히 빠르기 때문에 통신 속도에 지장을 주지는 않았다. 오히려 일정한 통신속도에서 데이터의 추가 없이 통신하므로 1/4 정도의 데이터 추가를 요하는 Kermit에 의한 방법에 비해 전체적인 전송속도는 훨씬 빨라진다는 사실을 실험을 통하여 확인하였다. 이 방법으로 구현된 데이터에 임의성이 취약하여 이 방법을 한 데이터에 두 번 적용하여 해결하였다. 니블 단위로 배타적 논리합을 두 번 취한 경우는 한 번 취한 경우에 비해 상대적으로 충분한 임의성을 가짐을 알 수 있었으며 연산이 많아지는데 따른 시간적 손실은 크지 않았다.

두 번째 방법은 합과 모듈로 연산을 이용한 방법으로 원시 데이터가 모의 제어문자를 갖지 않는다는 특성을 이용하여 이 값들을 0~189으로 매핑한 후 190으로 모듈로 연산을 취하여 암호화하였다. 이 방법은 첫 번째 방법과 비교해서 시간적인 측면에서도 뒤지지 않으며 암호화된 데이터의 안전성 면에서도 우수하다고 할 수 있다. 특히 키의 범위를 0~189로 하였을 때는 키수열과 대동한 임의성을 가짐을 알 수 있었다.

이상과 같이 제시된 방법들이 데이터의 안전성에 는 지장을 주지 않으면서 통신 효율을 좋게 하였다. 그러나 이런 방법을 이용하여 암호화하였을 때 정상적인 방법에 비해 공격에 약해졌는지에 대한 검증은 하기가 어렵기 때문에 단지 눈에 드러나게 약해지지 않은 것만 확인하였으며 수열의 임의성을 측정함으로써 안전하다는 것을 보였다.

참 고 문 헌

- [1] F. d. Cruz, 'Kermit Protocol Manual', Columbia University Center, pp. 11-30, 1984. 4.
- [2] U. Black, 'Data Link Protocols', PTR Prentice-Hall, pp. 84-85, 1993.
- [3] 정영일, 'PC 통신 입문 I', 영진출판사, pp. 43-85, 1988. 8.
- [4] H. Gohring, E. Jasper, 'PC-Host Communications-Strategies for Implementation', Addison-Wesley, pp. 1-35, 1993.
- [5] 한국전자통신연구소, '현대암호학', pp. 1-102, 1991.
- [6] R. A. Rueppel, 'Contemporary Cryptology: The Science of Information Integrity', IEEE Press, pp. 65-134, 1992.
- [7] H. Beker and F. Piper, 'Cipher Systems-The Protection of Communications', John Wiley & Sons Inc., New York, 1982.
- [8] D. E. Knuth, 'The Art of Computer Programming-Seminumerical Algorithms', Addison-Wesley, vol. 2, Second Edition, pp. 38-113, 1981.
- [9] I. J. Good, "On the Serial Test for Random Sequences", Ann. Math. Statist., vol. 28, pp. 262-264, 1957.



정 현 철

1989년 계명대학교 전자계산학과 졸업(공학사)
 1991년 경북대학교 대학원 컴퓨터공학과 졸업(공학석사)
 1991년 2월~현재 한국전자통신연구소 연구원

관심분야: 통신정보보호, 계산이론