ISUEA1과 ISUEA2의 암호화 방식 분석

Analysis of the encryption method between ISUEA1 and ISUEA2

수록처 「이 연구가 대단하다!」, vol.1, no.1, 2015.4., p. 5~21

작성자 이도현
연세대학교 컴퓨터과학과 소속
waylight3@naver.com
010-2001-4214

참고 본 논문은 「이 연구가 대단하다!」 2015년 4월호에 수록되어 있습니다.

목 차

ISUEA1과 ISUEA2의 암호화 방식 분석

Analysis of the encryption method between ISUEA1 and ISUEA2

이도현
Lee Do-Hyeon

초 록

ISUEA1과 ISUEA2는 국제공부연맹에서 개발한 두 가지 암호화 방식으로, 문자열을 블록화 하여 암호문을 생성하는 점이 비슷하지만 전자의 경우에는 블록에서 나온 대푯값을 그대로 이어붙이는 방식이라면, 후자의 경우에는 블록간의 연산을 통해 결과 문자열을 생성하는 점이 다르다.

두 암호화 방식의 과정과 특징을 자세히 살펴보고 그 결과와 효용성에 대해 분석해본다. 그리고 개선 방안과 향후 개발될 ISUEA3에 대해 조언을 하며 논의를 마친다.

키워드

ISUEA, ISUEA1, ISUEA2, 암호화, 암호화 방식, 국제공부연맹

Abstract

ISUEA1 and ISUEA2 is a two ciphers international search Federation developed, but that it generates a passphrase by blocking the string similar, the former case, as it continues to Depyotogapu exiting block If a method of attaching, in the latter case, that it produces a result string by using the operations between blocks is different.

And a closer look at the process and characteristics of both the encryption scheme, and try to analyze the results and utility. And improvement plan and finish the discussion by advice on ISUEA3 to be developed in the future.

Keywords

ISUEA, ISUEA1, ISUEA2, encryption, encryption method, International Study Union

I. 서론

ISUEA는 국제공부연맹에서 개발한 국제공부연맹 암호화 알고리즘(International Study Union Encryption Algorithm) 시리즈로 2015년 4월 현재까지 ISUEA1과 ISUEA2의 두 가지 방식이 개발 완료 상태이다. 기본적으로 ISUEA 시리즈는 블록 암호화 방식을 사용하는데, ISUEA1은 주어진 문자열을 블록으로 나눈 뒤 각각의 블록에서 대푯값을 계산한 뒤 이들을 이어 붙여 결과 문자열을 만드는 반면, ISUEA2는 블록 간의 연산이 추가되어 ISUEA1보다 복잡한 방식의 암호화를 진행한다. 그리고 이들은 모두 단방향 암호화 방식이기 때문에 복호화는 거의 불가능하다고 알려져 있다. 암호문의 길이는 블록의 개수를 n개로 설정했을 때 0부터 255까지의 수가 16진법으로 n개 생성되므로 $2n$이 되며, 가능한 경우의 수는 2^{64}가지이다.

ISUEA1과 ISUEA2는 이와 같이 비슷한 성격을 지니고 있으며 암호화 결과가 다르다는 것과 조금 더 복잡하다는 것 외에는 별다른 차이점이 없다고 여겨진다. 이를 깊이 있게 분석하고자 본 논문에서는 두 방식의 암호화 과정을 자세히 서술하고 특징을 분석한 뒤 결과에 대한 논의까지 덧붙인다. 그리고 이러한 분석을 바탕으로 정리된 앞으로의 개선 방안을 제기하고 향후 개발될 ISUEA3의 개발에 도움이 되고자 한다. 블록화를 어떻게 사용하는 것이 효율적인지, 암호화 결과가 만족해야 할 조건들은 무엇인지, 암호화 속도는 어느 정도 차이가 나는지 등 다양한 논의들을 통해 이를 확인할 수 있을 것이다.

II. ISUEA1

ISUEA1은 ISUEA 시리즈의 첫 번째로 주어진 문자열을 n개의 블록으로 자르고, 각블록의 대푯값을 구하여 이들을 이어 붙여 암호문을 생성한다. ISUEA1의 암호화 방식과 결과 그리고 이에 대한 분석을 제시한다.

1. 암호화 방식

먼저 주어진 문자열(Str)과 블록의 개수(BLOCK_NUMBER)를 입력받는다. 이때 Str은 String형 변수로 선언한다. 그 이후에는 다음과 과정을 거쳐 암호문을 생성한다. 각 단계에 대한 자세한 설명은 소스코드를 기재하는 것으로 대신한다.

1. 각 문자의 총 합(totalSum)을 구한 다음 이를 이용해 문자열 내의 임의의 글자를 선택한다. 구체적으로는 문자열 내의 모든 글자를 숫자로 변환하여 더한 총합을 문자열의 길이로 나눈 나머지를 선택한다.

```
// 문장 각 글자(char)의 총 합을 이용해 문장 중 임의의 글자 선택
for (int i = 0; i < str.Length; i++)
{
    totalSum += (int)str[i];
    totalSum %= str.Length;
}
```

[그림 1] 문자열의 총 합 구하기

2. 난수 생성 함수의 시드를 선택한다. 만약 입력이 없는 경우(Str == "")에는 (입력 문자열의 길이) + (블록 개수 * 1023)을 선택하고, 그 외의 경우에는 (입력 문자열의 길이) * (문자열 내의 totalSum + 1번째 문자)를 선택한다.

```
// 난수 생성 함수의 시드를 결정
if (str == "")
{
    rand = new Random(str.Length + BLOCK_NUMBER + 1023);
}
else
{
    rand = new Random(str.Length + BLOCK_NUMBER + str[totalSum]);
}
```

[그림 2] 난수 생성 함수의 시드 결정

3. 원래 문장과 생성한 난수를 뒤섞는다. 이때 블록 수보다 작으면 더 길게 넣는다. 구체적으로는 [그림 3] 참고.

```
// 원래 문장과 생성한 난수를 뒤섞음. 이때 블럭 수 보다 작으면 더 길게 넣음
String tempStr = "";

if (str.Length < BLOCK_NUMBER)
{
    for (int i = 0; i < str.Length; i++)
    {
        if (i % 2 == 0)
        {
            tempStr += rand.Next(10140585, 294076965) + str[i];
        }
        else
        {
            tempStr += str[i];
        }
    }
}
else
{
    for (int i = 0; i < str.Length; i++)
    {
        if (i % 2 == 0)
        {
            tempStr += rand.Next(1024, 65536) + str[i];
        }
        else
        {
            tempStr += str[i];
        }
    }
}

str = tempStr;
```

[그림 3] 문장 뒤섞기

4. 문장 길이를 블록 수의 배수가 되도록 설정한다. 글자 수가 부족한 경우 0부터 9까지의 자연수 중 하나를 삽입해가며 글자 수를 맞춘다.

```
// 문장의 길이를 블럭 수의 배수가 되도록 설정
for (int i = 0; str.Length % BLOCK_NUMBER != 0; i++)
{
    str += rand.Next(0,10).ToString();
}
```

[그림 4] 글자 수 맞추기

5. 각블록마다 연산을 시행한다. 구체적으로는 [그림 5] 참고.

```
// 블럭을 블럭 수 만큼 나누어 각 블럭마다 연산 시행
char[,] g = new char[BLOCK_NUMBER + 1, str.Length / BLOCK_NUMBER + 1];


for (int i = 0; i < str.Length; i++)
{
    g[i % BLOCK_NUMBER + 1, (int)(i / BLOCK_NUMBER + 1)] = str[i];
}

int[] sum = new int[BLOCK_NUMBER + 1];

for (int i = 1; i <= BLOCK_NUMBER; i++)
{
    sum[i] = 0;
}

for (int i = 1; i <= str.Length / BLOCK_NUMBER; i++)
{
    for (int j = 1; j <= BLOCK_NUMBER; j++)
    {
        if (i % 2 == 0)
        {
            sum[j] -= g[j, i];
        }
        else
        {
            sum[j] += g[j, i];
        }

    }
}
```

[그림 5] 블록별 연산 시행

6. 블록마다 나온 연산 값(대푯값)을 이용해 암호문을 생성한다. 구체적으로는 [그림 6] 참고.

```
// 블럭 마다 나온 연산 값을 이용해 암호문 생성
for (int i = 1; i <= BLOCK_NUMBER; i++)
{
    rand = new Random(sum[i]);
    result += Math.Abs((((sum[i] + rand.Next()) * i * i) % 256)).ToString("x2");
}
```

[그림 6] 암호문 생성

2. 암호화 결과

규칙을 바꾸어 가며 입력한 원문의 암호문은 다음과 같다.

입력	블록 수	암호문
0	16	52b824900290e2400ef86c006d38ee00
1	16	cc344d701480b20037fc6cd022a41300
2	16	7ea403403b482740a09072505454ee00
3	16	cc8003300248f4c09d8c72106d807c00
4	16	d7804d20023460c0e5b0a5405ca4ee00
5	16	ccd02450642cc480e5ec1cc0619cf700
6	16	046ca0403b80eb0037c883101d30ee00
7	16	cc6cd4201488e200b5c8b1305488b500
8	16	a0346e4014342780f990a5401d546000
9	16	ccb81e40b22ceb00a0b02d00a0a4be00

[표 1] 한 자리 수 입력

입력	블록 수	암호문
0	16	52b824900290e2400ef86c006d38ee00
00	16	a0b80390b2fcbb809dec6ce02ea48500
000	16	9a80a0b0ec70978061f8a4e09f80ed00
0000	16	a014b3e029d4f44017c4f770e45c2600
00000	16	1968fd70bd70350046d889a047f0d900
000000	16	44b87f008ab8a9004628a4a0e304ed00
0000000	16	25283f508a48f4c0a0801ce054887c00
00000000	16	52545ed0ffeceb00b544b1109c581100
000000000	16	4710a07064dca980a0ec60102580ee00
0000000000	16	95a4a050eb0cdf4077b060f06d304c00

[표 2] 0 개수 증가하며 입력

입력	블록 수	암호문
100000000000	16	471cb3e0f5840e001b149d304710d900
010000000000	16	45547fb029a849008cf89d702a5cbc00
001000000000	16	1a1cb3e0f5840e001b149d304710d900
000100000000	16	47d0b3e0f5840e001b149d304710d900
000010000000	16	471cb3e0f5840e001b1489304710d900
000001000000	16	471cb3e0f5840e001b149d104710d900
000000100000	16	471cb3e0f1840e001b149d304710d900
000000010000	16	471cb3e0f59c0e001b149d304710d900
000000001000	16	471cb3e0f5840e001b149d3047107b00
000000000100	16	471cb3e0f5840e001b149d304710d900
000000000010	16	471cb3e0f5840e401b149d304710d900
000000000001	16	471cb3e0f5840e0022149d304710d900

[표 3] 1 위치 바꾸어 가며 입력

입력	블록 수	암호문
000000	1	25
000000	2	528c
000000	3	990845
000000	4	f9d86e40
000000	5	52f00b20f6
000000	6	25806e40c22c
000000	7	1a089a20a0e4f1
000000	8	f91c24105ca81540
000000	16	44b87f008ab8a9004628a4a0e304ed00
000000	32	25348f40b2b0eb809d342de0229cf700608c60e0e4b075c03d70c1d0c3b8cc00

[표 4] 블록 수 바꾸어 가며 입력

입력	블록 수	암호문
1	16	cc344d701480b20037fc6cd022a41300
3	16	cc8003300248f4c09d8c72106d807c00
5	16	ccd02450642cc480e5ec1cc0619cf700
7	16	cc6cd4201488e200b5c8b1305488b500
9	16	ccb81e40b22ceb00a0b02d00a0a4be00
11	16	ccd0d4e01490a9c0a0f8b1101da46000
13	16	ccd0d4e01490a9c037f8b1101da46000
15	16	ccd0d4e01490a9c00ef8b1101da46000
17	16	ccd0d4e01490a9c0e5f8b1101da46000
19	16	ccd0d4e01490a9c044f8b1101da46000

[표 5] 홀수 입력

3. 결과 분석

[표 1]에서 나타나듯이 0부터 9까지의 수를 입력하였을 때 모두 00으로 끝났으며, 특히 홀수인 경우 cc로 시작하며 대부분이 비슷하다는 규칙성을 발견할 수 있다. 실제로 홀수들만을 모아보면 [표 5]와 같이 유사한 구간을 많이 찾을 수 있다. 11~19를 입력했을 때에는 중간 부분 극히 일부를 제외하고는 거의 같은 것을 볼 수 있다. 이러한 특징은 특히 입력한 문자열이 짧을수록 두드러지게 일어나는데, 이는 totalSum을 이용해 문자열 중 한 글자를 골라낼 때 가능한 경우가 제한적이므로 랜덤 함수의 시드값이 유사하게 설정되며, 블록 수가 입력 문자열의 길이에 비해 길기 때문에 결국 지나치게 짧은 입력이 원인이라고 생각할 수 있다. 실제로 블록 수에 비해 긴 숫자열을 입력하면 홀수 사이에서 나타나는 이러한 특성이 대부분 사라진다. 한글이나 영어와 같은 문자를 입력하는 대부분의 경우 이러한 특징이 잘 나타나지 않지만 학번이나 주민등록번호와 같이 숫자열을 다루는 경우가 분명히 존재하므로 이는 개선되어야 할 점이다.

[표 2]에서 나타나듯이 단순히 0의 개수를 증가시키는 것만으로는 ISUEA1의 취약점이 드러나지 않았다. 그러나 [표 3]에서와 같이 1의 위치만을 바꾸어 가며 입력한 경우 거의 유사한 암호문이 계속해서 나타나며 3번째 자리에 1이 들어갈 때만 다른 것을 볼 수 있다. 이러한 입력은 문자열의 총합과 길이가 같기 때문에 totalSum에 의해 선택되는 문자 하나가 동일하다는 것을 알 수 있다. 3번째 자리에 1이 들어가는 때에 결과가 달라지는 것을 통해 길이 12인 문자열에서 totalSum이 1인 경우 3번째 문자가 선택될 것이라고 유추할 수 있고 실제로도 그렇다. totalSum과 문자열의 길이에 변화를 줄 수 있는 개선 방안이 필요하다.

[표 4]에서 보듯이 단순히 블록 수를 증가시키는 경우에는 원문이 동일하더라도 암호문이 다르게 나타나는 것을 볼 수 있다. 따라서 ISUEA1은 블록 수에 영향을 많이 받으며 취약점 또한 각각의 블록 수에 따라 다르게 나타날 수 있다는 것을 의미한다. 이러한 특징을 이용할 수 있는 방안을 생각해보는 것도 좋을 것이다.

III. ISUEA2

ISUEA2은 ISUEA 시리즈의 두 번째로 주어진 문자열을 n개의 블록으로 자르고, 각블록의 대푯값을 구하여 이들 사이의 연산을 통해 암호문을 생성한다. ISUEA2의 암호화 방식과 결과 그리고 이에 대한 분석을 제시한다.

1. 암호화 방식

먼저 주어진 문자열(Str)과 블록의 개수(BLOCK_NUMBER)를 입력받는다. 이때 Str은 String형 변수로 선언한다. 그 이후에는 다음과 과정을 거쳐 암호문을 생성한다. 각 단계에 대한 자세한 설명은 소스코드를 기재하는 것으로 대신한다.

1. 문장의 각 글자의 총합을 이용해 문장 중 임의의 글자를 선택한다. 구체적으로는 총합을 문자열의 길이로 나눈 값을 totalSum에 저장한다.

```
// 문장 각 글자(char)의 총 합을 이용해 문장 중 임의의 글자 선택
for (int i = 0; i < str.Length; i++)
{
    totalSum += (int)str[i];
    totalSum %= str.Length;
}
```

[그림 7] 문자열의 총 합 구하기

2. 난수 생성 함수의 시드를 결정한다. 이때 입력이 없을 경우(Str == "")와 그렇지 않은 경우를 분리하며 구체적으로는 [그림 8] 참고.

```
// 난수 생성 함수의 시드를 결정
if (str == "")
{
    rand = new Random(str.Length + BLOCK_NUMBER * 1023);
}
else
{
    rand = new Random(str.Length * BLOCK_NUMBER + str[totalSum]);
}
```

[그림 8] 난수 생성 함수의 시드 결정

3. 원래 문장과 생성한 난수를 뒤섞는다. 이때 블록 수보다 어느 정도로 짧은지에 따라 다르게 적용한다. 만약 문자열의 길이가 6보다 작다면 한글 1글자, 영어 대문자 1글자, 영어 소문자 1글자를 추가한다. 만약 문자열의 길이가 블록 수의 절반보다 작다면 각각 5글자씩 넣고, 이보다는 크지만 블록 수보다 작다면 각각 3글자씩 넣는다. 만약 블록 수보다 많거나 같다면 각각 1글자씩만 넣는다.

```
// 원래 문장과 생성한 난수를 뒤섞음. 이때 블럭 수 보다 작으면 더 길게 넣음
String tempStr = "";

if (str.Length < 6)
{
    for (int i = 0; i < 6; i++)
    {
        tempStr += ((char)rand.Next('가', '힣')).ToString() + ((char)rand.Next('A', 'Z')).ToString() +
            ((char)rand.Next('a', 'z')).ToString();
    }
}
else if (str.Length < BLOCK_NUMBER / 2)
{
    for (int i = 0; i < str.Length; i++)
    {
        switch (i % 3)
        {
            case 0:
            tempStr += ((char)rand.Next('가', '힣')).ToString() + ((char)rand.Next('가', '힣')).ToString() +
                    ((char)rand.Next('가', '힣')).ToString() + ((char)rand.Next('가', '힣')).ToString() +
                    ((char)rand.Next('가', '힣')).ToString() + str[i];
                break;
            case 1:
            tempStr += ((char)rand.Next('A', 'Z')).ToString() + ((char)rand.Next('A', 'Z')).ToString() +
                    ((char)rand.Next('A', 'Z')).ToString()  + ((char)rand.Next('A', 'Z')).ToString() +
                    ((char)rand.Next('A', 'Z')).ToString() + str[i];
                break;
            case 2:
                tempStr += ((char)rand.Next('a', 'z')).ToString() + ((char)rand.Next('a', 'z')).ToString() +
                    ((char)rand.Next('a', 'z')).ToString()  + ((char)rand.Next('a', 'z')).ToString() +
                    ((char)rand.Next('a', 'z')).ToString() + str[i];
                break;
            default:
                break;
        }
    }
}
```

[그림 9] 문장에 난수 뒤섞기 1

```
else if (str.Length < BLOCK_NUMBER)
{
    for (int i = 0; i < str.Length; i++)
    {
        switch (i % 3)
        {
            case 0:
                tempStr += ((char)rand.Next('가', '힣')).ToString() + ((char)rand.Next('가', '힣')).ToString() +
                        ((char)rand.Next('가', '힣')).ToString() + str[i];
                break;
            case 1:
                tempStr += ((char)rand.Next('A', 'Z')).ToString() + ((char)rand.Next('A', 'Z')).ToString() +
                        ((char)rand.Next('A', 'Z')).ToString() + str[i];
                break;
            case 2:
                tempStr += ((char)rand.Next('a', 'z')).ToString() + ((char)rand.Next('a', 'z')).ToString() +
                    ((char)rand.Next('a', 'z')).ToString() + str[i];
                break;
            default:
                break;
        }
    }
}
```

[그림 10] 문장에 난수 뒤섞기 2

```
else
{
    for (int i = 0; i < str.Length; i++)
    {
        switch (i % 3)
        {
            case 0:
                tempStr += ((char)rand.Next('가', '힣')).ToString() + str[i];
                break;
            case 1:
                tempStr += ((char)rand.Next('A', 'Z')).ToString() + str[i];
                break;
            case 2:
                tempStr += ((char)rand.Next('a', 'z')).ToString() + str[i];
                break;
            default:
                break;
        }
    }
}

str = tempStr;
```

[그림 11] 문장에 난수 뒤섞기 3

4. 문장의 길이를 블록 수의 배수가 되도록 한글을 한 글자씩 추가한다.

```
// 문장의 길이를 블럭 수의 배수가 되도록 설정
for (int i = 0; str.Length % BLOCK_NUMBER != 0; i++)
{
    str += ((char)rand.Next('가', '힣')).ToString() + temp;
}
```

[그림 12] 문장의 길이 조정

5. 문자들을 블록에 나누어 넣는다.

```
// 문자들을 블럭에 나누어 넣음
char[,] g = new char[BLOCK_NUMBER + 1, str.Length / BLOCK_NUMBER + 1];

for (int i = 0; i < str.Length; i++)
{
    g[i % BLOCK_NUMBER + 1, (int)(i / BLOCK_NUMBER + 1)] = str[i];
}
```

[그림 13] 문자 나누어 담기

6. 각블록의 대푯값을 구한다. 이때 대푯값이 한글 한 글자가 되도록 한다.

```
// 각 블럭의 대푯값을 구함
for (int i = 1; i <= BLOCK_NUMBER; i++)
{
    g[i, 0] = (char)0;
    for (int k = 1; k <= str.Length / BLOCK_NUMBER; k++)
    {
        g[i, 0] = (char)(Math.Round((double)(g[i, 0] - g[i, k]) * (g[i, 0] - g[i, k]) * k) % '힣' + 1);
    }
}
```

[그림 14] 블록의 대푯값 구하기

7. 블록 간 연산을 시행한다. 이때 랜덤 함수를 이용하여 최대한 선택되는 값들의 연관성을 줄인다.

```
// 블럭 간 연산을 시행
char tempChar;
int a, b;

for (int i = 0; i < BLOCK_NUMBER; i++)
{
    rand = new Random(g[i, 0]);
    a = rand.Next() % BLOCK_NUMBER + 1;
    b = rand.Next() % BLOCK_NUMBER + 1;
    tempChar = (char)((g[a, 0] + g[b, 0]) % '힣' + 1);
    g[a, 0] = (char)(Math.Abs(g[a, 0] - g[b, 0]) % '힣' + 1);
    g[b, 0] = tempChar;
}
```

[그림 15] 블록 간 연산 시행

8. 결과를 16진수로 변환하여 암호문을 생성한다.

```
// 결과를 16진수로 변환
for (int i = 1; i <= BLOCK_NUMBER; i++)
{
    result += (g[i, 0] % 256).ToString("x2");
}
```

[그림 16] 결과를 16진수로 변환

2. 암호화 결과

규칙을 바꾸어 가며 입력한 원문의 암호문은 다음과 같다.

입력	블록 수	암호문
0	16	6c9023215495892d08405bb9b3cb480d
1	16	2c13295750dfb4352934bffc40926ede
2	16	da0c06624b7c8a3e0343402552a78e2f
3	16	c5a5c070f4aebf428bf7e5110fde3235
4	16	43294b46ae7b6c807faf10dd78282a45
5	16	e97e6d40522471a1f8657203f81fec3d
6	16	85f2fad2456c0b626837e8bd03f18bf8
7	16	51319650917b805c613adde66937091d
8	16	366c2088b4549929e7006e9fdde5b742
9	16	986c0933f222c945b43c182b9820d6bb

[표 6] 한 자리 수 입력

입력	블록 수	암호문
0	16	6c9023215495892d08405bb9b3cb480d
00	16	be27658e22b2aaf3e296f43a2cfa9a25
000	16	e8fe3ac7b7e21dbf990c431f1a7c6ff4
0000	16	5ab734f751b545400d3ae68a8e849f2a
00000	16	0ac2cea1e2291b4cba68c286e4b4adf9
000000	16	3c80f66a78860a0a7b1df00a241d330e
0000000	16	ce3aea6c597d7ea3008ef20b107f5bcf
00000000	16	1319127e73e821b24417af5975df60bd
000000000	16	3c391300fd9d88db2f5649e78c5b49f7
0000000000	16	44b0609eba2443b1a67184653fc06e4d

[표 7] 0 개수 증가하며 입력

입력	블록 수	암호문
100000000000	16	a4aca821b737b3fc5ef33db7487deafc
010000000000	16	c3882562bef6ffe535df2ed4127033fc
001000000000	16	a9d8451051813e493cc5f5b91b8a0bba
000100000000	16	4406d6104c810cfc2826a8c7ab5d708d
000010000000	16	a474a8b4b73701fc5edb9c9eabed43bf
000001000000	16	4435a810b7370c360bd39cc729bb0b12
000000100000	16	8fbba83d4d813bfaf2b699aa4013dcfc
000000010000	16	4406a810c5810cfc28d39cc729eb0b32
000000001000	16	e518a81db99716fc83559cfeab0228ac
000000000100	16	b476a810b9370c090bd39cc729bb0b12
000000000010	16	d6cba8904c8b9bfc5b82231174181090
000000000001	16	4406a8104c810cfc28d339c729590b89

[표 8] 1 위치 바꾸어 가며 입력

입력	블록 수	암호문
000000	1	11
000000	2	f793
000000	3	94a742
000000	4	66ea222f
000000	5	d770e8f800
000000	6	b436e5a5761b
000000	7	2912e0fdef528a
000000	8	3ab2b7fc763f1695
000000	16	3c80f66a78860a0a7b1df00a241d330e
000000	32	1901907fca019936589eb3f23bfea45460aaa1189a919dd2e8638cbcb3efb0c0

[표 9] 블록 수 바꾸어 가며 입력

입력	블록 수	암호문
12개의 0	16	4406a8104c810cfc0bd39cc729bb0b12
16개의 0	16	c91bc7814b2bfaa90d1d879d508a2634
18개의 0	16	e8fc2656579cfbd00ec17fc8bdb90d27
24개의 0	16	4dc225fcc7bf962b23fbca71b29efaa1
32개의 0	16	c76d50e21ac2a0df99c24a30df99195d
12개의 0	32	c04dea5acfb8a3e75d3344ba934d21422799ac5fc36c37ca34bccb5a7293f98f
16개의 0	32	49db6fa4a325a35c1c2d1636c610326cd623f6a4a2380922150d8ad3218a419e
18개의 0	32	28dedc979b98c4fc4e1b924330cfc81e734576c6bc4f415723f653b5bee4f5de
24개의 0	32	25b670265c996893f2d7387c393c98db4dff4b526a9fa6d672cc148edb5e8065
32개의 0	32	2f479f7a8388ab1bddc42a0be51b38687888d301c55fb794c51b29b98fa56f7a

[표 10] 0의 개수를 달리하여 입력

3. 결과 분석

[표 6]에서 나타나듯이 한 자리 수를 입력하여도 암호문간의 유사한 구간이 발견되지 않는 것을 알 수 있다. 또한 [표 7]에서 단순한 0의 개수의 증가로도 결과가 크게 달라지는 것을 알 수 있다. 그리고 [표 8]에서 1의 위치를 바꾸는 것은 ISUEA1에서는 취약점이 드러났지만 ISUEA2에서는 암호문의 결과가 크게 달라지는 것을 볼 수 있다. 일부 경우에서 맨 앞에 44가 오는 공통점이 있지만, 이는 원문이 지나치게 단순하여 처음 선택되는 임의의 문자가 제한적일 수 밖에 없음을 고려하면 ISUEA1에 비해 암호화 기능이 매우 향상되었다고 할 수 있다. [표 9]에서와 같이 블록의 개수를 바꾸거나 [표 10]과 같이 0의 개수를 계속해서 증가시키더라도 암호문 사이의 유사성을 발견하기 어렵다. 따라서 ISUEA2는 대부분의 특징에서 ISUEA1보다 적합함을 알 수 있다.

IV. 분석

지금까지의 결과를 바탕으로 ISUEA1과 ISUEA2를 분석한다. 장점과 단점을 정리하고 개선 방안을 제시하여 향후 개발될 ISUEA3에 대해 조언을 하는 것으로 분석을 마친다.

1. 장점과 단점

암호화 속도 측정을 위해 공백 포함 138,872자(공백 제외 109,296자, 한글 92,167자, 32,685어절)를 입력하여 몇 차례 암호화를 시행한 결과, ISUEA1은 평균 17초, ISUEA2는 평균 11초가 걸렸다. 이러한 특징들을 요약하면 다음과 같다.

특징	ISUEA1	ISUEA2
암호화 속도	느림	빠름
홀수 취약점	O	X
단순 반복 취약점	X	X
평행이동 취약점	O	X
블록 수 취약점	X	X

[표 11] ISUEA1과 ISUEA2 비교

2. 개선 방안

ISUEA2의 경우 본 논문에서는 별다른 취약점을 발견하지 못하였다. 그러나 ISUEA1에서는 다수의 취약점(홀수 취약점, 평행이동 취약점)이 발견되었으며, 이를 개선하기 위한 방안을 다음과 같이 생각해볼 수 있다.

첫째, 입력문의 앞과 뒤 또는 중간 중간에 특정 값을 추가한다. 이를 소금(salt)이라고도 하며 사이트의 이름이나 주소, 사용자의 이름이나 전화번호 같은 정보들을 입력문에 추가함으로서 문장을 최대한 복잡하게 구성하려는 노력이다. 이 경우 사용자가 굳이 비밀번호를 복잡하게 입력해야 할 필요가 없으므로 효율적인 방법이다. 나아가, 아이디와 비밀번호를 합쳐서 암호화 하는 방법은 사용자에게 개인정보를 요구할 필요가 없으므로 최선의 방법으로 볼 수 있다.

둘째, 다양한 블록 수를 종합하여 사용한다. 예를 들어 2, 3, 5, 7, 11, 13과 같은 소수 블록 수를 여러 개 사용하여 문자열을 구성한 뒤 이를 다시 원하는 블록 수로 암호화 하는 것이다. 굳이 소수를 예로 든 이유는 블록을 구성할 때 전체 문장 수를 블록 수의 배수로 맞추는 과정을 고려하여 최대한 중복을 피하기 위함이다. ISUEA1에서도 블록 수가 바뀌면 암호문이 크게 달라졌으므로 이를 적절히 이용하면 기존의 구조를 크게 바꾸지 않고도 충분한 개선 효과를 볼 수 있을 것이다.

셋째, 가장 중요한 것은 적절한 블록 수를 사용하는 것이다. 아무리 알고리즘이 효과적이라고 해도 매우 큰 블록 수에 매우 작은 입력문이 주어진다면, 사용할 수 있는 정보가 거의 없으므로 필연적인 반복이 나타나기 마련이다. 위와 같은 개선 방안들은 모두 이러한 반복이 나타나는 경향을 보이는 블록 수의 최댓값을 증가시키기 위한 노력이며 결코 임의의 큰 블록 수에까지 적용되는 것은 아니다.

V. 결론

이와 같이 ISUEA1과 ISUEA2의 암호화 과정과 결과를 분석하였으며, 이를 통해 ISUEA2가 ISUEA1보다 효과적이라는 것을 알 수 있었고 ISUEA1의 개선 방안을 세 가지 제시하였다. 이때 세 번째 제안인 적절한 블록 수의 사용은 ISUEA2를 개발하는 과정에서 중요하게 생각하여 실제로 글자를 추가하는 과정이 복잡하게 구성된 것이다. ISUEA3에서는 이러한 장단점과 개선 방안을 고려하여 더욱 효과적이고 빠른 암호화 방식이 구현되길 희망한다. 그리고 지금까지 ISUEA 시리즈는 단방향 암호화만을 고려했는데, 만약 ISUEA3나 그 이상의 버전에서 복호화가 가능한 양방향 암호화 알고리즘이 개발될 수 있다면 이는 ISUEA 시리즈의 혁신적인 변화를 가져올 것이다. 단순히 암호문의 반복과 같은 취약점뿐만 아니라, 암호문을 복호화 하려는 외부 침입자로부터 정보를 안전하게 보호해야 하기 때문이다.

그리고 암호문의 여러 특징들을 분석하면서 ISUEA 시리즈가 만족해야 할 몇 가지 특성을 다음과 같이 정리할 수 있었다.

1. 같은 원문에 몇 글자를 앞뒤나 중간에 추가할 경우 결과가 매우 달라져야 한다.
2. 같은 원문에 대해 블록 수가 달라지면 결과가 매우 달라져야 한다.
3. 한 두 글자를 치환했을 때 결과가 매우 달라져야 한다.
4. 반복되는 문자가 많더라도 결과에서 반복이 나타나지 않아야 한다.
5. 공백과 같이 매우 적은 입력이 주어지더라도 결과에서 반복이 나타나지 않아야 한다.

참고문헌

- Dream Maker [http://waylight3.blog.me/]
- 국제공부연맹 [http://cafe.naver.com/dostudynow]
- ISUEA1 프로그램 [http://waylight3.blog.me/220237139443]