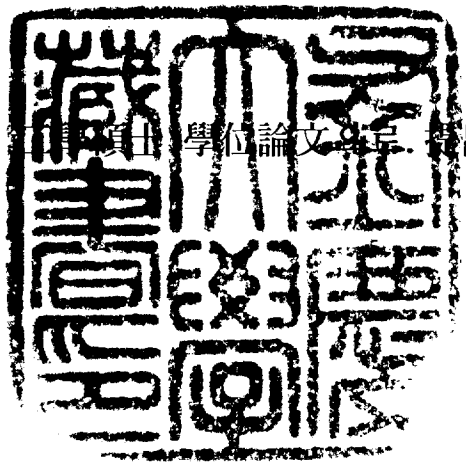
工學碩士 學位論文

데이터 의존 셀룰라 오토마타를 사용한 블록 암호 알고리즘

指導敎授 趙 璟 衍



이 論文을 工學碩士 學位論文으로 提出함

２００２년 2월

釜慶大學校 産業大學院

컴 퓨 터 공 학 과

金 吉 浩

이 論文을 金吉浩의 工學碩士
學位論文으로 認准함

2001년 12월 15일

主 審 工學博士 曺 佑 鉉 

委 員 工學博士 辛 奉 烋 

委 員 工學博士 趙 璟 衍 

목 차

Abstract v

1. 서 론 1

2. 블록 암호 알고리즘 3

2.1 DES 3

2.1.1 DES 암호화 알고리즘 3

2.1.2 F 함수 5

2.2 AES 6

2.2.1 암호화 과정 7

2.2.2 키 스케줄링 10

2.2.3 복호화 과정 11

2.3 암호방식의 운영모드 12

2.3.1 ECB 모드 13

2.3.2 CBC 모드 14

2.3.3 CFB 모드 15

2.3.4 OFB 모드 15

2.4 프로그램 가능 셀룰라 오토마타 16

3. Pkc128 블록 암호화 알고리즘 20

3.1 암호화 과정 20

3.1.1 F 함수 22
3.2 확장키 알고리즘 24
3.2.1 암호화 키 스케줄링 25
3.2.2 복호화 키 스케줄링 28
3.3 복호화 과정 31
4. 구현과 통계검정 33
4.1 빈도 검정 33
4.2 계열 검정 34
4.3 포커 검정 35
4.4 런 검정 37
4.5 자기 상관 검정 38
5. 결 론 40
참고문헌 41
부록 42

그 림 목 차

그림 2-1 DES 암호화 과정 4
그림 2-2 f (R,K)의 동작 6
그림 2-3 ECB 모드 13
그림 2-4 CBC 모드 14
그림 2-5 CFB 모드 15
그림 2-6 OFB 모드 16
그림 2-7 PCA의 구조 18
그림 3-1 암호화 과정 21
그림 3-2 F 함수 22
그림 3-3 PCA 구현 24
그림 3-4 확장키 생성과정 27
그림 3-5 복호화 과정 32
그림 4-1 빈도 검정 34
그림 4-2 계열 검정 35
그림 4-3 포커 검정 36
그림 4-4 런 검정 37
그림 4-5 자기 상관 검정 38

표 목 차

표 2-1 라운드 수의 정의 8

표 2-2 쉬프트 오프셋 9

표 2-3 전환함수 17

표 3-1 각 라운드에서 사용된 확장키 30

표 4-1 통계 검정 결과 39

A Study on Block Cipher using Data Dependent Cellular Automata

Gil-Ho Kim

Department of Computer Engineering
Graduate School of Industry
Pukyong National University

Abstract

In this paper, a block cipher algorithm tentatively called Pkc128 is proposed. In the Pkc128 block cipher algorithm, the size of input block, output block and key are all 128 bits. It has the Feistel Network structure.

The F-function consists of 3 parts, the one part is the data dependant rotation of input data and round key, and the one part is the round key switched programmable cellular automata. And the other part is the data dependant rotation of the previous two parts.

The round key generation also uses the data dependant rotation and the programmable cellular automata. The round key could be generated while encryption and decryption is proceeding, which is a very important feature at the limited memory application such as a smart card.

To verify the security of proposed Pkc128 algorithm, this paper takes a FIPS(Federal Information Processing Standards) 140-1 statistic tests on the stream of the output. As a result, it is passed through FIPS 140-1 statistic test.

1. 서론

1977년 IBM이 개발하고, 미국 정부에 의해 수정되어 미국 정부의 암호 표준으로 채택된 DES(Data Encryption Standard)는 최근까지 널리 사용되고 있는 암호 알고리즘이다. 그러나 최근 컴퓨터 계산 능력과 암호 해독 기술의 발달로 인해 DES가 해독되는 등 보안, 관리상의 취약점 및 문제점이 발견되었다. 이에 미국 국립 표준 기술연구소(NIST, National Institute Standards & Technology)에서는 새로운 표준 블록 암호 알고리즘을 선정하기 위해 1997년 초 새로운 표준 암호 알고리즘(AES, Advanced Encryption Standard) 선정 프로젝트를 발표했다. NIST는 차세대 표준 암호 알고리즘으로 128비트 블록 암호 알고리즘, 다양한 길이의 키(128, 192, 256비트)를 사용할 수 있고, 취약키를 갖지 않으며, 소프트웨어와 하드웨어 상에서 효율적이며, 스마트 카드 상에서도 동작 할 수 있는 알고리즘 등의 기준을 제시했다.

AES 선정 프로젝트는 전 세계적으로 후보 알고리즘을 공모하였고, 여러 차례 평가에 의해 5개의 후보 알고리즘(Rijndael, Twofish, MARS, RC6, Serpent)을 선정하였다. 5개의 후보 중 NIST 자체평가와 전 세계적인 공개 검정을 통해 지난 2000년 10월에 최종적으로 Rijndael이 선정되었다[1,2].

그런데 Rijndael은 소프트웨어 및 하드웨어 구현 시에 속도가 느린 문제점을 가지고 있다. 한편 AES의 후보 알고리즘 중 하나인 RC6은 데이터 의존 회전 기법과 32비트 곱셈 연산을 사용한 알고리즘으로 하드웨어로 구현 시에 속도가 빠른 장점을 가지지만, 곱셈기 구현에 많은 전자회로를 필요로 한다[3,4]. 따라서 이들 알고리즘들은 스마트카드 등 소규모 하드웨어로 암호 기능을 구현하여야 하는 응용분야에 적합하지 않다.

이러한 문제점을 해결하기 위해 본 논문에서는 데이터 의존 회전 기법

과 프로그램 가능 셀룰라 오토마타[5] 기법을 사용한 가칭 Pkc128(PuKyong Cipher 128) 블록 암호 알고리즘을 제안한다.

제안한 Pkc128은 Feistel Network 구조를 취하며 128 비트 블록 크기를 가진다. 키는 128 비트 이상 가변 길이를 가지며, 본 논문에서는 128 비트 키로 고정하여 설명하였다.

또한 스마트 카드와 같은 메모리가 한정된 응용분야에도 적합하도록 암복호 과정과 병행하여 실시간적인 확장키 생성을 가능하도록 하였다.

제안한 알고리즘의 안전성을 검정하기 위하여 출력 스트림에 대해서 통계적 검정을 실시하였다. 테스트는 FIPS140-1에서 제안한 빈도, 계열, 포커, 런 그리고 자기상관 검정을 수행하였다[6]. 16 회전 시에 검정 결과 모든 검정 과정을 통과하여 알고리즘이 안전함을 확인하였다.

제2장에서는 전반적인 블록 암호 기술인 DES, AES에 대해 알아보고 블록 암호 운영 방법[7]과 PCA에 대해 소개한다. 제3장에서는 Pkc128 블록 암호 알고리즘에 대해서 자세히 서술한다. 제4장에서는 구현 및 통계검정을 하고, 제5장에서 결론으로 끝맺는다.

2. 블록 암호 알고리즘

2.1 DES(Data Encryption Standard)

2.1.1 DES 암호화 알고리즘

DES는 평문 64비트를 암호문 64비트로 변환시키는 암호 방식으로 64비트의 키를 사용하고 있다. 이 키는 8 비트마다 패리티(Parity) 비트 하나씩을 포함하고 있어 DES의 암호화 과정에는 56비트만이 적용된다.

DES 암호 알고리즘의 기본 동작은 전치, 치환과 mod 2 연산으로 구성되어 있다. 다시 전치는 평형 전치, 확대 전치 그리고 축약 전치 등의 세 종류가 있으며 치환은 S-box를 통해서 이루어진다. 전치와 치환 그리고 mod 2 연산으로 구성된 DES의 암호화 과정은 그림 2-1과 같다.

DES의 동작 과정을 살펴보면 다음과 같이 크게 세 가지 과정으로 나눌 수 있다.

1. 평문 M의 64비트는 초기 전치(initial permutation) IP를 거쳐$IP(M) = M_0$는 32비트씩 나누어져 L_0 , R_0로 나누어진다.

2. 초기 전치 출력 R_0, L_0는 아래와 같은 함수계산을 16회 반복한다.

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

⊕는 mod 2 연산을 하는 XOR를 의미하며 L_i , R_i는 그림 2-1의 32비트씩의 중간 데이터를 말한다. 서브키 K_i는 48비트의 DES 암호화키로서 $K_1, K_2, \cdots K_{16}$의 값은 서로 다르며 16회 암호화 과정에 사용된다. 함수 f

는 S-Box를 포함한 치환 과정을 의미한다.

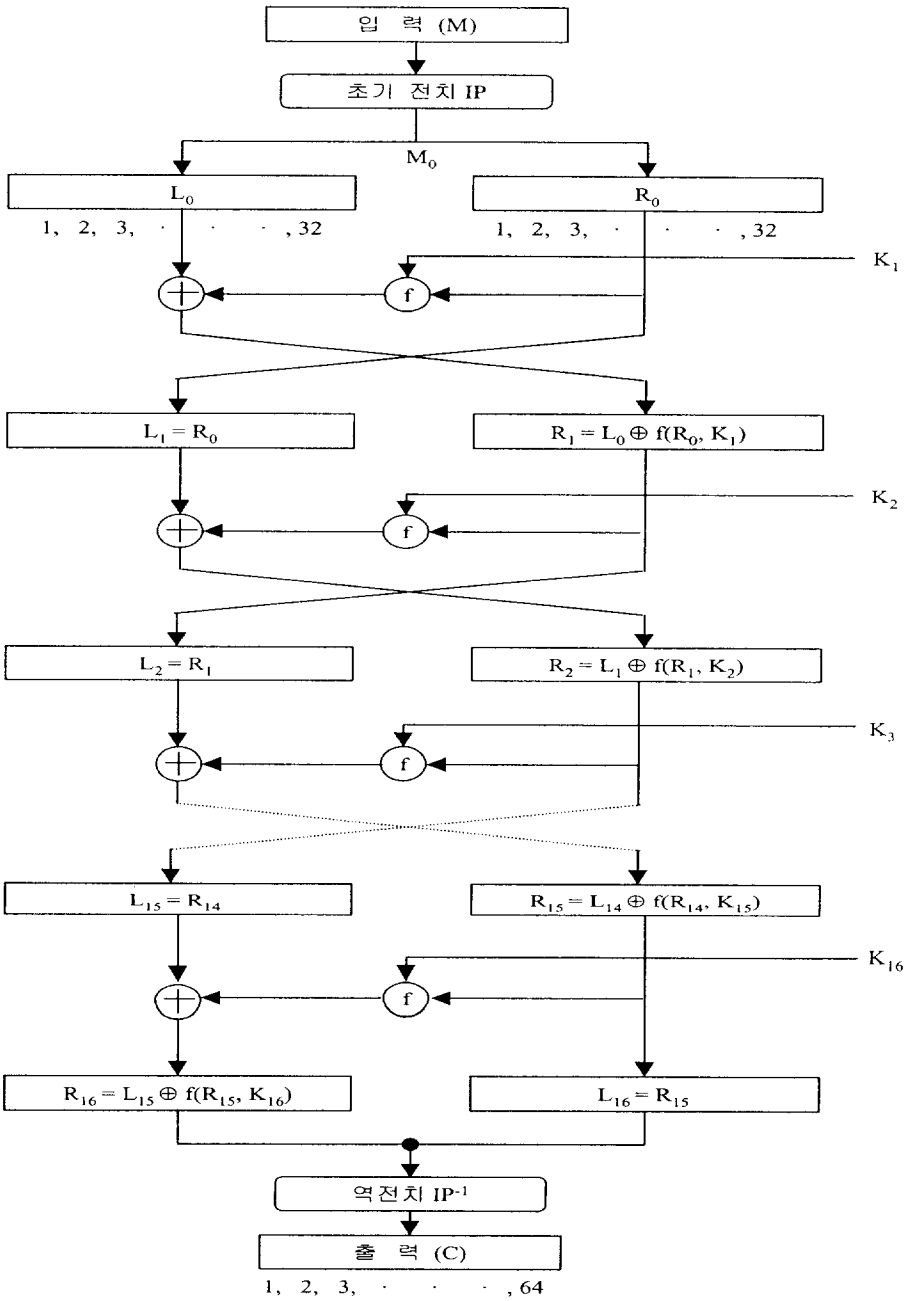


그림 2-1 DES 암호화 과정

3. R_{16}과 L_{16}은 초기 전치의 역전치인 IP^{-1}를 거쳐 64비트의 암호문이 된다. 즉, $C = IP^{-1}(R_{16}, L_{16})$으로 L_{16}과 R_{16}이 서로 반대로 되어 있는 것에 유의해야 한다. IP^{-1}는 IP의 역전치이다.

2.1.2 F 함수

그림 2-2의 f함수는 32비트의 R_i입력과 48비트의 서브키 K_{i-1}입력에 대하여 32비트의 중간 데이터로 치환시키는 과정이다. 함수 f의 동작 과정은 다음과 같다.

1. R_i입력 32비트는 확대 전치(Expansion Permutation) E를 거쳐 48비트로 확대된다. $E(R_i)$는 입력 R_i의 32비트 중 16비트는 두 번 나타나게 된다. 즉 1, 4, 5, 8, 9, 12, 13, 16, 17, 20, 21, 24, 25, 28, 29, 32번째 비트는 2회씩 나타난다.

2. 확대 전치 출력 $E(R_i)$는 서브키 K_{i-1}과 XOR연산 후 6비트씩 8개로 나누어져 각각 8개의 S-box에 입력이 된다. S_j-box의 입력 $B_j = b_1b_2b_3b_4b_5b_6$은 S_j-box의 표에서 b_1b_6은 행을 지정하고 $b_2b_3b_4b_5$는 열을 지정하여 행 과 열이 만나는 지점의 숫자가 2진수로 바뀌어 S_j-Box출력이 된다.

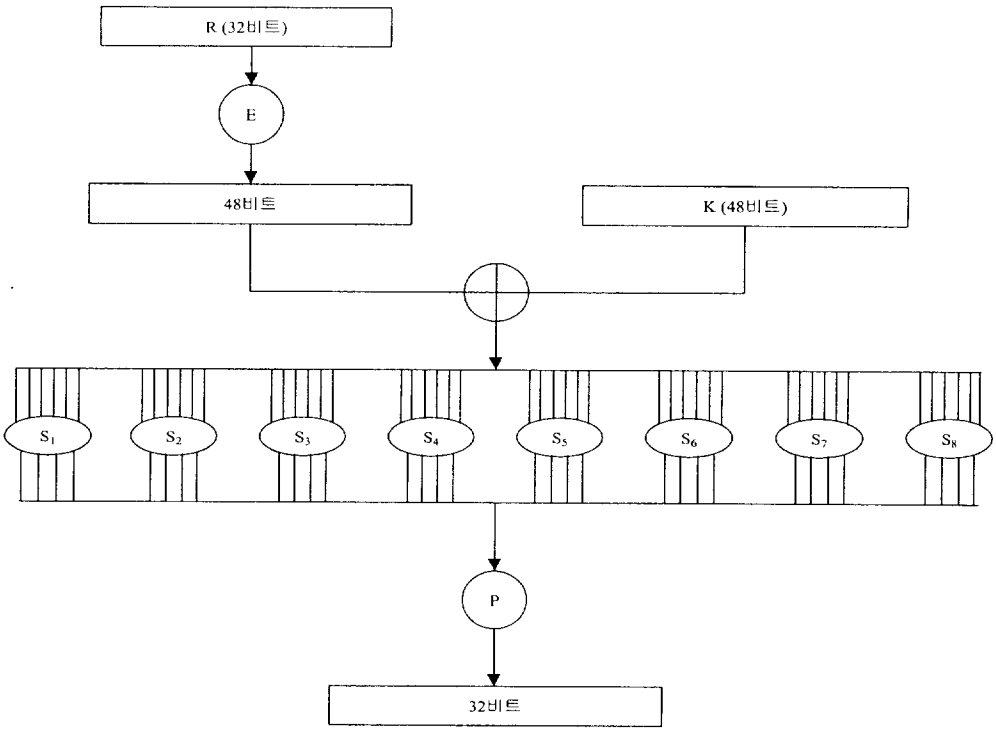


그림 2-2 F 함수

3. S_j-box의 출력이 4비트이므로 8개 S-box출력의 합은 32비트가 된다. S-box의 출력은 다시 평형 전치 P를 거쳐 f함수의 출력 $f(R_i, K_{i+1})$이 된다.

2.2 AES(Advanced Encryption Standard - Rijndael)

Rijndael은 Daemen과 Rijnmen에 의해 개발되었고 2000년 10월에 AES 알고리즘으로 최종 선정되었다.

가변블록 길이를 지원하는 블록 암호 알고리즘으로 지원 블록 길이는 128, 192, 256비트이며, 각 블록 길이에 대해 128, 192, 256비트의 키를 사용

할 수 있다. 라운드 수는 키의 길이에 의해 결정되며 128비트 블록을 사용하는 경우 128, 192, 256비트 키에 대해 각각 10, 12, 14라운드를 사용하도록 권고되고 있다.

Rijndael 디자인의 세 가지 특징으로는 다음과 같다.

- 모든 알려진 공격에 강하다.
- 여러 플랫폼에서 빠르고 코드가 간단하다.
- 디자인이 간단하다.

Rijndael 은 대개의 블록 암호 알고리즘과 달리 Feistel networks 구조가 아니다. 대신에 암호화 과정에서 세 개의 균등한 변환 형태를 구성하는 layer로 각 State의 모든 비트들은 비슷한 방법으로 다루어진다.

-Linear mixing layer : 여러 라운드에 걸친 높은 확산
-Non-linear layer : S-box의 병렬 적용
-Key addition layer : 중간 State에 라운드 키의 XOR연산

2.2.1 암호화 과정

State는 암호화 과정의 중간 결과로서 2차원 바이트 배열로 구현된다. 배열은 4행이며 그 배열의 열의 수는 Nb로 표시한다. Nb는 블록 길이를 32로 나눈 값이다.

암호화 키 역시 4행의 바이트 배열로 구성되고 열의 수는 Nk로 표시하며 Nk는 키의 길이를 32로 나눈 값이다. 라운드 수는 Nr로 표시하며, Nb나 Nk의 값에 의해 정의된다.

표 2-1 라운드 수 정의

Nr	Nb=4	Nb=6	Nb=8
Nk=4	10	12	14
Nk=6	12	12	14
Nk=8	14	14	14

암호화 과정은 4개의 다른 함수로 구성된다. C-언어로 표시하면 다음과 같다.

```
Cipher (State, RonudKey)
{
  ByteSub(State);
  ShiftRow(State);
  MixColunm(State);
  AddRoundKey(State, RoundKey);
}
```

암호화 과정에서 마지막 라운드는 약간 다르며 이것은 다음과 같다.

```
FinalCipher(State, RoubdKey)
{
  ByteSub(State);
  ShiftRow(State);
  AddRoundKey(State, RoundKey);
}
```

암호화 과정에서 ByteSub()는 비 선형 바이트 치환으로 각각의 State내의 바이트는 자유롭게 S-box와 치환이 된다. S-box는 다음과 같이 하여 구할 수 있다.

- $GF(2^8)$에서의 곱셈상의 역함수를 구한다.
- GF(2)에 대해 아래의 연산을 수행한다.

$$b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus C_i$$

여기서 C_i는 이진수로 {0 1 1 0 0 0 1 1}을 나타내다.

ShiftRow()는 State의 열을 다른 오프셋을 통하여 주기적으로 쉬프트 된다. 제로 열은 쉬프트 되지 않고 1열은 C1바이트를 통하여 쉬프트 되고 2열은 C2, 3열은 C3바이트를 통하여 쉬프트 된다.

표 2-2 쉬프트 오프셋

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

MixColunm()는 State의 행은 GF상의 다항식으로 여기고 고정된 다항식 C(x)의 모듈러 $X^4 + 1$을 곱한다. C(x)는 다음과 같이 주어진다.

$$C(x) = '03'X^3 + '01'X^2 + '01'X + '02'$$

C(x)는 다항식 $X^4 + 1$와 서로 소이므로 역원이다. 이것은 곱셈배열처럼 다음과 같이 쓸 수 있다.

$$b(x) = C(x) \otimes a(x)$$

MixColunm()의 역은 MixColunm()과 유사하다. 다항식 d(x)와의 곱셈연

산은 다음과 같이 표시된다.

$$('03'X^3 + '01'X^2 + '01'X + '02') \otimes d(x) = '01'$$

여기서 d(x)는 '0B'X^3 + '0D'X^2 +'09'X +'0E'와 같이 주어진다.

AddRoundKey()는 State와 RoundKey를 XOR연산을 수행하며, RoundKey는 키 스케줄링 알고리즘에 의해 만들어진다.

2.2.2 키 스케줄링

확정키를 생성하는 키 스케줄링은 다음과 같다.

```
KeyExpansion(byte Key[4*Nk], word W[Nb*(Nr-1)])
{
  for(i=0; i<Nk; i++)
      W[i] = word[Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3]];
  for(i=Nk; i<Nb*(Nr-1); i++)
     {
       temp = W[i-1];
       if(i % Nk == 0)
         temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
       W[i] = W[i-Nk] ^ temp;
     }
}
```

RotByte()는 입력된 워드를 1바이트 왼쪽 회전을 수행하는 것이고 SubByte()는 암호화 과정에서 사용한 S-box를 그대로 적용하는 함수이다. Rcon[i]는 라운드 수에 따라 주어지는 상수이다.

2.2.3 복호화 과정

복호는 암호과정을 역으로 수행한다. 간략하게 표현하면 다음과 같다.

```
InvCipher(State, RoundKey)
{
  AddRoundKey(State, RoundKey);
  InvMixColunm(State);
  InvShiftRow(State);
  InvByteSub(State);
}
InvFinalRound(State, RoundKey);
{
  AddRoundKey(State, RoundKey);
  InvShiftRow(State);
  InvByteSub(State);
}
```

동등한 복호화 과정

```
EqInvCipher(State, dw)
{
  AddRoundKey(State, dw[Nr*Nb]);
  for(i=Nr-1; i=>1; i--)
      {
          InvSubByte(State);
          InvShiftRow(State);
          InvMixColunm(State);
```

```
            AddRoundKey(State, dw[i*Nb]);
        }
    InvSubByte(State);
    InvShiftRow(State);
    AddRoundKey(State, dw);
}
```

동등한 복호화 과정은 일반적인 복호화 과정과 다른 방법이며 두 가지 Stste 전환의 특징으로 동등한 복호화 작업을 할 수 있다. 첫 번째로 ShiftRow()와 ByteSub()의 순서는 상관이 없으며, 단지 ShiftRow()에서 바이트 값은 아무런 영향을 주지 않고 바이트 단위로 간단히 자리만 바꾸는 것이다. ByteSub()는 위치와는 독립적으로 바이트 단위로 S-box와 치환이 일어난다.

두 번째로 AddRoundKey(), InvMixColumn()순으로 적용하던 것을 InvMixColumn(), AddRoundKey()순으로 적용해서 라운드 키와 복호화 라운드 키를 얻을 수 있다.

복호화 작업에 필요한 확장키는 다음과 같이 얻을 수 있다.

- 확장키 생성 알고리즘을 그대로 적용.
- 첫 번째와 마지막을 제외한 모든 라운드 키에 InvMixColumn()을 적용.

2.3 암호방식의 운영모드

블록 암호 방식의 이용모드는 암호 할 평문의 크기가 한 블록보다 큰 가변적인 평문의 경우에는 암호가 적용될 수가 없는 것처럼 보인다. 하지만 이러한 문제점을 해결하고 또한, 다양한 응용 환경에서 적절한 암호화 도구

로 사용될 수 있는 여러 유형의 효율적인 운영 방식들이 제시되고 있다. 이러한 운영 방식은 ECB 모드(electronic code book mode), CBC 모드(cipher block chaining mode), CFB 모드(cipher feed back mode), 그리고 OFB 모드(output feed back mode) 등이 있다.

2.3.1 ECB(electronic code book)

ECB 모드는 암호 운영 방식 중 가장 간단한 방법으로 평문을 64비트씩 나누어 암호화하는 방식이다. 평문을 64비트씩 나눌 때 마지막 블록이 64비트가 되지 않을 때는 임의의 약속된 비트 모양을 패딩(padding)하게 된다. 64비트 블록의 평문을 그림 2-3과 같이 암호화한다.

이 방식은 동일한 평문 블록 모양에 따라 항상 동일한 암호문이 출력되므로 암호 해독자 들의 해독 가능성을 높게 만든다. 즉, 문서의 종류에 따라 동일한 문서 모양을 갖고 있으므로 암호문 단독 공격의 가능성을 높게 해준다. 따라서 심각한 보안상의 문제점으로 제기될 수 있다. 특히 모든 블록들이 독립적으로 암호화되기 때문에 암호화된 블록들을 임의로 재배열하면 결국 해당 평문들을 재배열한 것과 동일한 효과를 얻을 수 있게 된다.

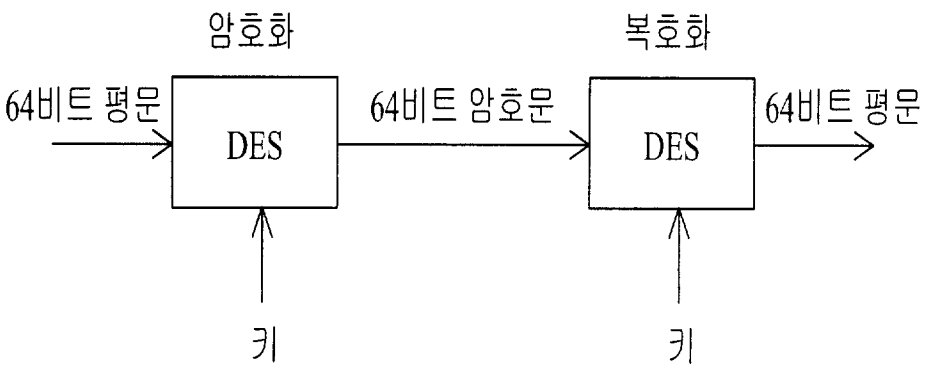


그림 2-3 ECB 모드

2.3.2 CBC(cipher block chaining)

암호 방식 중 CBC 모드는 출력 암호문이 다음 평문 블록에 영향을 미치게 하여 각 암호문 블록이 전단의 암호문의 영향을 받도록 만든 방식으로 ECB에서 발생하는 동일한 평문에 의한 동일한 암호문이 발생하지 않도록 구성한 동작 모드이다.

CBC 모드 동작은 그림 2-4와 같이 처음 입력된 평문 블록 Pi-1은 초기 벡터 IV_0(initial vector)와 XOR되어 암호기에 입력된다. 암호기 출력 암호문 Ci-1은 다음 단 평문 블록 Pi와 XOR되어 암호기에 다시 입력된다.

CBC 모드의 특징은 현 단계에서 생성된 암호문 블록이 그 다음으로 생성되는 암호문 블록에 영향을 미치기 때문에 특정 암호문 블록이 전달되는 과정에서 발생되는 채널상의 잡음에 의한 오류는 해당 암호문 뿐만 아니라 그 다음 암호문에도 그 효과가 연장된다. 이것을 오류의 파급(error propagation)이라 한다.

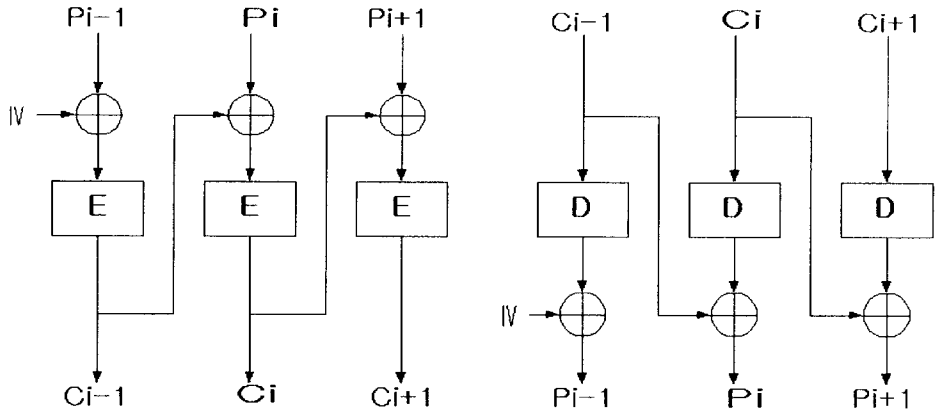


그림 2-4 CBC 모드

2.3.3 CFB(cipher feed back)

CFB 모드도 CBC 모드와 마찬가지로 평문 블록이 동일한 경우 동일한 암호문이 나타나지 않도록 전단의 암호문이 다음 단의 평문에 영향을 미치게 구성하는 방식이다. CFB 모드 동작은 그림 2-5와 같이 CBC를 연상시키게 구성되어 있으나, 다른 점은 암호문이 수신자의 암호기 입력으로 사용되는 것이다.

CFB 모드는 현재 생성된 암호문이 이후에 생성되는 암호문에도 영향을 미친다는 측면에서 CBC모드와 유사하다. 따라서 암호문에 발생되는 채널상의 잡음에 의한 오류는 해당 암호문의 복호화뿐만 아니라 연속적으로 발생되는 암호문의 복호화에도 영향을 미친다.

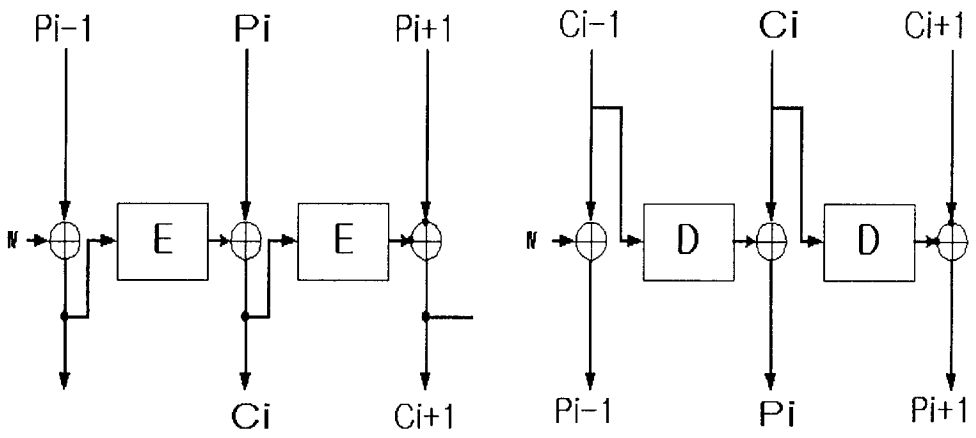


그림 2-5 CFC 모드

2.3.4 OFB(output feed back)

OFB 모드 동작은 평문을 서로 독립적으로 암호화하는 ECB 모드의 단점과 오류 전파가 발생하는 CBC 모드와 CFB 모드를 개선한 동작 모드이다. OFB 모드 동작 구성도는 그림 2-6과 같다.

OFB 모드 동작은 암호기의 출력과 평문을 XOR하여 암호문을 생성하고 있으므로 오류 전파가 발생하지 않고 단지 해당 암호문만이 영향을 받는다. 그러나 암호문 송신자와 수신자 사이에 동기를 조절해야 한다. 즉, 전송중인 암호문의 비트 손실이나 삽입 등에 유의해야 하는 방식이다.

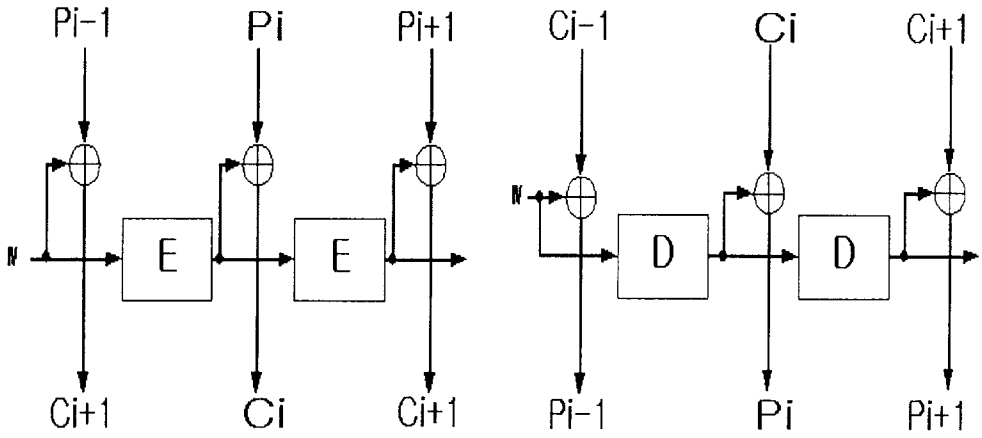


그림 2-6 OFB 모드

2.4 프로그램 가능 셀룰라 오토마타

셀룰라 오토마타는 복잡한 자연 시스템을 간단한 항등식으로 구성한 수학적 모델이다. 셀룰라 오토마타는 암호학, 인공지능, 행렬연산, 에러 체크, VLSI등 다양한 여러 분야에서 응용되고 있다.

셀룰라 오토마타에 대해 자세히 살펴보면 셀룰라 오토마타는 셀이라 하는 동일한 사이트의 배열로 구성된 유한 상태머신이다. 이산 시간 단계에서 각각의 셀들은 전환함수나 룰에 의해 동기적으로 출력을 만들어 낸다. 여기서 전환함수는 주어진 셀의 다음 상태를 결정하는 함수로서 주어진 셀의 이웃 셀들의 상태값과 자신이 현재 상태값을 기본으로 다음 상태를 결정한다.

만일 셀 i가 k개의 이웃 셀과 연결되었다면 k개의 이웃 셀룰라 오토마타라고 하며 셀 i의 다음 상태는 셀 i의 현재 상태와 전환함수에 의한 k개의 이웃 셀의 현재 상태에 의해 결정된다.

또한 셀룰라 오토마타는 다양한 한계조건이 있는데 맨 끝 셀이 null과 연결 되어있다면 NBCA(null boundary CA)라고 하고 맨 끝 셀이 연속적인 셀처럼 사용된다면 PBCA(periodic boundary CA) 라고 한다.

표 2-3 전환함수

Rnles	111	110	101	100	011	010	001	000
rule60	0	0	1	1	1	1	0	0
rule90	0	1	0	1	1	0	1	0
rule150	1	0	0	1	0	1	1	0
rule240	1	1	1	1	0	0	0	0

표 2-3은 2상태 3이웃 NBCA의 전환함수를 설명하는 것이다. 표 2-3에서 각 행들은 8bit의 2진수로서 rule옆에 표시된 십진수 값은 각각의 셀의 전환함수에 의해 정해진 다음 상태값을 십진수로 표시한 것이다. 이 십진수 값이 전환함수의 이름이다.

2상태 3이웃 셀은 2^{2^3} 개의 전환함수와 2^3 개의 이웃을 가지고 있다.

i번째 셀의 다음 상태 전환은 (i-1)번째 셀과 i번째 셀 그리고 (i+1)번째 셀의 현재 상태 함수로서 표시된다.

$$S_i^{t+1} = f\ (\ S_{i-1}^t\ ,\ S_i^t\ ,\ S_{i+1}^t\ )$$

여기서 f 함수는 t 시간단계의 i번째셀 S_i^t 의 상태와 조합논리로 구성된

셀룰라 오토마타의 전환함수를 나타낸다. 표 2-3에 나타난 rule90 과 rule150은 다음과 같다.

$$\text{rule90} : S_i^{t+1} = S_{i-1}^t \oplus S_{i-1}^t$$

$$\text{rule150} : S_i^{t+1} = S_{i-1}^t \oplus S_i^t \oplus S_{i+1}^t$$

셀룰라 오토마타와 동일한 구조이면서 적당한 논리연산을 적용한 스위치를 사용해서 셀룰라 오토마타를 다르게 구성 할 수 있는데 이를 프로그램 가능 세룰라 오토마타(Programmable Cellular Automata)라고 부른다. 그림 2-7은 2상태 3이웃 PCA를 나타내며 여기서 Cl, Cs, 그리고 Cr은 각각의 시간에 각 셀에 적용할 컨트롤 신호이다.

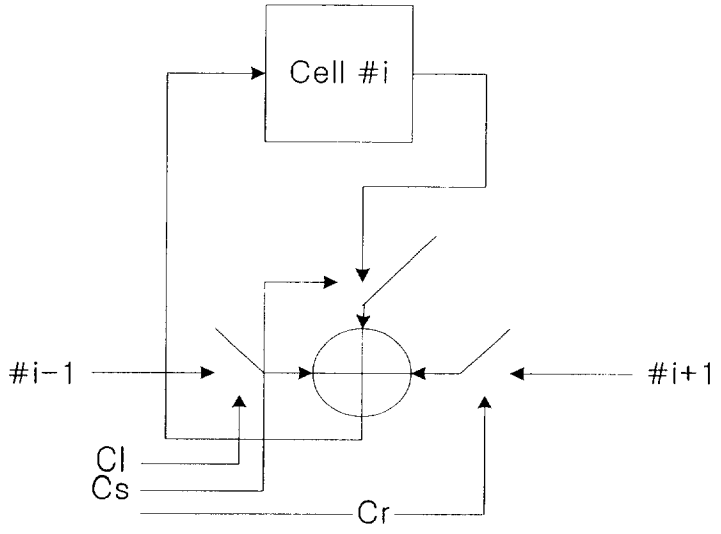


그림 2-7 PCA의 구조

셀룰라 오토마타는 다양한 방법으로 분류 할 수 있는데 셀룰라 오토마타를 XOR연산만으로 구성된 것을 선형 셀룰라 오토마타(linear CA)라고 하고 그렇지 않을 경우는 비선형 셀룰라 오토마타(nonlinear CA)라고 한다.

그리고 셀룰라 오토마타내의 모든 셀에 같은 전환함수를 적용하는 경우에는 균질 셀룰라 오토마타(uniform CA)라고 하고 그렇지 않고 셀마다 서로 다른 전환함수를 적용할 경우에는 하이브리드 셀룰라 오토마타(hybrid CA)라고 한다.

3. Pkc 128 블록 암호 알고리즘

Pkc 128 블록 암호 알고리즘은 Feistel Networks 구조로 16라운드를 수행하며 128비트 블록 크기에 키는 128비트 이상 가변이며 128비트 한 블록에서 확장키는 각 라운드마다 두 개의 확장키를 사용하고 라운드 전후에 각각 두 개씩 확장키를 사용하여 총 36개의 확장키를 사용한다.

본 논문에서 제안한 Pkc 128 블록 암호 알고리즘은 스마트 카드와 같은 소규모 하드웨어 상에 암호 기능을 구현하는 응용 분야에 사용할 목적으로 제안한 블록 암호 알고리즘으로 하드웨어 구현을 쉽고 간단하게 하고 수행 속도를 빠르게 하며 알고리즘의 안정성을 향상시키기 위해서 데이터 의존 회전 기법과 프로그램 가능 셀룰라 오토마타 기법을 사용했다.

그리고 암호화 복호화 수행 중에 확장키를 실시간적으로 생성이 가능하도록 키 스케줄링 알고리즘을 정의하였다.

3.1 암호화 과정

Pkc 128 블록 암호 알고리즘은 128비트 단위로 평문을 암호문으로 만든다. 먼저 평문 128비트를 파일로부터 읽고 난 다음 4개의 32비트 임시 저장 레지스터 L0, L1, R0, R1에 저장한 후 라운드를 수행하기 전에 데이터 의존 회전을 수행한다. 초기 데이터 의존 회전에는 확장키 2개를 사용하여 L0, L1 만을 업데이트 한다.

그 다음 라운드를 수행하는데 제1라운드에서는 L0, L1과 라운드 확장키 2개를 사용하여 F 함수를 수행한다. F 함수는 L0, L1 값과 32비트 확장키 2개를 사용하여 4개의 32비트 중간 결과 값을 만들어 낸다.

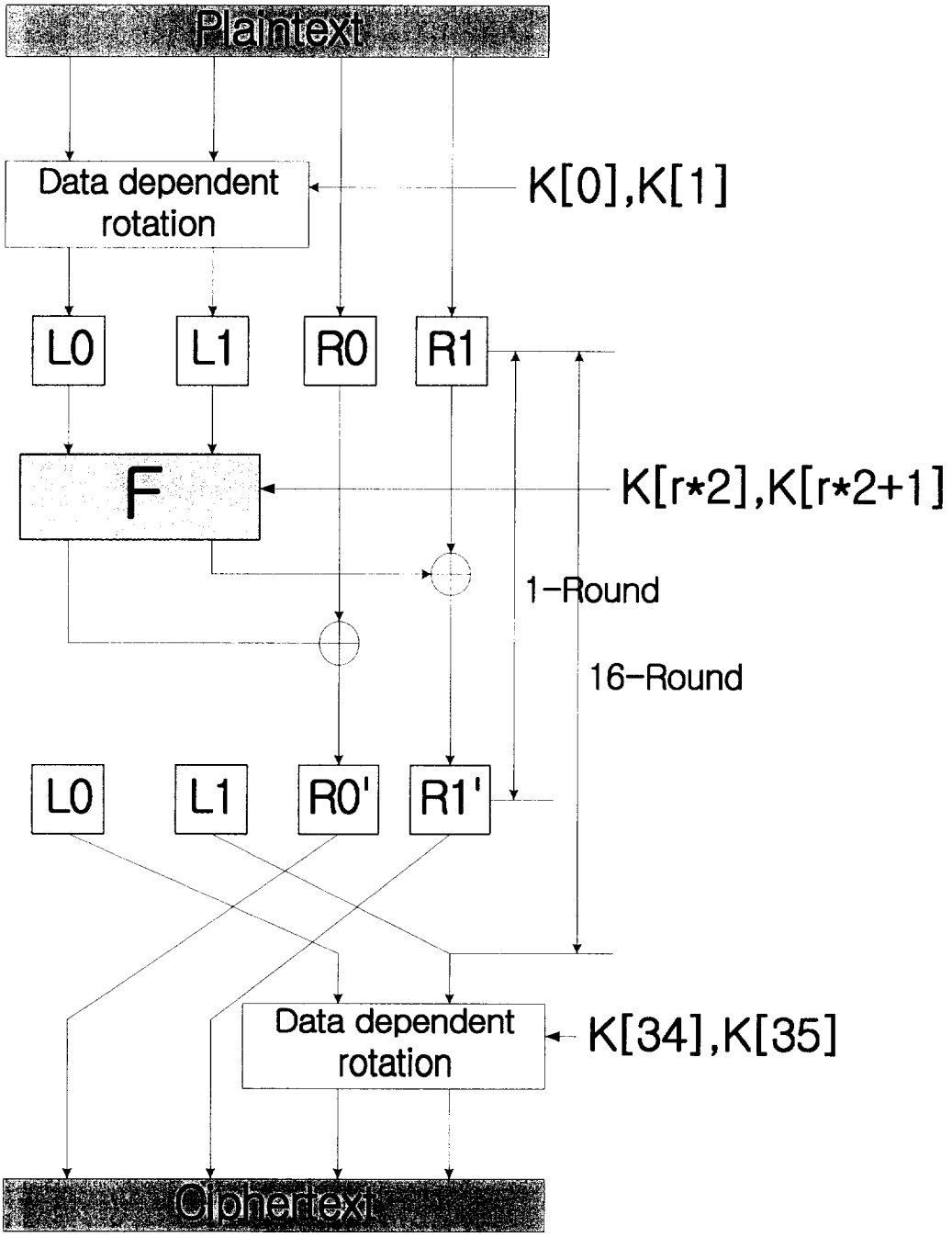

Plaintext
Data dependent rotation
K[0],K[1]
L0
L1
R0
R1
F
K[r*2],K[r*2+1]
1-Round
16-Round
L0
L1
R0'
R1'
Data dependent rotation
K[34],K[35]
Ciphertext

그림 3-1 암호화 과정

그 중간 결과 값은 R0와 R1값과 XOR연산을 취하여 최종적으로 R0′와 R1′로 업데이트 된다. 이때 L0, L1은 아무런 변화가 없고 단지 R0, R1을 업데이트 시키는 피 연산자로만 사용된다. 여기까지가 1라운드로 다음 라운드로 진행할 때는 4개의 임시저장 레지스터가 각각 2개씩 나누어 서로의 위치가 바뀌어서 입력된다. 다시 말해서 L0, L1은 다음 라운드 입력으로 R0, R1 위치로 가게 되고 R0′, R1′은 다음 라운드 입력으로 L0, L1의 위치로 입력된다.

이와 같은 과정을 16회 반복 수행한 후 마지막 데이터 의존 회전을 수행하는데 여기서는 마지막 확장키 2개를 사용하여 R0′와 R1′을 업데이트 시킨다. 이 과정을 끝으로 해서 최종적으로 128비트 암호문을 생성해 낸다.

3.1.1 F 함수

본 논문의 핵심이라고 할 수 있는 F 함수는 그림 3-2에 잘 나타나 있다.

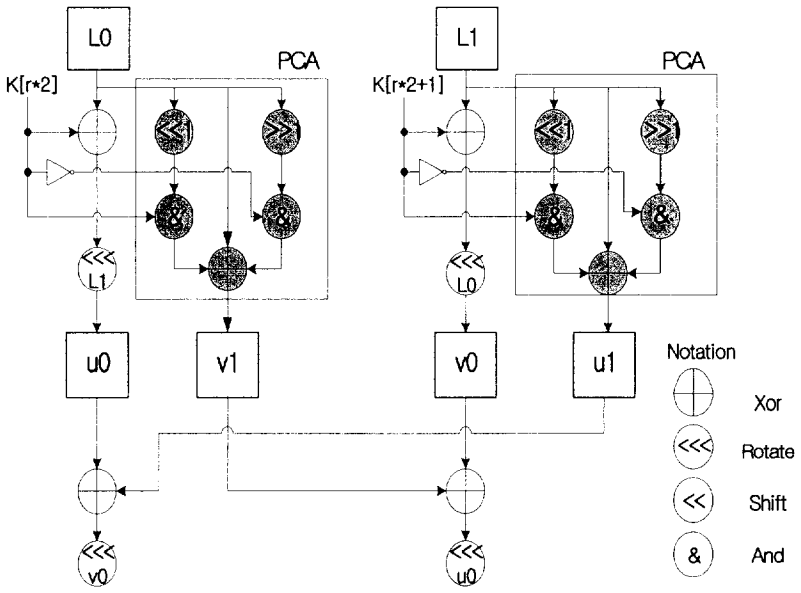


그림 3-2 F 함수

L0 값과 첫 번째 라운드 키 값과 XOR연산을 취하고 난 다음 L1 값만큼 데이터 의존 회전을 수행하여 32비트 중간 결과 값 u0을 만든다. 그리고 L0을 1비트 왼쪽 쉬프트시키고 난 다음 첫 번째 라운드 키와 AND연산을 취하고, L0을 1비트 오른쪽 쉬프트시키고 난 다음 첫 번째 라운드 키를 인버터 시킨 값과 AND연산을 취하고, L0을 아무런 연산을 취하지 않은 3개의 값을 모두 XOR연산을 취하여 v1이라는 32비트 중간결과 값을 만든다. 이와 같은 방법으로 L1역시 수행하여 32비트 중간결과 값 2개 v0과 u1을 만든다.

그리고 u0과 u1을 XOR 연산을 취하고 난 다음 v0만큼 데이터 의존 회전을 수행하여 그림 3-2에는 나타나 있지는 않지만 R0과 XOR 연산을 취하여 R0값을 업데이트 시킨다. 같은 방법으로 R1도 R1'로 업데이트 시킨다.

본 논문에서 제안한 Pkc 128 블록 암호 알고리즘의 하드웨어 구현은 다음과 같다. 우선 하드웨어 구현은 F 함수내의 모든 연산들이 XOR, 회전, 쉬프트 그리고 AND연산과 같은 로직 연산만으로 이루어져 있어 하드웨어 구현을 쉽게 할 수 있으며 특히 그림 3-2에서 PCA로 표시된 부분은 프로그램 가능 셀룰라 오토마타로서 간단히 제작할 수 있다.

PCA에 대해서 간단히 설명하면 셀 이라 불리는 동일 배열로 구성된 일종의 유한 상태 머신으로 이산 시간단위로 변환 함수라 불리는 함수에 의해서 각각의 셀 들은 동기식으로 출력을 만들어 낸다. 이때 변환 함수는 주어진 셀의 이웃 셀들의 값을 기초로 하여 주어진 셀의 다음 상태 값을 결정하는 함수이다.

그림 3-3은 F 함수를 PCA로 구현한 모습이다. F 함수 수행에서 1비트 왼쪽 쉬프트와 1비트 오른쪽 쉬프트 연산은 실제로 로직연산이 수행하는 것이 아니라 그림 3-3의 프로그램 가능 셀룰라 오토마타로서 구현할 수 있다. 여기서 L0(i) 번째 값과 L0(i-1)은 L0(i) 번째의 1비트 이전 값이 되고 L0(i+1)은 L0(i) 번째의 1비트 다음 값이 된다. 그래서 이를 하드와이어드

(hard wired)방식으로 연결하여 간단히 구현할 수 있다.

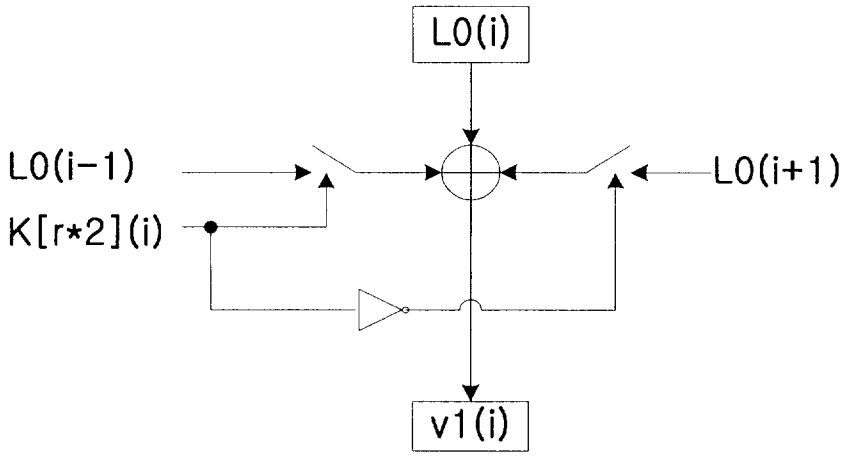


그림 3-3 PCA 구현

이때 라운드 키 K[r*2](i)는 L0(i)의 1비트 이전 값(L0(i-1))을 선택 할 것인지 아니면 L0(i)의 1비트 다음 값(L0(i+1))을 선택 할 것인지를 결정하는 제어 신호 역할을 하게 된다. 하드웨어 제작 시 데이터의 의존 회전은 실제로 하드웨어 제작 시에는 배럴 쉬프트(Barrel Shifter)를 사용하여 구현할 수 있다.

3.2 확장키 알고리즘

블록 암호 알고리즘의 안전성에 확장키를 만드는 키 스케줄링 알고리즘이 많은 영향을 미친다. 그래서 키 스케줄링 알고리즘 또한 안전성이 보장되어야 한다. 먼저 키 스케줄링 알고리즘을 만들 때 몇 가지 고려할 사항을 살펴보면 다음과 같다.

- 취약키를 갖지 말 것
- 1비트의 변화에도 빠르고 넓게 확산이 일어날 것
- 여러 가지 공격에 강할 것

이와 같은 사항을 고려하여 본 논문에서는 키 스케줄링 알고리즘을 고안하였고 암호화 과정에서 사용했던 데이터 의존 회전과 프로그램 가능 셀룰라 오토마타 기법을 그대로 적용함으로서 하드웨어 면적을 줄였을 뿐 아니라 안전성도 확보했다. 암호화 복호화 알고리즘과 키 스케줄링 알고리즘이 유사하거나 같은 것을 사용하는 것은 기존의 블록 암호 알고리즘에서도 많이 볼 수 있으며 특히 AES와 RC6이 그러하다. AES는 암호화 과정에서 $GF(2^8)$내에서 만들어진 S-box를 사용하는데 이는 키 스케줄링 알고리즘 내에서도 똑같은 S-box를 사용하여 확장키를 만들어 내고 있다. RC6 또한 데이터 의존 회전과 32비트 곱셈 연산으로 암호화를 수행하고 키 스케줄링은 데이터 의존 회전만을 사용하여 확장키를 만들어 낸다.

본 논문에서 제안한 확장키를 만드는 키 스케줄링 알고리즘은 일종의 스트림 암호로 연속해서 32비트 값을 만들어내고 있는 구조이다.

3.2.1 암호화 키 스케줄링

Pkc 128 블록 암호 알고리즘은 확장키 크기는 128비트 이상 가변이지만 본 논문에서는 128비트로 고정하여 설명한다. 먼저 사용자가 입력한 키를 확장을 통해 128비트로 만들고 그것은 4개의 32비트 K_0, K_1, K_2, K_3으로 나눈다. 이 4개의 32비트를 사용하여 암호화 과정에서 필요한 확장키를 데이터 의존 회전 기법을 사용해서 연속적으로 만들어 낸다. 확장키를 만드는 키 스케줄링 알고리즘은 다음과 같다.

K_0

K_1

K_2

K_3

$K_4 = ROTL\ (U_1\ ,\ V_1)\ \hat{}\ K_0$

$K_5 = ROTL\ (V_1\ ,\ U_1)\ \hat{}\ K_1$

$K_6 = ROTL\ (U_2\ ,\ V_2)\ \hat{}\ K_2$

$K_7 = ROTL\ (V_2\ ,\ U_2)\ \hat{}\ K_3$

•

•

•

$K_i = ROTL\ (U_{(i/2)-1},\ V_{(i/2)-1})\ \hat{}\ K_{i-4}$

$K_{i+1} = ROTL\ (V_{((i+1)/2)-1},\ U_{((i+1)/2)-1})\ \hat{}\ K_{i-3}$

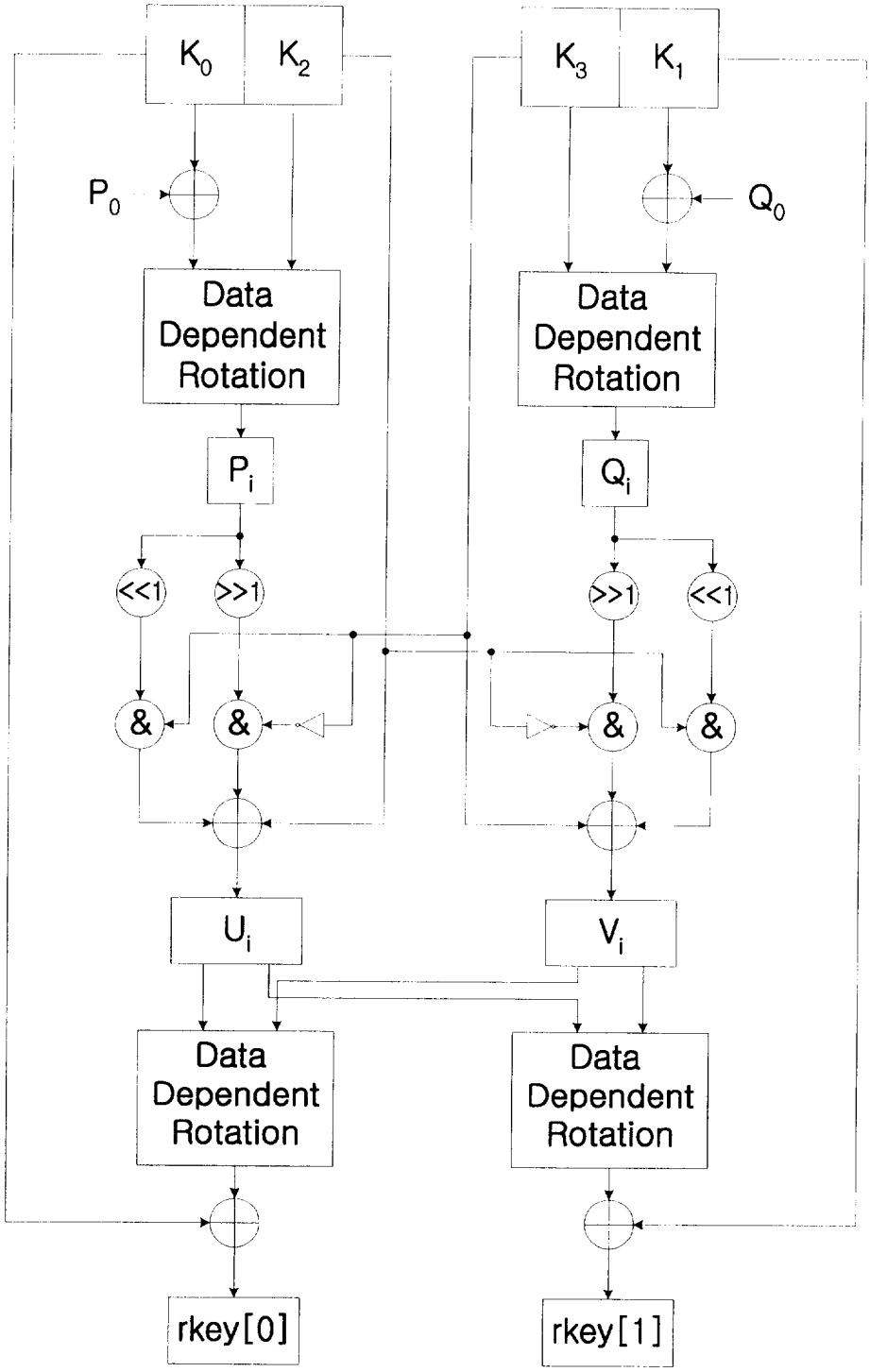


그림 3-4 확장키 생성과정

그림 3-4는 확장 키 생성 알고리즘 전체를 설명하는 것으로 여기서 K_0과 P_0을 XOR연산을 수행한 후 K_2 값에 의한 데이터 의존 회전을 수행하여 새로운 P_i 값을 만든다. 이때 Pi는 S-box 역할을 수행하는 것으로 확장 키 생성과정이 수행될 때마다 새로운 32비트 값을 만들고 있다. 그리고 K_1과 Q_0을 XOR연산을 수행 후 K_3 값에 의한 데이터 의존 회전을 수행하여 새로운 Q_i 값을 만든다. Q_i 역시 P_i와 마찬가지로 S-box 역할을 한다.

만들어진 P_i 와 Q_i를 프로그램 가능 셀룰라 오토마타 기법을 사용하여 새로운 중간결과 값 Ui과 Vi을 만들고 최종적으로 Ui과 Vi을 데이터 의존 회전을 수행 후 K_0과 XOR연산을 취하여 첫 번째 확장 키 rkey[0]을 만들고 K_1과 XOR연산을 통하여 두 번째 확장 키 rkey[1]를 만든다.

여기서 P_0과 Q_0을 32비트로 초기에 주어지는 값은 다음과 같다.

$$P_0 = 0xb7e15163$$
$$Q_0 = 0x9e3779b9$$

3.2.2 복호화 키 스케줄링

본 논문에서는 128비트 한 블록을 암호 또는 복호화 할 때 사용되는 확장키의 총 수는 36개이다. 복호화 확장키는 암호화 과정에서 만들어진 마지막 4개의 32비트 라운드 키 K_{39}, K_{38}, K_{37}, K_{36}을 가지고 복호화 키 스케줄링 함수를 사용하여 확장키 K_{35}, K_{34}, K_{33} 순으로 생성한다. 암호화 과정과 마찬가지로 복호화 키 스케줄링에서도 데이터 의존 회전과 프로그램 가능 셀룰라 오토마타 기법으로 복호화 할 때 사용할 확장키를 생성한다. 다음은 복호화 키 생성 과정이다.

$$K_{39} = ROTL\ (V_{18},\ U_{18})\ \hat{}\ K_{35}$$
$$K_{38} = ROTL\ (U_{18},\ V_{18})\ \hat{}\ K_{34}$$
$$K_{37} = ROTL\ (V_{17},\ U_{17})\ \hat{}\ K_{33}$$
$$K_{36} = ROTL\ (U_{17},\ V_{17})\ \hat{}\ K_{32}$$

여기서 우리가 알 수 있는 확장키는 K_{39}, K_{38}, K_{37}, K_{36} 4개이다. 이 4개의 32비트 확장키를 기초로 하여 K_{35}, K_{34}, K_{33}, K_{32} 순으로 확장키를 생성해야 한다. 먼저 K_{35}를 계산하는 과정은 K_{39}에 V_{18}을 U_{18}만큼 데이터 의존 회전을 수행한 값과 XOR연산을 취하여 K_{35}를 만들 수 있다. 그런데 여기서 V_{18}과 U_{18}를 알 수 없기에 V_{18}과 U_{18}의 계산에 의해서 만들어야 한다. V_{18}과 U_{18} 계산 과정은 다음과 같다.

$$V_{18} = K_{37}\ \hat{}\ (Q_{18} << 1\ \&\ K_{36})\ \hat{}\ (Q_{18} >> 1\ \&\ \tilde{}K_{36})$$
$$U_{18} = K_{36}\ \hat{}\ (P_{18} << 1\ \&\ K_{37})\ \hat{}\ (P_{18} >> 1\ \&\ \tilde{}K_{37})$$

여기서 K_{36}과 K_{37}은 이미 알려진 값이고 다만 P_{18}과 Q_{18}만 알 수 없는 값이다. 이것 역시 계산에 의하여 찾을 수 있다. 계산 과정은 다음과 같다.

$$Q_{18} = ROTR\ (Q_{17}\ ,\ K_{37})\ \hat{}\ K_{35}$$
$$P_{18} = ROTR\ (P_{17}\ ,\ K_{36})\ \hat{}\ K_{34}$$

표 3-1은 확장키 생성 알고리즘에 의해 생성된 확장키를 보여주는 것으로 첫 번째 경우는 32비트의 입력을 모두 0으로 하여 생성된 결과로 표 3-1에서 보는 바와 같이 어떠한 패턴도 보이지 않으며 난수에 가까운 결과를 나타내고 있다. 그리고 두 번째 경우는 K_0 ,K_1 ,K_2는 0을 K_3은 1을 입력으로 하여 확장키를 생성한 결과로 첫 번째와는 1비트의 차이가 있다. 표에 잘

나타나 있듯이 처음 생성된 2개의 확장키만이 유사한 패턴을 보이고 그 이후부터 만들어진 확장키는 첫 번째와는 아무런 연관성을 찾을 수 없을 정도로 난수에 가까운 키를 생성하고 있다. 이것은 본 논문에서 제안한 키 스케줄링 알고리즘이 빠른 확산이 일어나고 있다는 것을 보여주는 것이다. 그리고 세 번째 네 번째 다섯 번째 경우도 1비트 차이를 갖지만 비슷한 패턴 없이 빠른 확산이 일어나는 결과를 보여 주고 있다.

마지막 경우는 취약 키(Weak Keys)로 의심이 되는 입력으로 K_0에 P를 K_1에 Q를 그리고 K_2 ,K_3에 0을 적용하여 생성된 결과이다. 표 3-1에서와 같이 난수에 가까운 확장키를 생성하고 있다.

표 3-1 각 라운드에서 사용된 확장키

0000		0001		0010	
15bf0a8b	79b89e37	15bf0a8a	79b89e37	15bf0a8b	79b8936
5083a807	2d6044cc3	aaffa0f9	96b02661	d003a841	0da4fb75
19e508e2	89c03be5	ef6cc565	ef9e4271	c7c188b2	6cb3787c
e6283ec6	b41f74d	67ad6cb5	ce334788	d9f21fdf	f4a1a923
36930fa6	db5ec08c	f02d500e	6559b52	1319fce3	f6ead64f
433f85e1	5a751652	447b2bac	ad3fb5da	2d4d110f	7759c65b
f8a397d9	a22be35	f9246df0	010c1999	30748c14	5e704d9d
e7d04194	4b0a0cd6	2511fd83	6796652c	7326d50e	3a7086c8
876c5f0a	8b666ca7	48fc4756	e59567e	58732f2c	98a7ce13
061f26e1	fa7f6ffe	fa42467a	8a146452	932bc00a	7ec91e51
33079b21	74da284c	53046d08	1f02e87e	3bf61228	af22d8f0
1ace7a94	c39c2382	414467e9	a7a01edb	20a06d6a	402546fa
a8df9fbc	7d66a412	6fd37708	ae3cb021	d3aef82e	cace700f
38eb9922	28c902c4	2404bal	38a1a433	7abb275b	fe8f6561
02239618	84d23b8	775085af	41a68afa	721151cf	21c8c937
0abbfb40	f8ab7460	22cf7507	8a873081	9953ac88	450f847f
677c9679	c3f8bbed	e674c211	50414c16	40acc8bd	5a1eafc3
0ff4427d	40f0c149	dcfba703	e0725a8b	d50af6c1	2d1042d3

0100		1000		PQ00	
237e1516	3c6ef371	b05bf0a8	79b81e37	b7e15163	9e3779b9
3d9fc425	1a25a511	87daba37	491d60c0	c76fc2a2	f1bbcdcc
b407b52a	692b5ff6	39637013	cb1d5c94	0bf3e5b5	0654987a
68418454	ff474be0	3b328f62	acef23ca	2b3a2779	ebf080ff
e37a8323	e57c3e07	400fd7c4	0171752c	f886752f	a86b5474
1b2fca82	ac052d6d	fe38d10d	142c972c	bbdbf35f	67023b6f
86dc78dc	e6a34a3b	5dfe5c8d	7a69e943	4187a93b	d899d80e
5b1f7ff4	1eee84ff	fec38ba1	7059b06a	dbf4b6b3	13cfb28c
12e74442	6d375dc8	78eee16e	608e0c33	321d6b9b	0e8be49a
fa041873	3e3c1edb	f663b871	1a1182a7	f0e5957f	e42c8176
5e79b638	c25e4742	c74e7fd1	443feeea	3487feed	3e45eecb
ad17ae2e	1f4eebfa	f2c9076c	e434d83c	1997c987	c6848b7e
02897a5d	0b4e520a	8a14ec61	67377d21	c72000d4	67a2d68d
609e937d	68128bcb	e77828bf	20d233af	6dd74bc6	fadf2ef4
7e76cc1f	24217808	c44cb817	28fbd40d	c8d1364e	9143e9b1
22adc0b7	7b81b9ed	9d9718bf	3a9789b9	4edd9f7f	119f374d
466058cd	acba7a84	e00f106d	74b2ceaf	0548b4d2	494d7552
5bda01e3	c1d3cd95	523cd35a	e0a31c49	67b7641a	882157ea

3.3 복호화 과정

복호화 알고리즘은 암호화 알고리즘을 그대로 적용하고 단지 암호화 과정에서 사용된 확장키를 역순으로 적용하면 된다. 그런데 본 논문에서의 키 스케줄링 알고리즘은 36개의 고정된 확장키만으로 암호화 복호화를 수행하는 것이 아니라 128비트 단위로 36개의 확장키를 계속해서 만들어 낸다. Feistel Networks구조 특성대로 암호화 과정에서 사용된 확장키를 역순으로 적용해야 한다.

그래서 128비트 단위로 암호화에 사용된 마지막 32비트 라운드 키 4개와 그때 사용된 P와 Q를 임시 저장 버퍼에 저장한 다음 그 저장된 값들을 기초로 복호화 키를 만든다. 자세히 설명하면 그림 3-5에서 K_{36}, K_{37}, K_{38}, K_{39}, P_{18}, Q_{18}은 암호화 키 스케줄링 알고리즘을 라운드 수만큼 반복 수행을 통해 만들어 진 것으로 임시 저장 버퍼에 저장하고 그것은 다음 128비트 블록의 암호문을 복호 할 때 사용 할 확장키를 만들 때 사용한다. 이와 같은 과정을 암호문의 끝까지 반복 수행한다.

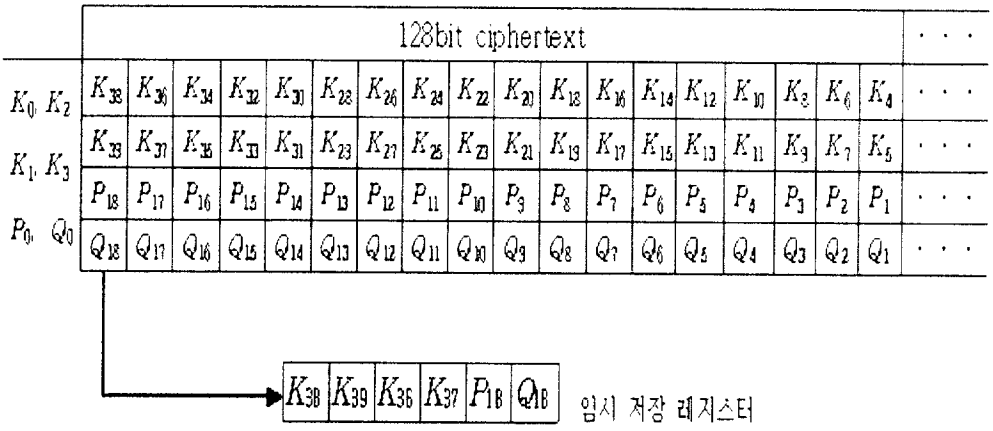


그림 3-5 복호화 과정

그리고 만들어진 K_{36}, K_{37}, K_{38}, K_{39}, P_{18}, Q_{18}을 가지고 복호화 작업에서 사용될 확장키를 만드는 알고리즘을 수행하여 한번에 두 개씩 순차 적으로 복호키를 만들어 낸다.

4. 구현 및 통계검정

Pkc128 블록 암호 알고리즘의 소프트웨어 구현은 Microsoft Visual C++ 6.0으로 암호 모드는 CBC로 구현하였으면 블록 암호 알고리즘의 안전성을 검정하는 것에 초점을 두었다. 안전성 검정은 FIPS(Federal Information Processing Standards) 140-1에서 제안된 5가지 통계 검정 빈도검정(Frequency test), 계열검정(Serial test), 포커검정(Poker test), 런검정(Run test), 자기상관검정(Autocorrelation test)을 시행하였다.

4.1 빈도검정(Frequency Test)

빈도 검정은 n-bit 2진 수열에 대하여 0과 1의 수가 일양적으로 분포하고 있는지를 검정하는 것이다. 빈도 검정은 비트 생성기에 의해 발생하는 스트림이 확률적으로 독립이고 일양적으로 분포된 확률변수를 갖는 BMS (Binary Memoryless source) 모델을 기초로 한다. 대상 수열로부터 0의 개수와 1의 개수를 구하여 이로부터 통계량을 얻는다.

통계량의 분포는 자유도 1인 χ^2-분포를 따르며 유의수준 α에 대한 기각역은 $\chi_0^2 > \chi^2\ (1,\ \alpha)$ 이다. 제안된 비트 스트림 생성기의 출력 스트림에 대한 테스트 결과 통계량 x^2=1.84519로서 5% 유의수준에 대한 통계량 $\chi_{0.05}^2 = 3.84146$을 만족함을 알 수 있다.

그림 4-1에 빈도 검정 결과를 보여주고 있다.

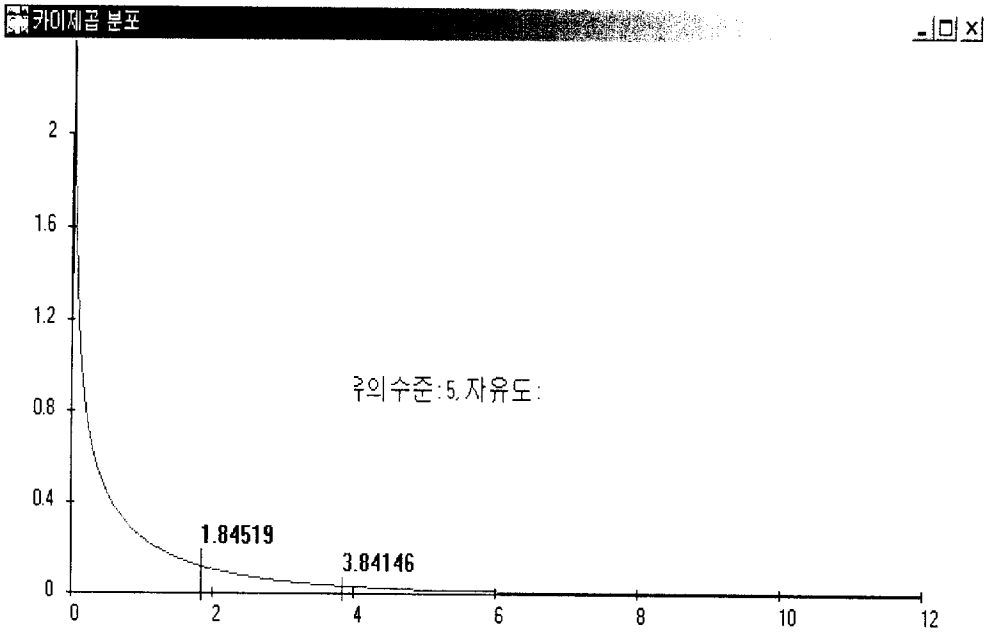


그림 4-1 빈도 검정

4.2 계열검정(Serial Test)

계열 검정은 n-bit 2진 수열에서 00, 01, 10, 11의 각 비트 쌍이 고려되는 검정이다. 즉, 2진 수열에서 한 비트가 그 다음 비트로 전이되어 가는 과정이 random 한가를 검정하는 검정법이며, 비트 스트림이 이 검정을 통과하면 각 비트가 그 앞의 비트와 독립임을 제시하여 준다.

통계량의 분포는 자유도 2인 χ^2-분포를 따르며 유의수준 α에 대한 기각역은 $\chi_0^2 > \chi^2(2, \alpha)$이다. 제안된 알고리즘의 출력 비트 스트림에 대한 테스트 결과 통계량 x^2=3.80291로서 5% 유의수준에 대한 통계량

$\chi^2_{0.05} = 5.99148$를 만족함을 알 수 있다.

그림 4-2에 계열 검정 결과를 보여주고 있다.

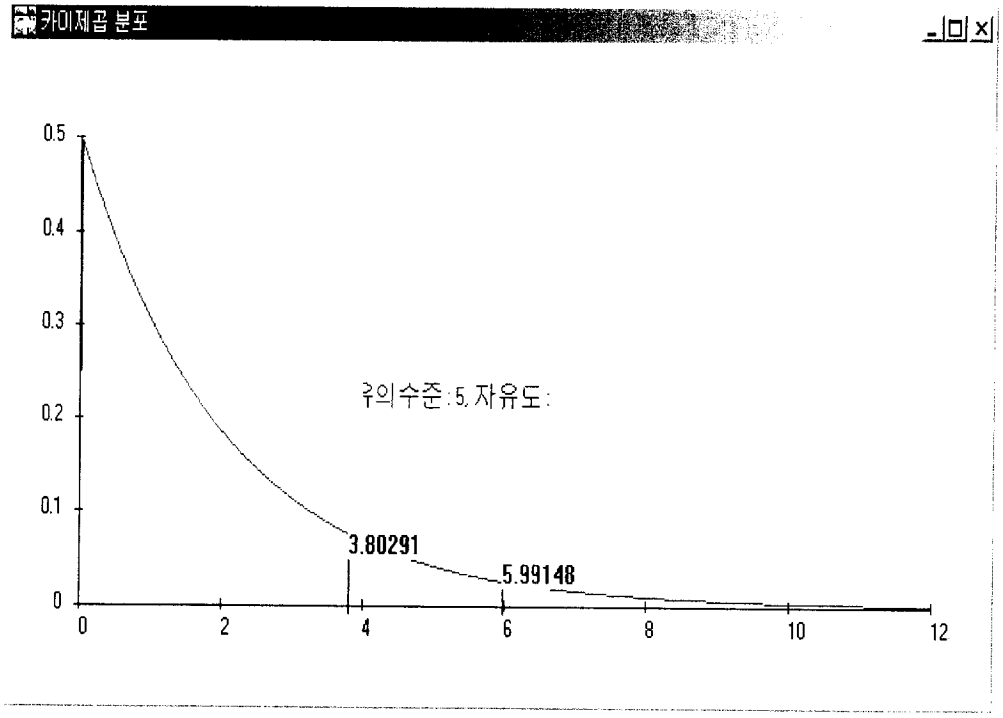


그림 4-2 계열 검정

4.3 포커검정(Poker Test)

포커 검정은 n-bit 2진 수열에서 임의의 m-bit 패턴을 고려하는 검정이다. 주어진 비트 스트림을 k개의 중첩되지 않는 m-bit 길이로 나누고 i번째 비트 패턴의 출현 횟수를 n_i라고 둔다. 이때 2^m개의 서로 다른 패턴이 존재한다. 검정의 목적은 주어진 스트림의 각 m-bit 패턴이 동일하게 나타나는지를 결정하는 것이다.

통계량의 분포는 자유도 2^m-1인 χ^2-분포를 따르며 유의수준 α에 대한 기각역은 $\chi_0^2 > \chi^2\ (2^m-1,\ \alpha)$이다. 테스트 결과 $m=8$에 대하여 통계량은 x^2=246.23449로서 5% 유의수준에 대한 통계량 $\chi_{0.05}^2=293.14777$을 만족함을 알 수 있다.

그림 4-3에 포커 검정 결과를 보여주고 있다.

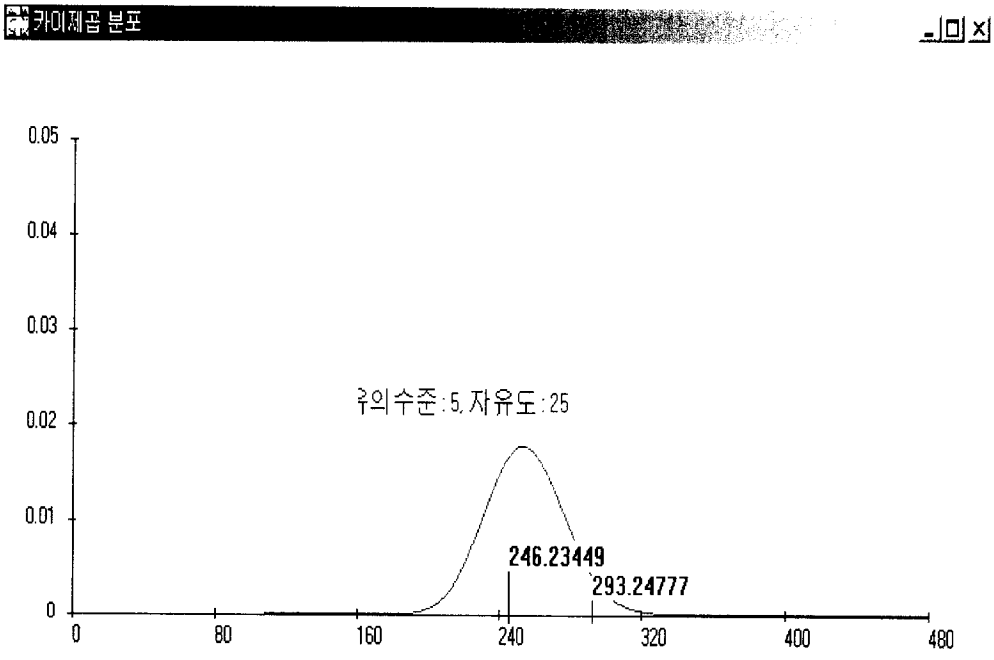


그림 4-3 포커 검정

4.4 런 검정(Run Test)

런 검정은 n-bit 2진 비트 스트림에 대하여 다양한 길이의 런(run)들의 수가 이상적인 난수열에서 기대되는 것처럼 고려되는가를 결정하기 위한 검정이다. 런(run)이란 비트 스트림에서 0이나 1이 연속하여 나타나는 부분 스트림을 말한다.

통계량의 분포는 자유도 $2(L-1)$인 χ^2-분포를 따르며 유의수준 α에 대한 기각역은 $\chi_0^2 > \chi^2\ (2(L-1),\ \alpha)$이다. 여기서 L은 런의 길이를 의미한다. L = 8에 대한 테스트 결과 통계량 x^2=18.230933로서 5% 유의수준에 대한 통계량 $\chi_{0.05}^2 = 24.995$에 대하여 만족한다.

그림 4-4에 런 검정 결과를 보여주고 있다.

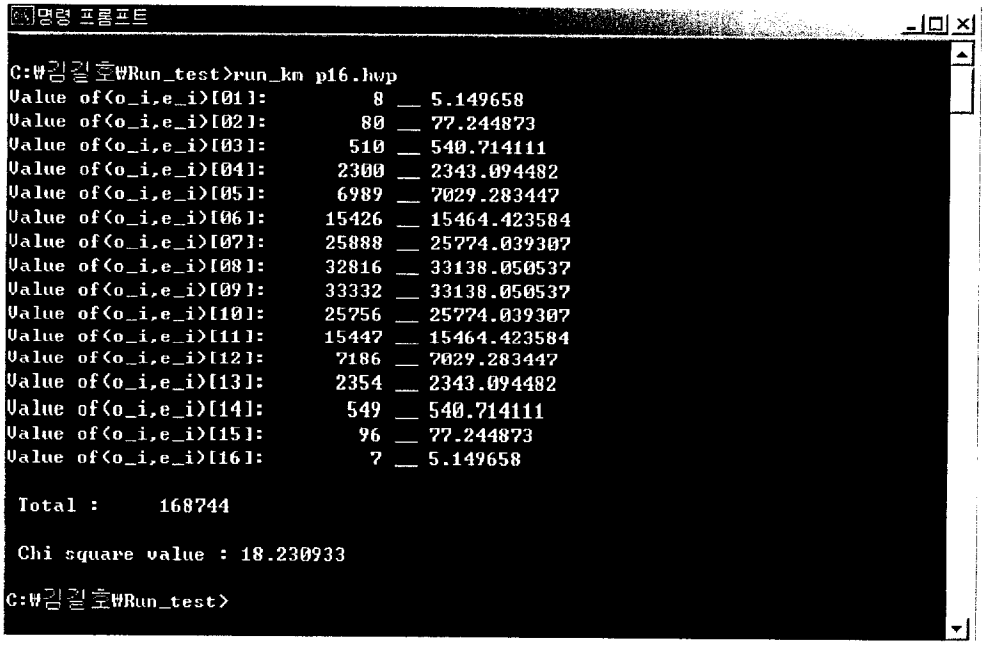


그림 4-4 런 검정

4.5 자기상관검정(Autocorrelation Test)

자기 상관 검정은 n-bit 2진 비트 스트림 $s^n=(s_0, s_1, \cdots, s_{n-1})$에 대하여, s^n에서 d-bit 만큼 전이시켜 생성한 비트 스트림 s^{n+d}와의 상관관계를 조사하는 검정이다. 여기서, d는 고정된 상수이며 범위는 $1 \leq d \leq \lfloor n/2 \rfloor$ 이다.

통계량의 분포는 자유도 1인 χ^2-분포를 따르며 유의수준 α에 대한 기각역은 $\chi_0^2 > \chi^2(1, \alpha)$이다. 제안한 의사 랜덤 비트 스트림 생성기에 대한 테스트 결과 $d=8$에 대하여 통계량 x^2=3.48179로서 5% 유의수준에 대한 통계량 $\chi_{0.05}^2=3.841$을 만족하고 있다.

그림 4-5에 자기상관 검정 결과를 보여주고 있다.

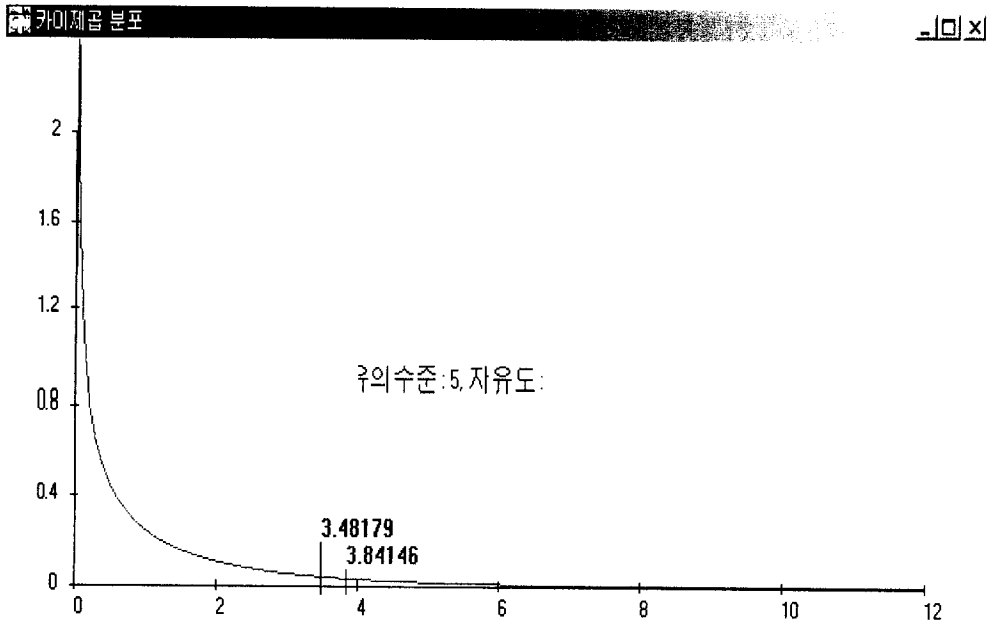


그림 4-5 자기상관 검정

16라운드를 수행 후 5가지 통계 검정 결과를 간략히 표 4-1에 요약하였다.

표 4-1 통계 검정 결과

Test	Threshold Value	Result	Remark
Frequency	3.84146	1.84519	**0** : 1,348,836 **1** : 1,351,068
Serial	5.99148	3.80291	**00** : 673,285 **01** : 675,551 **10** : 675,551 **11** : 675,516
Poker	293.24777	246.23449	**m** = 8
Run	24.995	18.230933	
Autocorrelation	3.84146	3.48179	**d** = 8
Total Bit : 2,699,904			

5. 결 론

Pkc128 블록 암호 알고리즘은 데이터 의존 회전 기법과 프로그램 가능 셀룰라 오토마타 기법을 사용한 하드웨어 구현을 위한 암호 알고리즘이다. 128비트 블록크기에 Feistel Networks 구조로 16라운드를 수행하고 키는 128비트 이상 가변이며, 한 블록 당 확장키는 라운드마다 2개씩 그리고 라운드 전후에 2개씩 사용하여 총 36개를 사용한다.

본 논문에서 제안한 알고리즘은 스마트 카드와 같은 소규모 하드웨어 상에서 암호 기능을 구현하기 위하여 하드웨어 면적을 줄이면서 안전성도 검증하였다. 또한 하드웨어 면적을 줄이기 위해 메모리를 없애고 확장키는 암호화 및 복호화 시에 필요한 확장키를 실시간적으로 생성하는 것이 가능하도록 하였다.

그리고 제안한 알고리즘의 안전성을 검정하기 위해서 FIPS140-1에서 제안된 5가지 통계검정 빈도검정, 계열검정, 포커검정, 런 검정, 자기상관검정을 시행하여 모두 통과하여 암호 알고리즘의 안전성을 검증하였다.

향후 연구과제는 제안한 알고리즘의 안전성 검증은 5가지 통계검정만 시행하였는데 좀더 고급적인 수학적 모델인 선형 분석 및 차분 분석을 통한 안전성 검증과 하드웨어로 제작하여 다른 블록 암호 알고리즘들과 성능을 종합적으로 평가하고, 그리고 본 논문에서 제안된 확장키를 만드는 키 스케줄링 알고리즘은 무한히 확장키를 만드는 것은 아니다. 어떤 주기가 있을 것으로 예상되는데 현재로서는 그 주기를 계산 할 방법을 모르고 있으며, 향후 확장키 생성 주기를 계산하여 확장키 생성 알고리즘의 안전성을 검증하는 것을 연구 과제로 남겨 놓았다.

참고문헌

[1] NIST, "Advanced Encryption Standard Development Effort." http://csrc.nist.gov/encryption/aes.

[2] Joan Daemen, Vincent Rijmen, "AES Proposal: Rijndael", 1999

[3] M.Riaz and H.M.Heys, "The FPGA Implementation of the RC6 and CAST256 Encryption Algorithm",1999

[4] Scott Contini, Ronald L.Rivest, M.J.B. Robshaw and Y.L.Yin," The Security of the RC6 Block Cipher", 1998

[5] S. Nandi, B. K. Kar, and P. Pal Chaudhuri "Theory and Applications of Cellular Automata in Crytography", 1994

[6] FIPS180-1, "Secure hash standard hash standard", Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/Nist, 1995

[7] 박창섭 저 "암호이론과 보안" 대영사 1999

부 록

Pkc128 블록 암호 알고리즘 C 소스 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
#include <time.h>

typedef unsigned char word8 ;
typedef unsigned short int word16;
typedef unsigned long int word32;

/*
#define  TTEST
*/

      /* number of rounds    */
#define NROUND    16

       /* magic constants     */
#define     P        0xb7e15163
#define     Q        0x9e3779b9

/* Rotation operators. x must be unsigned, to get logical right shift*/
#define ROTL(x,y) (((x)<<(y)) | ((x)>>(32-(y))))

#define GetB0(A)   ( (word8)((A)      ) & 0x0ff)
#define GetB1(A)   ( (word8)((A)>> 8) & 0x0ff)
#define GetB2(A)   ( (word8)((A)>>16) & 0x0ff)
#define GetB3(A)   ( (word8)((A)>>24) & 0x0ff)

#define ercround(L0, L1, R0, R1, K) {                          \
    u0 = ROTL(L0 ^ (K)[0], L1) ;                               \
    v0 = ROTL(L1 ^ (K)[1], L0) ;                               \
```

```
    u1 = L1 ^ (L1<<1 & (K)[1]) ^ (L1>>1 & !(K)[1]) ;        \
    v1 = L0 ^ (L0<<1 & (K)[0]) ^ (L0>>1 & !(K)[0]) ;        \
    R0 = R0 ^ ROTL(u0 ^ u1, v0) ;                           \
    R1 = R1 ^ ROTL(v0 ^ v1, u0) ;                           \
    }

word32 *rkey ;          /* expanded key */

word32 *buf4 ;
word8 *bufc ;

void encrypt(word32 *pt)
{
word32 u0, u1, v0, v1 ;
word32 L0, L1, R0, R1 ;
int    ia ;

  L0 = pt[0] ^ ROTL(rkey[0], rkey[1]) ;
  L1 = pt[1] ^ ROTL(rkey[1], rkey[0]) ;
  R0 = pt[2] ;
  R1 = pt[3] ;

  for (ia=0; ia<NROUND; ia+=2 )
     { ercround(L0, L1, R0, R1, rkey+ia*2+2) ;
       ercround(R0, R1, L0, L1, rkey+ia*2+4) ;
     }

  pt[0] = R0 ^ ROTL(rkey[NROUND*2+2], rkey[NROUND*2+3]) ;
  pt[1] = R1 ^ ROTL(rkey[NROUND*2+3], rkey[NROUND*2+2]) ;
  pt[2] = L0 ;
  pt[3] = L1 ;
}

void decrypt(word32 *pt)
{
word32 u0, u1, v0, v1 ;
word32 L0, L1, R0, R1 ;
int    ia ;
```

```
    L0 = pt[0] ^ ROTL(rkey[NROUND*2+2], rkey[NROUND*2+3]) ;
    L1 = pt[1] ^ ROTL(rkey[NROUND*2+3], rkey[NROUND*2+2]) ;
    R0 = pt[2] ;
    R1 = pt[3] ;

    for (ia=NROUND-2; ia>=0; ia=ia-2)
        { ercround(L0, L1, R0, R1, rkey+ia*2+4) ;
          ercround(R0, R1, L0, L1, rkey+ia*2+2) ;
        }

    pt[0] = R0 ^ ROTL(rkey[0], rkey[1]) ;
    pt[1] = R1 ^ ROTL(rkey[1], rkey[0]) ;
    pt[2] = L0 ;
    pt[3] = L1 ;
}

void keysched(word32 *key)
{
word32 A, B ;
int    ia, ib, ic ;

    for (rkey[0]=P,ia=1; ia<(NROUND*2+4); ia++) rkey[ia] = rkey[ia-1]+Q ;
    for (A=B=ia=ib=ic=0 ; ic<(NROUND*6+12) ; ic++,ib=ib&3)
          { A = rkey[ia] = ROTL( rkey[ia] + A + B, 3 );
            B = key[ib] = ROTL( key[ib] + A + B, ( A + B ) ) ;
            ia++ ;
            ib++ ;
            if (ia==(NROUND*2+4)) ia=0 ;
          }
}

void makekey(char *string, word32 fsize)
{
int ia, ib, len = strlen(string) ;
word8 *keyp ;

    buf4 = (word32 *)malloc(128) ;
    bufc = (word8 *)buf4 ;
```

```
    rkey = (word32 *)malloc(NROUND*8+16) ;

    for ( ia=0, keyp=bufc ; ia<128 ; ia++) *keyp++ = ia & 0x01f ;

    if ( len > 128 ) len = 128 ;
    for ( keyp=bufc ; len > 0 ; len--)
       { *keyp++ = *string ;
        *string++ = 0 ;
       }

    buf4[8] ^= fsize ;

    keysched(buf4) ;
    encrypt(buf4+4) ;

    for(ia=2 ; ia < 8 ; ia++)
       { for (ib=0 ; ib < 4 ; ib++ )
                buf4[ia*4+ib] ^= buf4[ia*4-4+ib] ;
        encrypt(buf4+ia*4) ;
       }

    keysched(buf4+28) ;
}

int rekey()
{
int ia, ib ;

    encrypt(buf4+8) ;
    for (ia=3 ; ia < 8 ; ia++)
       { for (ib=0 ; ib < 4 ; ib++ ) buf4[ia*4+ib] ^= buf4[ia*4-4+ib] ;
         encrypt(buf4+ia*4) ;
       }
    keysched(buf4+28) ;
    return ((int)bufc[48] << 2) ;
}

void flushkey()
{
```

```
int ia ;

    for(ia=0 ; ia < (NROUND*2+4) ; ia++) rkey[ia]=0 ;
    for(ia=0 ; ia < 32 ; ia++) buf4[ia]=0 ;
    free(rkey) ;
    free(buf4) ;
}

void blockenc(char *string, char *input, char *output)
{
FILE *fp1, *fp2 ;
int ia, section, bytecnt, eoflag ;
word8 *abufp, *ctp ;
word32 fsize[2] ;

#ifdef TTEST
time_t lt ;
#endif

    if((fp1=fopen(input, "rb")) == NULL)
        { printf("Source file open error !!!\n");
          exit(1);
        }
    if((fp2=fopen(output, "wb")) == NULL)
        { printf("Destination file open error !!!\n");
          fclose(fp1) ;
          exit(1);
        }

    fseek(fp1, 0, 2) ;
    fgetpos(fp1, fsize) ;
    fseek(fp1, 0, 0) ;

    makekey(string, fsize[0]);
    section = (int)bufc[48] << 2 ;

#ifdef TTEST
    lt=time(NULL);
#endif
```

```
while ( !feof(fp1) ) {

    if (++section > 1088) section=rekey() ;

    for(bytecnt=0, abufp=bufc ; bytecnt < 16 ; bytecnt++)
        {
          *abufp++ = fgetc(fp1) ;
          if ( ferror(fp1) )
                { printf("Source file reading error !!!\n");
                  flushkey();
                  fclose(fp1);
                  fclose(fp2);
                  exit(1) ;
                }
          eoflag = feof(fp1) ;
          if ( eoflag != 0 ) break ;
        }

   if (eoflag == 0)
        {  for (ia=0, abufp=bufc, ctp=bufc+32; ia<16 ; ia++)
                 *abufp++ ^= *ctp++ ;

           encrypt(buf4);

           for (ia=0, abufp=bufc, ctp=bufc+32 ; ia<16 ; ia++)
                *ctp++ = *abufp++ ;
        }
        else
           {
              encrypt(buf4+8);
              for (ia=0, abufp=bufc, ctp=bufc+32 ; ia<16 ; ia++)
                  *abufp++ ^= *ctp++ ;
           }

   for(ia=0, abufp=bufc ; ia < 16 ; ia++)
       { if ( bytecnt == 0 ) break ;
         bytecnt-- ;
         putc(*abufp++, fp2) ;
```

```
            if ( ferror(fp2) )
                { printf("Destination file writing error !!!\n");
                  flushkey();
                  fclose(fp1);
                  fclose(fp2);
                  exit(1) ;
                }
            }
        }

#ifdef TTEST
    printf("\n Elapsed time = %d sec\n", time(NULL)-lt);
#endif

    flushkey();
    fclose(fp1);
    fclose(fp2);

    printf("\nSuccessfully encoded\n") ;
}

void blockdec(char *string, char *input, char *output)
{
FILE *fp1, *fp2;
int ia, section, bytecnt, eoflag ;
word8 *abufp, *ctp, *tmpp ;
word32 fsize[2] ;

#ifdef TTEST
time_t lt ;
#endif

    if((fp1=fopen(input, "rb")) == NULL)
        { printf("Source file open error !!!\n");
          exit(1);
        }
    if((fp2=fopen(output, "wb")) == NULL)
        { printf("Destination file open error !!!\n");
          exit(1);
```

```
        }

    fseek(fp1, 0, 2) ;
    fgetpos(fp1, fsize) ;
    fseek(fp1, 0, 0) ;

    makekey(string, fsize[0]);
    section = (int)bufc[48] << 2 ;

#ifdef TTEST
    lt=time(NULL);
#endif

    while ( !feof(fp1)) {

        if (++section > 1088) section=rekey() ;

        for(bytecnt=0, abufp=bufc, tmpp=bufc+16 ; bytecnt < 16 ; bytecnt++)
            {
              *abufp++ = *tmpp++ = fgetc(fp1) ;
              if ( ferror(fp1) )
                    { printf("Source file reading error !!!\n");
                      flushkey();
                      fclose(fp1);
                      fclose(fp2);
                      exit(1) ;
                    }
              eoflag = feof(fp1) ;
              if ( eoflag != 0 ) break ;
            }

        if (eoflag == 0)
            {
                decrypt(buf4);

                for (ia=0, abufp=bufc, ctp=bufc+32, tmpp=bufc+16 ; ia<16 ; ia++)
                    { *abufp++ ^= *ctp ;
                      *ctp++ = *tmpp++ ;
                    }
```

```
            }
            else
                {
                    encrypt(buf4+8);
                    for (ia=0, abufp=bufc, ctp=bufc+32 ; ia<16 ; ia++)
                        *abufp++ ^= *ctp++ ;
                }

        for(ia=0, abufp=bufc ; ia < 16 ; ia++)
            { if ( bytecnt == 0 ) break ;
              bytecnt-- ;
              putc(*abufp++, fp2) ;
              if ( ferror(fp2) )
                    { printf("Destination file writing error !!!\n");
                      flushkey();
                      fclose(fp1);
                      fclose(fp2);
                      exit(1) ;
                    }
            }
      }

#ifdef TTEST
    printf("\n Elapsed time = %d sec\n", time(NULL)-lt);
#endif

    flushkey();
    fclose(fp1);
    fclose(fp2);

    printf("\nSuccessfully decoded\n") ;
}

void main(argc, argv)
int argc;
char *argv[];
{
    printf("\n*** PKC128 file encrypt/decrypt version 1.0 ***\n");
    printf("          Algorithm          -- PuKyung 128 bit Cipher\n");
```

```
    printf("            128 bit key, 128 bit block, %d round\n", NROUND);
    printf("        Cipher mode      -- CBC\n");
    printf("        Maximum key phrase -- 128 character\n\n");

    if (argc>1) argv[1][1] |= 0x20 ;

    if ((argc==5) && (argv[1][0] == '-' && argv[1][1] == 'e'))
        blockenc(argv[2], argv[3], argv[4]);
      else if ((argc==5) && (argv[1][0] == '-' && argv[1][1] == 'd'))
              blockdec(argv[2], argv[3], argv[4]);
          else {
                  printf("\nBad Command or Option...\n");
                  printf("Using : pkc128 [ -d | -e ] key infile outfile\n");
                }
}
```