128비트 블록 암호알고리즘 (SEED) 개발 및 분석 보고서

- A Design and Analysis of SEED -

Version 1.1

2003. 10. 23

한국정보보호진흥원

목 차

그림 목차

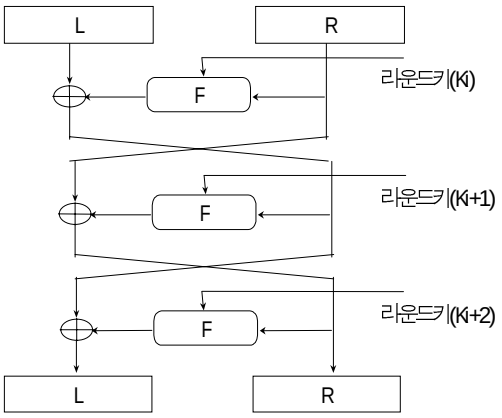
표 목차

제 1 장 서론

암호알고리즘은 암·복호화에 사용되는 키의 특성에 따라 암·복호화 키가 같은 대칭키 암호알고리즘과 암·복호화 키가 다른 공개키 암호알고리즘으로 크게 대별하여 볼 수 있으며, 대칭키 암호알고리즘은 메시지 처리 형식에 따라 스트림 암호알고리즘과 블록 암호알고리즘으로 나누어 볼 수 있다.

개발된 SEED는 대칭키 암호알고리즘으로써, 블록 단위로 메시지를 처리하는 블록 암호알고리즘이다.

대칭키 블록 암호알고리즘은 기밀성을 제공하는 암호시스템의 중요 요소이다. n비트 블록 암호알고리즘이란 고정된 n비트 평문을 같은 길이의 n비트 암호문으로 바꾸는 함수를 말한다(n비트 : 블록 크기). 이러한 변형 과정에 암호키가 작용하여 암호화와 복호화를 수행한다.



(그림 1-1) Feistel 구조

대부분의 블록 암호알고리즘은 Feistel 구조로 설계되고 있다(예, DES, FEAL, LOKI, MISTY, Blowfish, CAST, Twofish 등). Feistel 구조란 각 t비트인 L_0, R_0 블록으로 이루어진 2t비트 평문 블록 (L_0, R_0)을 r라운드($r \geq 1$)를 거쳐 암호문 (L_r, R_r)을 내는 반복 구조를 말한다. 반복 구조란 평문 블록이 몇 번의 라운드를 거쳐 암호화를 수행하는 것을 말하고, 라운드 i

$(1 \le i \le r)$란 암호키 K로부터 유도된 각 서브키 K_i (또는, 라운드 키라 불림)를 중요 입력으로 하는 $L_i = R_{i-1}$, $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$를 통해 $(L_{i-1}, R_{i-1}) \xrightarrow{K_i} (L_i, R_i)$로 바꾸어 주는 함수를 말한다. 또한, 전체 알고리즘의 라운드 수는 요구되는 보안 강도와 수행 효율성의 상호 절충적 관계에서 결정된다. 보통 Feistel 구조는 3라운드 이상이며, 짝수 라운드로 구성된다. 이러한 Feistel 구조는 ①라운드 함수에 관계없이 역변환이 가능하며(즉, 암복호화 과정이 같음), ②두 번의 수행으로 블록간의 완전한 Diffusion이 이루어지며, ③알고리즘의 수행속도가 빠르고, ④H/W 및 S/W 구현이 용이하고, ⑤아직 구조상의 문제점이 발견되고 있지 않다는 장점을 지니고 있다.

암호알고리즘의 분석은 크게 전수조사 공격과 알고리즘 구조의 취약성 분석으로 나누어 볼 수 있으며, 현재 전수조사 공격을 피하기 위해서는 블록크기 및 키길이가 128비트 이상이면 충분한 안전도를 제공한다고 간주된다(참고 : DES는 56비트의 키길이를 가지며, '98년 7월 $250,000 칩으로 56시간만에 해독됨 ; RSA DES Challenge II). 이는 현재 계산 능력과 알고리즘의 사용기간 등을 고려하여 결정되며, 비용 효율적 안전도를 제공하는 기준이 된다. 또한 블록 암호알고리즘의 구조상의 취약성을 분석하는 방법에는 현재 가장 강력한 수단으로 알려져 있는 DC(차분해독법) 및 LC(선형해독법)에 안전하여야 하고, 키생성 알고리즘에 특정한 취약성이 없어야 한다.

SEED는 이러한 설계기준을 바탕으로 설계되었으며, 제 2 장에서는 SEED의 소개, 제 3 장에서는 안전성 분석, 제 4 장에서는 효율성 등의 분석 결과를 기술한다.

제 2 장 SEED의 소개

제 1 절 개발 현황

1. 추진 현황

- 1997년 9월 ~ 1998년 9월
  - 128비트 블록암호알고리즘(SEED) 초안 개발 완료

- 1998년 9월 2일 ~ 1998년 9월 30일
  - 암호알고리즘 초안 1차 자체 평가 완료

- 1998년 10월 29일 ~ 1999년 2월 15일
  - 공개 검증을 위한 의견 수렴 공고(알고리즘 공개)

- 1999년 3월 30일
  - 표준 최종안 확정

- 1999년 9월 28일
  - 표준 제정 완료(TTAS KO-12.0004)

제 2 절 설계 기준

1. 전체구조

- o 데이터 처리단위 : 8, 16, 32비트 모두 가능
- o 암·복호화 방식 : 블록 암호방식
- o 입·출력문의 크기 : 128비트
- o 입력키의 크기 : 128비트
- o 안전성 : DC/LC에 대하여 안전하도록 설계
- o 효율성 : 암·복호화 속도는 3중 DES이상
- o 구조 : Feistel 구조
- o 내부함수 : SPN 구조이며, 비선형함수를 Look-up 테이블로 변형하여 사용
- o 라운드 수 : 안전성은 키전수 조사공격에 필요한 계산복잡도 및 평문·암호문 쌍(2^{128})이하가 되지 않아야 하며, 효율성 요구조건을 만족하여야 함
- o 키생성 알고리즘 : 알고리즘의 라운드 동작과 동시에 암·복호화 라운드 키가 생성될 수 있도록 설계

2. 안전성에 대한 설계조건

- o 안전성이 증명가능한 구조로 설계
- o 차분해독법(Differential Cryptanalysis, DC)에 대하여 안전하여야 한다.
- o 선형해독법(Linear Cryptanalysis, LC)에 대하여 안전하여야 한다.
- o 기타 공격방식(Higher Order DC, Related Key Attack 등)이 적용되기 어렵게 한다.

- Higher Order DC에 강하기 위하여 대수적 차수가 3이상인 부울함수를 사용한다.
- Related Key Attack에 강하기 위하여 Key Schedule에 비선형 함수를 사용한다.

3. 효율성에 대한 설계조건

o S/W로 구현시 3중 DES보다 고속이어야 한다

4. SEED의 일반적 특징

o 데이터 처리단위 : 8, 16, 32비트 모두 가능
o 암·복호화 방식 : 블록암호방식
o 입·출력문의 크기 : 128비트
o 입력키의 크기 : 128비트
o 라운드 수 : 16라운드
o 구조 : Feistel 구조
o 내부함수
  - 연산은 ⊞, ⊕만 사용
  - 2개의 안전성이 입증된 S-Box 사용
  - DC, LC에 대한 이론적 안전성 증명 가능
  - 내부함수 F의 구조는 DES, MISTY등과 비교하여 우수함

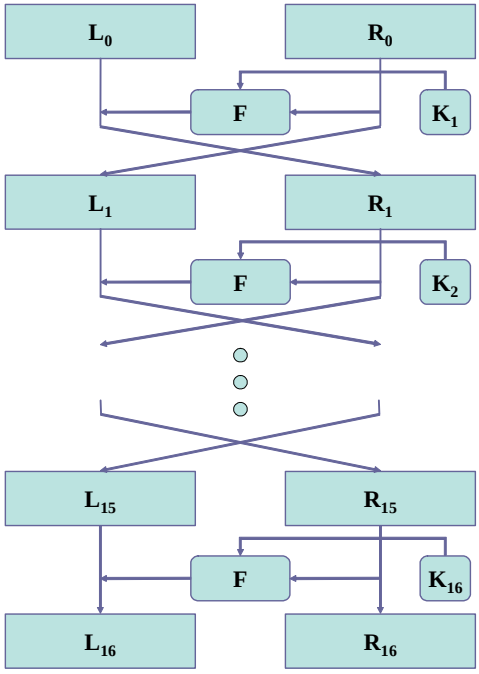
제 3 절 SEED 초안

1. 연산기호의 정의

- $a \oplus b$: a bit-wise Exclusive-Or b
- $a \boxplus b$: $(a + b) \mod 2^{32}$
- $a \,\&\, b$: a bit-wise AND b
- $X^{\ll s}$: X를 s비트 만큼 왼쪽으로 순환 이동하는 연산
- $X^{\gg s}$: X를 s비트 만큼 오른쪽으로 순환 이동하는 연산
- L_i : i 라운드에서 출력된 왼쪽 메시지 블록(64비트)
- R_i : i 라운드에서 출력된 오른쪽 메시지 블록(64비트)
- $K_i = (K_{i,0}, K_{i,1})$: i 라운드의 라운드키(64비트)
- $K_{i,0}$: i 라운드 F 함수의 오른쪽 입력키(32비트)
- $K_{i,1}$: i 라운드 F 함수의 왼쪽 입력키(32비트)
- $X = (X_3 \| X_2 \| X_1 \| X_0)$: G함수의 입력값(32비트)
- $Y = (Y_3 \| Y_2 \| Y_1 \| Y_0)$: G함수에서 S-Box(S_1, S_2)의 출력값(32비트)
- $Z = (Z_3 \| Z_2 \| Z_1 \| Z_0)$: G함수의 출력값(32비트)
- m_i : 상수
- KC_i : 라운드 키 생성과정에서 사용되는 i+1 라운드 상수

2. 알고리즘 전체 구성도

알고리즘 전체 구조는 Feistel 구조로 이루어져 있으며, 128비트의 평문 블록단위당 128비트 키로부터 생성된 64비트의 라운드 키(16개)를 입력으로 받아 총 16라운드를 거쳐 128비트 암호문 블록을 출력으로 낸다. 다음 (그림 2-1)은 전체구조를 도식화한 것이다.



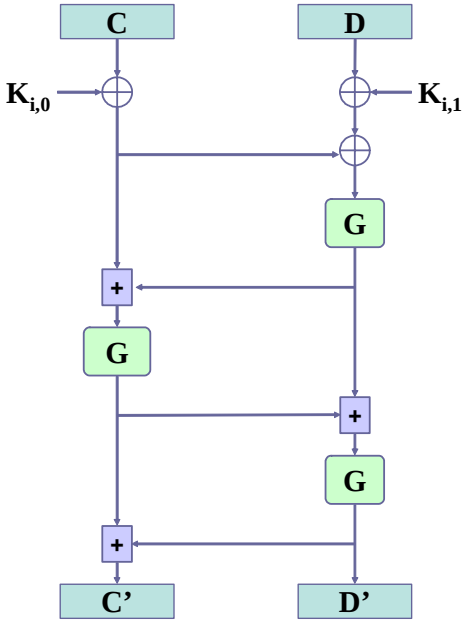
(그림 2-1) SEED 전체 구조도

3. F 함수

Feistel 구조의 블록 암호알고리즘은 F 함수의 특성에 따라 구분할 수 있다. SEED의 F 함수는 수정된 64비트 Feistel 암호알고리즘으로 구성된다. F 함수는 각 32비트 2개의 블록(C, D)을 입력으로 받아, 32비트 2개의 블록(C′, D′)를 출력한다. 즉, 암호화 과정에서 64비트 (C, D)와 64비트 키 K_i(=($K_{i,0}$; $K_{i,1}$))를 F 함수의 입력으로 처리하여 (C′, D′)을 출력으로 낸다.

$$C' = G[G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} \boxplus (C \oplus K_{i,0})] \boxplus G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\}] \boxplus G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} \boxplus (C \oplus K_{i,0})]$$

$$D' = G[G[G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\} \boxplus (C \oplus K_{i,0})] \boxplus G\{(C \oplus K_{i,0}) \oplus (D \oplus K_{i,1})\}]$$

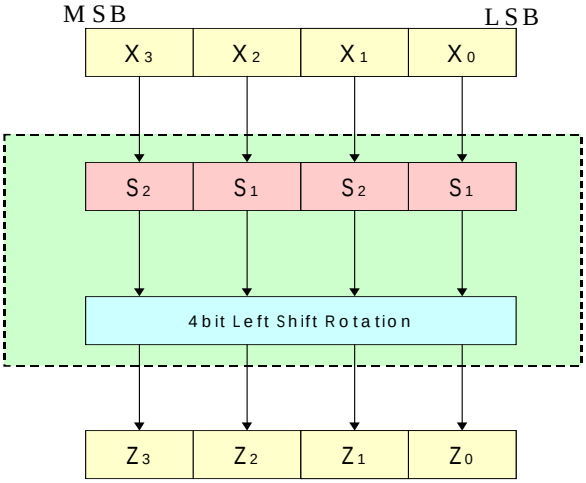


(그림 2-2) F-함수 구조도

4. G 함수

G 함수는 매우 좋은 특성을 갖는 두 개의 8비트 S-Box를 이용하여 입력의 각 바이트를 비선형 변환 후, 그 결과 32비트를 4비트 왼쪽 회전이동한 후 출력한다. 즉, G 함수의 입력 32비트를 4개의 8비트 블록 (X_3, X_2, X_1, X_0)으로 분할하여 2개의 S-Box를 (S_2, S_1, S_2, S_1) 순서로 적용시켜 (Y_3, Y_2, Y_1, Y_0)를 생성하고, 4비트 왼쪽 회전이동 후, 4개의 8비트 블록 (Z_3, Z_2, Z_1, Z_0)을 생성한다.

$$\begin{array}{ll} Y_3 = (Y_{31}, Y_{30}) = S_2(X_3) & Z_3 = (Y_{30}, Y_{21}) \\ Y_2 = (Y_{21}, Y_{20}) = S_1(X_2) & Z_2 = (Y_{20}, Y_{11}) \\ Y_1 = (Y_{11}, Y_{10}) = S_2(X_1) \quad , & Z_1 = (Y_{10}, Y_{01}) \\ Y_0 = (Y_{01}, Y_{00}) = S_1(X_0) & Z_0 = (Y_{00}, Y_{31}) \end{array}$$



(그림 2-3) G-함수 구조도

5. S-Box들에 사용된 부울함수

전단사함수 $x^n, 0 \le n \le 255$에서 DC 및 LC 특성(DC 및 LC 확률이 2^{-6})이 가장 우수한 2개의 $n = 247, 251$을 선택하고, $GF(2^8)$상에서의 지수승을 구하기 위해 $GF(2^8)$상에서의 모든 원소를 원시원소(primitive element) a의 멱승으로 표현하자.

(사용된 원시원소는 $a = p(x) = x^8 + x^6 + x^5 + x + 1$)

가. 선형변환

지수함수의 입력 $x=0$이 출력 0, $x=1$이 출력 1으로 고정되는 것을 방지하기 위하여, 지수함수를 $S(x) = (A \cdot x^n) \oplus b$로 선형변환한다.

$$A = \begin{matrix} \text{MSB} & & & & & & & \text{LSB} \end{matrix}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

각 S-Box에 사용된 행렬 $A^{(1)}$, $A^{(2)}$는 A의 행을 교환하여 사용한다.

($A^{(1)} = A$의 2행과 3행을 교환, 4행과 6행을 교환

$A^{(2)} = A$의 1행과 5행을 교환, 6행과 7행을 교환)

A는 nonsingular matrix이므로, 서로 다른 2개의 입력 x, x'에 대하여 $Ax \neq Ax'$ 이다.

$b_1 = (1,0,1,0,1,0,0,1)^T = 169, \quad b_2 = (0,0,1,1,1,0,0,0)^T = 56$

이라 하고, 두 개의 S-box $S_1(x)$, $S_2(x)$는 각각

$$\begin{cases} (A^{(1)} \cdot x^{247}) \oplus b_1 = (P_7, P_6, P_5, P_4, P_3, P_2, P_1, P_0)^T \rightarrow \\ S_1(x) = P_7 \times 2^7 + P_6 \times 2^6 + P_5 \times 2^5 + P_4 \times 2^4 + P_3 \times 2^3 + P_2 \times 2^2 + P_1 \times 2^1 + P_0 \\ \\ (A^{(2)} \cdot x^{251}) \oplus b_2 = (Q_7, Q_6, Q_5, Q_4, Q_3, Q_2, Q_1, Q_0)^T \rightarrow \\ S_2(x) = Q_7 \times 2^7 + Q_6 \times 2^6 + Q_5 \times 2^5 + Q_4 \times 2^4 + Q_3 \times 2^3 + Q_2 \times 2^2 + Q_1 \times 2^1 + Q_0 \end{cases}$$

으로 구성한다.

(단, $x = (x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0)$ 이고, x_i는 2^i의 계수임)

□ S-box의 특성은 $\Delta_f = 4$, $\Lambda_f = 16$, algebraic degree = 7이다.

부울함수를 $f: Z_2^8 \rightarrow Z_2^8$ 라 하면,

- $D_f(a,b) = \{x \in Z_2^8 \mid f(x \oplus a) \oplus f(x) = b\}$, $a \neq 0$이고,

$\Delta_f(a,b) = \max_{a,b\neq 0}|D_f(a,b)|$ 일 때, $\Delta_f = 4$

- $L_f(a,b) = \{x \in Z_2^8 \mid (a \cdot x) \oplus (b \cdot f(x)) = 0\}$, $b \neq 0$이고,

$\Lambda_f(a,b) = \max_{a,b\neq 0}||L_f(a,b)| - 2^{n-1}|$ 일 때, $\Lambda_f = 16$

- algebraic degree는 7이다.

i	$S_1(i)$	i	$S_1(i)$	i	$S_1(i)$	i	$S_1(i)$	i	$S_1(i)$	i	$S_1(i)$	i	$S_1(i)$	i	$S_1(i)$
0	169	1	133	2	214	3	211	4	84	5	29	6	172	7	37
8	93	9	67	10	24	11	30	12	81	13	252	14	202	15	99
16	40	17	68	18	32	19	157	20	224	21	226	22	200	23	23
24	165	25	143	26	3	27	123	28	187	29	19	30	210	31	238
32	112	33	140	34	63	35	168	36	50	37	221	38	246	39	116
40	236	41	149	42	11	43	87	44	92	45	91	46	189	47	1
48	36	49	28	50	115	51	152	52	16	53	204	54	242	55	217
56	44	57	231	58	114	59	131	60	155	61	209	62	134	63	201
64	96	65	80	66	163	67	235	68	13	69	182	70	158	71	79
72	183	73	90	74	198	75	120	76	166	77	18	78	175	79	213
80	97	81	195	82	180	83	65	84	82	85	125	86	141	87	8
88	31	89	153	90	0	91	25	92	4	93	83	94	247	95	225
96	253	97	118	98	47	99	39	100	176	101	139	102	14	103	171
104	162	105	110	106	147	107	77	108	105	109	124	110	9	111	10
112	191	113	239	114	243	115	197	116	135	117	20	118	254	119	100
120	222	121	46	122	75	123	26	124	6	125	33	126	107	127	102
128	2	129	245	130	146	131	138	132	12	133	179	134	126	135	208
136	122	137	71	138	150	139	229	140	38	141	128	142	173	143	223
144	161	145	48	146	55	147	174	148	54	149	21	150	34	151	56
152	244	153	167	154	69	155	76	156	129	157	233	158	132	159	151
160	53	161	203	162	206	163	60	164	113	165	17	166	199	167	137
168	117	169	251	170	218	171	248	172	148	173	89	174	130	175	196
176	255	177	73	178	57	179	103	180	192	181	207	182	215	183	184
184	15	185	142	186	66	187	35	188	145	189	108	190	219	191	164
192	52	193	241	194	72	195	194	196	111	197	61	198	45	199	64
200	190	201	62	202	188	203	193	204	170	205	186	206	78	207	85
208	59	209	220	210	104	211	127	212	156	213	216	214	74	215	86
216	119	217	160	218	237	219	70	220	181	221	43	222	101	223	250
224	227	225	185	226	177	227	159	228	94	229	249	230	230	231	178
232	49	233	234	234	109	235	95	236	228	237	240	238	205	239	136
240	22	241	58	242	88	243	212	244	98	245	41	246	7	247	51
248	232	249	27	250	5	251	121	252	144	253	106	254	42	255	154

(표 2-1) S_1-Box

i	$S_2(i)$	i	$S_2(i)$	i	$S_2(i)$	i	$S_2(i)$	i	$S_2(i)$	i	$S_2(i)$	i	$S_2(i)$	i	$S_2(i)$
0	56	1	232	2	45	3	166	4	207	5	222	6	179	7	184
8	175	9	96	10	85	11	199	12	68	13	111	14	107	15	91
16	195	17	98	18	51	19	181	20	41	21	160	22	226	23	167
24	211	25	145	26	17	27	6	28	28	29	188	30	54	31	75
32	239	33	136	34	108	35	168	36	23	37	196	38	22	39	244
40	194	41	69	42	225	43	214	44	63	45	61	46	142	47	152
48	40	49	78	50	246	51	62	52	165	53	249	54	13	55	223
56	216	57	43	58	102	59	122	60	39	61	47	62	241	63	114
64	66	65	212	66	65	67	192	68	115	69	103	70	172	71	139
72	247	73	173	74	128	75	31	76	202	77	44	78	170	79	52
80	210	81	11	82	238	83	233	84	93	85	148	86	24	87	248
88	87	89	174	90	8	91	197	92	19	93	205	94	134	95	185
96	255	97	125	98	193	99	49	100	245	101	138	102	106	103	177
104	209	105	32	106	215	107	2	108	34	109	4	110	104	111	113
112	7	113	219	114	157	115	153	116	97	117	190	118	230	119	89
120	221	121	81	122	144	123	220	124	154	125	163	126	171	127	208
128	129	129	15	130	71	131	26	132	227	133	236	134	141	135	191
136	150	137	123	138	92	139	162	140	161	141	99	142	35	143	77
144	200	145	158	146	156	147	58	148	12	149	46	150	186	151	110
152	159	153	90	154	242	155	146	156	243	157	73	158	120	159	204
160	21	161	251	162	112	163	117	164	127	165	53	166	16	167	3
168	100	169	109	170	198	171	116	172	213	173	180	174	234	175	9
176	118	177	25	178	254	179	64	180	18	181	224	182	189	183	5
184	250	185	1	186	240	187	42	188	94	189	169	190	86	191	67
192	133	193	20	194	137	195	155	196	176	197	229	198	72	199	121
200	151	201	252	202	30	203	130	204	33	205	140	206	27	207	95
208	119	209	84	210	178	211	29	212	37	213	79	214	0	215	70
216	237	217	88	218	82	219	235	220	126	221	218	222	201	223	253
224	48	225	149	226	101	227	60	228	182	229	228	230	187	231	124
232	14	233	80	234	57	235	38	236	50	237	132	238	105	239	147
240	55	241	231	242	36	243	164	244	203	245	83	246	10	247	135
248	217	249	76	250	131	251	143	252	206	253	59	254	74	255	183

(표 2-2) S_2Box

6. 키생성 알고리즘

SEED의 키 생성 알고리듬은 128비트의 암호키를 64비트씩 좌우로 나누어 이들을 교대로 8비트씩 좌/우로 회전이동 후, 결과의 4워드들에 대한 간단한 산술연산과 G 함수를 적용하여 라운드 키를 생성한다. 키 생성 알고리듬은 기본적으로 하드웨어나 (모든 라운드 키를 저장할 수 없는) 제한된 자원을 갖는 스마트카드와 같은 응용에서의 효율성을 위하여, 암호화나 복호화시 암호키로부터 필요한 라운드 키를 간단히 계산할 수 있도록 설계하였다.

각 라운드에 사용되는 라운드 키는 다음과 같은 방식으로 생성한다.

① 128비트 입력키를 32비트 4개의 조각으로 쪼갠 후 (A, B, C, D),

② $K_{1,0}$ = G(A+C-KC_0) ; $K_{1,1}$ = G(B+KC_0-D)

(단, KC_0 : 라운드상수)로 1라운드 키를 생성하고,

③ B||A=(B||A)>>8

④ $K_{2,0}$ = G(A+C-KC_1) ; $K_{2,1}$ = G(B+KC_1-D)

(단, KC_1 : 라운드상수)로 2라운드 키를 생성하고,

⑤ D|| C=(D||C)<<8

⑥ $K_{3,0}$ = G(A+C-KC_2) ; $K_{3,1}$ = G(B+KC_2-D)

(단, KC_2 : 라운드상수)로 3라운드 키를 생성하고,

⑦ 이후 16라운드 키 생성까지 반복한다.

for(i=1; i≤16; i++) {

$K_{i,0} \leftarrow G(A+C-KC_{i-1})$;

$K_{i,1} \leftarrow G(B-D+KC_{i-1})$;

if(i%2==1) $A||I \leftarrow (A||B)^{>>8}$;

else $C||I \leftarrow (C||D)^{<<8}$;

}

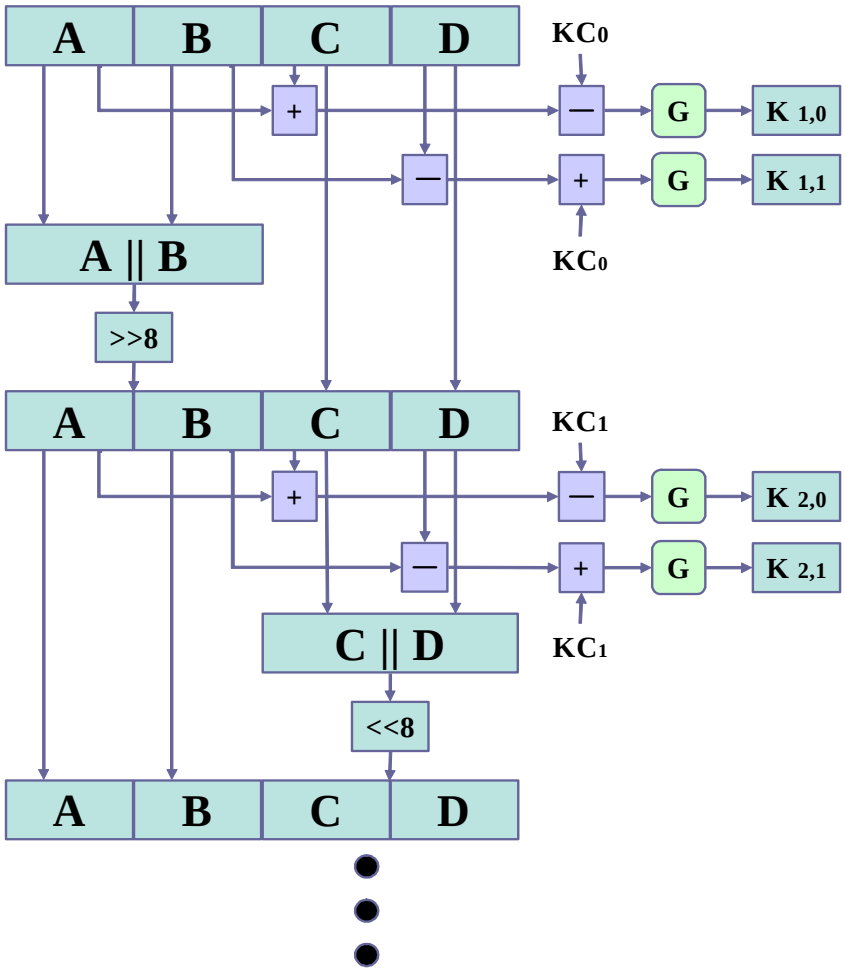
○ 라운드 상수 KC_i는 황금비의 소수 부분으로부터 다음과 같이 생성한다

$$KC_0 = \text{int}(\frac{\sqrt{5}-1}{2} \times 2^{32}) = 0x9e3779b9,$$

$$KC_i = KC_{i-1}{}^{\ll 1} \quad \text{for } 1 \le i \le 15.$$

상 수	
KC0 = 0x9e3779b9	KC8 = 0x3779b99e
KC1 = 0x3c6ef373	KC9 = 0x6ef3733c
KC2 = 0x78dde6e6	KC10 = 0xdde6e678
KC3 = 0xf1bbcdcc	KC11 = 0xbbcdccf1
KC4 = 0xe3779b99	KC12 = 0x779b99e3
KC5 = 0xc6ef3733	KC13 = 0xef3733c6
KC6 = 0x8dde6e67	KC14 = 0xde6e678d
KC7 = 0x1bbcdccf	KC15 = 0xbcdccf1b

(표 2-3) 키생성 알고리즘 사용 상수



(그림 2-4) 키생성 알고리즘 구조도

제 4 절 SEED 수정안

1. SEED 초안의 수정 배경

SEED의 자체평가에서는 암호알고리즘의 안전성과 효율성 측면을 고려함에 있어 민간에서 사용된다는 점이 중시되어 효율성이 우선적으로 고려되었다. 따라서 F함수의 G함수에서 4비트 회전이동만을 시켰으나, 추후 안전성을 충분히 고려하여 효율성의 저하없이 안전성을 강화하는 방안을 개발하여 SEED 초안을 수정하게 되었다.

SEED 초안의 F 함수는 G 함수가 충분한 diffusion을 준다면, 안전성 측면에서 상당히 좋은 구조라 할 수 있다. G 함수에서 S-box 통과 후 단지 4비트 회전 이동만 시킴으로써 자체만으로는 다른 바이트들 간에 diffusion을 주지 못하고, G 함수의 결과들이 3라운드의 변형 Feistel network을 통과하면서 서로 섞이게 되지만, S-box의 확률적 특성을 이용하면 각 워드의 상위 바이트의 difference가 다른 바이트에 영향을 주지 않는 truncated differential들이 높은 확률로 발생할 수 있다. 즉, F 함수의 튼튼한 구조를 지원하기 위해 SEED 초안의 G 함수 회전이동부분에 대한 약간의 수정을 가하였다.

F 함수 구조 변경을 통해 안전도를 높이는 방법은 G 함수에서 S-box 출력들을 적절한 permutation을 통해 충분히 섞어 주는 것이다. 즉, 4비트 회전이동 대신 다른 좋은 성질을 갖는 permutation을 사용하는 것이다. 여기에 사용할 수 있는 permutation으로 가장 간단하면서도 좋은 성질을 갖는 것이 exclusive-or이다. 즉, $X = x_3x_2x_1x_0$에 대해 S-box를 통과한 $Y = y_3y_2y_1y_0$를 $y_0=S_1(x_0)$, $y_1=S_2(x_1)$, $y_2=S_1(x_2)$, $y_3=S_2(x_3)$와 같이 얻고, 최종 출력 $Z=z_3z_2z_1z_0$를 $z_i=y_i \oplus z$, $z=y_0 \oplus y_1 \oplus y_2 \oplus y_3$와 같이 계산하는 것이다. 이 permutation은 diffusion order가 4로 임의의 n(n<4)개의 입력 바이트에서의 변화는 최소한 (4-n) 개의 출력 바이트

에 변화를 주게 된다. 그러나 이 permutation의 단점은 difference들이 바이트 단위로 몰려다닌다는 것이다. 예를 들어 첫 바이트의 입력 difference가 임의의 값 δ이면, 출력 difference는 두 번째에서 네 번째의 세 바이트에 걸쳐 동일한 값 δ를 주게 된다.

비록 위의 성질들이 permutation의 결과가 32비트 덧셈과 결합되면서 발생되는 carry로 인해 어느 정도 파괴되기는 하지만, 가능한 최상의 permutation을 사용하여 F 함수를 보강함으로써 가능하면 라운드 수를 줄여 효율성도 개선시키는 방향으로 변경을 시도하기로 한다. 이런 방향에서 위의 permutation을 좀 더 보강시키는 방법으로 exclusive-or 후의 결과에 다시 다음과 같은 bit-permutation을 추가하여 최종 출력 Z를 구하도록 한다 (exclusive-or 후의 결과를 $W=w_3w_2w_1w_0$로 둔다).

$$
\begin{aligned}
z_0 &= (w_0 \& m_0) \oplus (w_1 \& m_1) \oplus (w_2 \& m_2) \oplus (w_3 \& m_3), \\
z_1 &= (w_0 \& m_1) \oplus (w_1 \& m_2) \oplus (w_2 \& m_3) \oplus (w_3 \& m_0), \\
z_2 &= (w_0 \& m_2) \oplus (w_1 \& m_3) \oplus (w_2 \& m_0) \oplus (w_3 \& m_1), \\
z_3 &= (w_0 \& m_3) \oplus (w_1 \& m_0) \oplus (w_2 \& m_1) \oplus (w_3 \& m_2), \\
&\text{(여기서 \&는 bit-wise AND).}
\end{aligned} \tag{1}
$$

여기서 masking byte m_i들은 다음과 같이 정의된다 :

$$m_0 = 0x03,\ m_1 = 0x0c,\ m_2 = 0x30,\ m_3 = 0xc0$$

이 bit-permutation은 각 입력 바이트의 두 비트씩을 따서 출력 바이트 값을 내는 연산이다. 예를 들면 z_0는 w_0의 첫 두 비트, w_1의 다음 두 비트, w_2의 다음 두 비트, 그리고 w_3의 마지막 두 비트씩을 연접하여 형성된다. 따라서 입력 difference의 nonzero 비트들의 위치에 따라 그 nonzero 비트들이 4개의 출력 바이트 중에 하나 이상으로 가게 된다. 즉, 한 바이트의 nonzero 비트들을 4개의 출력 바이트에 두

비트 단위로 고루 분산시키게 된다. 이 성질로 인해 위의 bit-permutation이 exclusive-or와 결합되면 input difference의 n(<4)개의 바이트가 다음의 30개의 값 중의 하나로 모두 동일한 바이트 값을 가질 때만 diffusion order가 4가 되고, 그 나머지의 경우는 최소한 diffusion order 5가 된다 (from exhaustive computer search):

Minimal diffusion set = {0x01, 0x02, 0x03, 0x04, 0x08, 0x0c, 0x10, 0x20, 0x30, 0x40, 0x80, 0xc0, 0x11, 0x12, 0x13, 0x21, 0x22, 0x23, 0x31, 0x32, 0x33, 0x44, 0x48, 0x4c, 0x84, 0x88, 0x8c, 0xc4, 0xc8, 0xcc}.

예를 들면 S-box의 출력차가 (0x01, 0x01, 0x01, 0x00)이면 G 함수의 출력차는 (0x00, 0x00, 0x00, 0x01)이 되며, (0x01, 0x01, 0x00, 0x00)은 (0x00, 0x01, 0x01, 0x00)으로 간다. 특히 마지막 18개의 값(0x11에서 0xcc까지)은 2개의 입력 nonzero byte들이 떨어져 나타나는 경우에만 diffusion order 4를 주고, 그 외에는 diffusion order 5 이상이 된다. 예를 들어 (0x11, 0x00, 0x11, 0x00)은 (0x11, 0x00, 0x11, 0x00)으로 가지만, (0x00, 0x11, 0x11, 0x00)은 (0x10, 0x10, 0x10, 0x10)으로, (0x11, 0x00, 0x00, 0x00)은 (0x01, 0x11, 0x10, 0x11)로 가서 4개의 출력 바이트 모두가 nonzero가 된다 (각각 diffusion order 6, 5).

참고로 masking byte m_i들은 각각이 Hamming weight 2이고 4개가 partition만 되면 (즉, nonzero bit들의 위치가 겹치지만 않으면) bit-permutation의 성질에는 크게 영향을 미치지 않는다. 예를 들면 m_0=0x81, m_1=0x24, m_2=0x18, m_3=0x42를 사용해도 무방하다.

이 bit-permutation은 exclusive-or와 결합시켜 식 (1)과 같은 형태로 간단히 표현할 수 있다. 즉, S-box의 출력 Y에 대해 exclusive-or 연산과 bit-permutation이 결합되어 출력 Z를 줄 때, Z는 위의

bit-permutation식에서 masking byte m_i들 대신에 이들의 2의 보수를 사용하면 된다. 즉, 전체 G 함수는 다음과 같이 기술된다:

$$y_0 = S_1(x_0),\ y_1 = S_2(x_1),\ y_2 = S_1(x_2),\ y_3 = S_2(x_3),$$

$$\begin{aligned} z_0 &= (y_0 \& m_0) \oplus (y_1 \& m_1) \oplus (y_2 \& m_2) \oplus (y_3 \& m_3) \\ z_1 &= (y_0 \& m_1) \oplus (y_1 \& m_2) \oplus (y_2 \& m_3) \oplus (y_3 \& m_0) \\ z_2 &= (y_0 \& m_2) \oplus (y_1 \& m_3) \oplus (y_2 \& m_0) \oplus (y_3 \& m_1) \\ z_3 &= (y_0 \& m_3) \oplus (y_1 \& m_0) \oplus (y_2 \& m_1) \oplus (y_3 \& m_2) \end{aligned} \qquad (2)$$

$$(m_0 = 0\text{xfc},\ m_1 = 0\text{xf3},\ m_2 = 0\text{xcf},\ m_3 = 0\text{x3f})$$

위의 G 함수는 기존의 G 함수와 마찬가지로 4개의 확장된 4바이트 SS-box들(4Kbytes)의 exclusive-or로 구현할 수 있다. 이를 위해 다음과 같은 4개의 SS-box들을 저장해야 한다.

$$\begin{aligned} SS_0 &= S_1(x_0)\&m_3 \parallel S_1(x_0)\&m_2 \parallel S_1(x_0)\&m_1 \parallel S_1(x_0)\&m_0, \\ SS_1 &= S_2(x_1)\&m_0 \parallel S_2(x_1)\&m_3 \parallel S_2(x_1)\&m_2 \parallel S_2(x_1)\&m_1, \\ SS_2 &= S_1(x_2)\&m_1 \parallel S_1(x_2)\&m_0 \parallel S_1(x_2)\&m_3 \parallel S_1(x_2)\&m_2, \\ SS_3 &= S_2(x_3)\&m_2 \parallel S_2(x_3)\&m_1 \parallel S_2(x_3)\&m_0 \parallel S_2(x_3)\&m_3, \end{aligned} \qquad (3)$$

(여기서, ||는concatention).

이 확장 SS-box들을 이용하면 G 함수는 다음과 같이 구현될 수 있다:

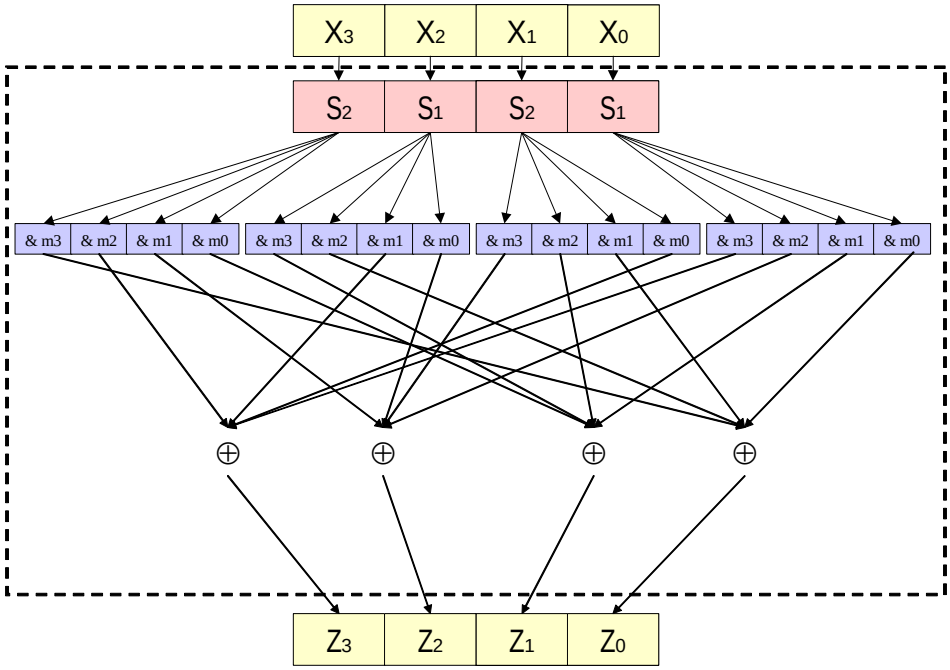
$$Z = SS_0(x_0) \oplus SS_1(x_1) \oplus SS_2(x_2) \oplus SS_3(x_3) \qquad (4)$$

따라서, 전혀 효율성의 저하 없이 안전도를 훨씬 강화할 수 있다.

2. SEED 수정안

※ 참고 : SEED 초안에서 G 함수만 변경되었음

위에서 살펴보았듯이 수정된 SEED는 초안의 G 함수만을 변경하여 안전도를 더욱 강화시켰으며, SEED 초안과 비교하여 효율성에 저하가 없도록 변경되었다. 수정 G 함수를 살펴보면 다음과 같다.



(그림 2-5) 수정된 G 함수

(단, $m_0 = 0xfc$, $m_1 = 0xf3$, $m_2 = 0xcf$, $m_3 = 0x3f$)

제 3 장 안전성 분석

제 1 절 차분해독법

1. SEED 초안의 F 함수 DC 특성 분석

여기에서는 32비트 워드의 상위 두 비트만이 nonzero인 differential을 계산하는 과정만을 소개한다(유사한 방법으로 상위 4비트, 8비트 등이 nonzero인 differential에 대한 확률들을 계산할 수 있다). 우선 상위 두 비트가 nonzero인 differential 확률을 계산하기 위해 G 함수 및 32비트 덧셈에 대한 확률을 먼저 계산한다. 이들 확률들이 (표 3-1)와 (표 3-2)에 나타나 있다. 덧셈의 경우 상위 2비트가 nonzero인 xor difference에 대해 주어진 xor difference pair를 주는 모든 가능한 경우를 고려하여 덧셈 후에 나타날 수 있는 xor difference 및 그 확률을 계산한 것이다.

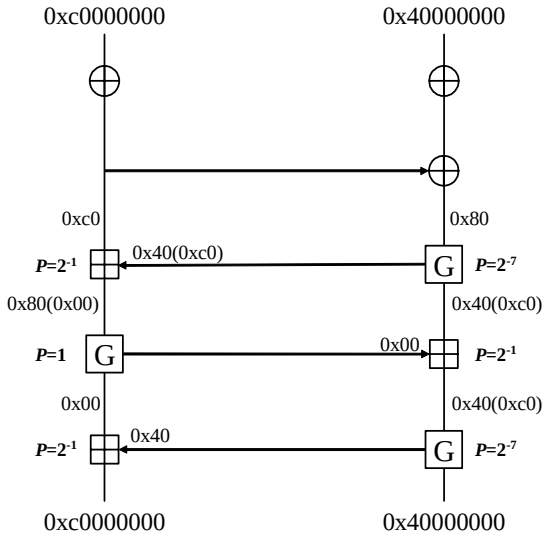
입력차이값	출력차이값과 그 확률
0x400000000	0x400000000 : 확률 1/2^7
0x800000000	0x400000000 : 확률 1/2^7 0x800000000 : 확률 1/2^7 0xc00000000 : 확률 1/2^7
0xc00000000	0x400000000 : 확률 1/2^7 0xc00000000 : 확률 1/2^7

(표 3 1) 상위 2비트만을 고려한 G 함수에 대한 characteristic 확률

두 입력차이값	출력차이값과 그 확률
(0x00000000, 0x80000000)	0x80000000 : 확률 1
(0x80000000, 0x80000000)	0x00000000 : 확률 1
(0x00000000, 0x40000000) (0x00000000, 0xc0000000) (0x40000000, 0x80000000) (0x80000000, 0xc0000000)	0x40000000 : 확률 1/2 0xc0000000 : 확률 1/2
(0x40000000, 0x40000000) (0x40000000, 0xc0000000) (0xc0000000, 0xc0000000)	0x00000000 : 확률 1/2 0x80000000 : 확률 1/2

(표 3-2) 상위 2비트만을 고려한 32비트 덧셈에 대한 characteristic 확률

위의 두 표를 이용하면 F 함수의 characteristic에 대한 확률을 계산할 수 있다. 예를 들어 F 함수에 대한 최상의 characteristic 중 하나인 (0xc0000000, 0x40000000) → (0xc0000000, 0x40000000)의 확률을 계산해 보자(아래에서는 각 difference pair의 최상위 바이트만을 표시한다. E.g., 0x40000000 → 0x40). (그림 3-1)을 참조하여 설명한다.



(그림 3-1) 확률 2^{-15}의 F 함수 differential 예

우선 함수 G는 0x80을 2^{-7}의 확률로 0x40 혹은 0xc0으로 보낸다 (0x80이 될 확률도 2^{-7}이지만, 이 경우는 원하는 characteristic (0x40, 0xc0)을 주지 못한다). 그리고 0x40이나 0xc0이 32비트 덧셈을 통과하면 1/2의 확률로 0x00 혹은 0x80이 나온다. 여기서 0x00이 나오는 경우만 보면(0x80의 경우는 전체적으로 2^{-24}의 확률을 주는 characteristic을 낳는다), 이 0x00이 확률 1로 G 함수를 통과하여 두 번째 덧셈에서 0x40이나 0xc0과 더해지면 각각이 확률 1/2로 0x40(혹은 0xc0)이 되고, 이들 0x40(0xc0)은 다시 마지막 G 함수에 의해 각각이 2^{-7}의 확률로 0x40으로 보내진다. 마지막으로 이 0x40이 오른쪽의 0x00과 더해지면 확률 1/2로 0xc0이 된다. 따라서 (0xc0, 0x40) → (0xc0, 0x40)의 characteristic은 총 4개가 생기고 각각이 확률 2^{-17}으로 일어나므로 전체적인 확률은 평균 2^{-15}이 된다. 이 4개의 characteristic이 생기는 경로를 보면 아래와 같다.

- (c0, 40) → G1(80→40; 2^{-7}) → A1((40, c0)→00; 2^{-1}) → G2(00→00;1)→ A2((40, 00)→40; 2^{-1}) → G3(40→40; 2^{-7}) → A3((40, 00)→c0; 2^{-1})

- (c0, 40) → G1(80→40; 2^{-7}) → A1((40, c0)→00; 2^{-1}) → G2(00→00;1) → A2((40, 00)→c0; 2^{-1}) → G3(c0→40; 2^{-7}) → A3((40, 00)→c0; 2^{-1})

- (c0, 40) → G1(80→c0; 2^{-7}) → A1((c0, c0)→00; 2^{-1}) → G2(00→00;1) → A2((c0, 00)→40; 2^{-1}) → G3(40→40; 2^{-7}) → A3((40, 00)→c0; 2^{-1})

- (c0, 40) → G1(80→c0; 2^{-7}) → A1((c0, c0)→00; 2^{-1}) → G2(00→00;1) → A2((40, 00)→c0; 2^{-1}) → G3(c0→40; 2^{-7}) → A3((40, 00)→c0; 2^{-1})

위와 같은 방법으로 모든 경우를 확인하면 (표 3-3)과 같이 함수 F가 특정한 입력 차이값에 대해 특정한 출력 차이값을 가질 확률을 계산할 수 있다 (counted by a computer program).

출력 / 입력	00-40	00-80	00-c0	40-00	40-40	40-80	40-c0	80-00	80-40	80-80	80-c0	c0-00	c0-40	c0-80	c0-c0
00-40	100	0	100	2	102	2	102	0	2	0	2	2	81	2	81
00-80	300	0	300	6	200	6	200	0	6	400	6	6	200	6	200
00-c0	200	0	200	4	204	4	204	0	4	0	4	4	102	4	102
40-00	180	0	180	3	0	3	0	0	3	0	3	3	0	3	0
40-40	0	0	0	300	0	300	0	0	0	0	0	180	0	180	0
40-80	300	0	300	6	0	6	0	0	6	0	6	6	0	6	0
40-c0	200	0	200	a	204	a	204	0	4	0	4	7	102	7	102
80-00	300	400	300	e	0	e	0	0	6	0	6	a	0	a	0
80-40	200	0	200	4	204	4	204	0	4	0	4	4	102	4	102
80-80	0	0	0	400	200	400	200	400	0	0	0	200	200	200	200
80-c0	100	0	100	2	102	2	102	0	2	0	2	2	81	2	81
c0-00	300	0	300	6	0	6	0	0	6	0	6	6	0	6	0
c0-40	200	0	200	a	204	a	204	0	4	0	4	7	102	7	102
c0-80	180	0	180	3	0	3	0	0	3	0	3	3	0	3	0
c0-c0	0	0	0	300	0	300	0	0	0	0	0	180	0	180	0

(표 3-3) 상위 두 비트씩이 nonzero인 F 함수 characteristic 확률

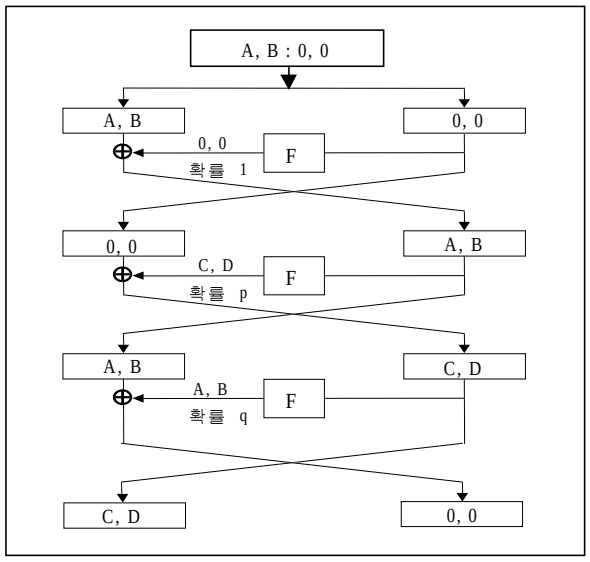
(표 3-3)의 수는 16진수이고, 단위는 1/2^{24}이며 확률을 나타낸다. 예를 들면 c0-40/c0-40의 값 204는 함수 F가 (0xc0000000, 0x40000000)을 (0xc0000000, 0x40000000)으로 보낼 확률이 204/2^{24}임을 의미하며, 이 확률은 2^{-15}+2^{-22}이다. (0xc0000000, 0x00000000) → (0xc0000000, 0x00000000)도 거의 같은 확률로 일어난다.

2. SEED 초안의 n-Round Differential 확률 계산

위에서 구한 F 함수의 DC 특성을 이용하여 3라운드의 truncated differential을 구해 보자. 여기서 관심이 있는 것은 다음과 같은 형태의 3라운드 iterative truncated differential이다:

$$(A, B\ 0, 0) \rightarrow (C, D, 0, 0)$$

여기서 A, B, C, D는 모두 상위 두 비트만이 nonzero인 32비트 워드이다. 즉, 각각의 가능한 A, B에 대해 3라운드 후 상위 두 비트만 nonzero인 모든 가능한 C, D로 가는 characteristic들에 대한 확률들을 구하고자 한다. 이러한 3라운드 characteristic들은 주어진 A, B 및 임의의 C, D에 대해 (A, B) → (A, B) 혹은 (A, B) → (C, D) 및 (C, D) → (A, B)를 만족하는 F 함수 differential이 존재하는 경우이다 (그림 3-2 참조).



(그림 3-2) 3라운드 truncated differential

이러한 성질을 만족하는 F 함수 characteristic 중 큰 확률을 갖는 4개의 difference 값은 다음과 같다:

I_0 = (00, 40, 00, 00),

I_1 = (00, c0, 00, 00),

I_2 = (40, c0, 00, 00),

I_3 = (c0, 40, 00, 00).

(표 3-3)에 주어진 이들의 F 함수 확률을 이용하여 $I_1 \rightarrow I_2$의 3라운드 characteristic 확률 P_{12}을 구해 보면

$$P_{12} = \Pr\{(00, c0) \rightarrow_F (40, c0)\} \times \Pr\{(40, c0) \rightarrow_F (00, c0)\}$$
$$= 2^{-15} \times 2^{-15} = 2^{-30}.$$

마찬가지 방법으로 모든 P_{ij}를 계산하여 (표 3-4)에 정리하였다.

입력 \ 출력	I_0	I_1	I_2	I_3	dif. prob.
I_0	P_{00}=2^{-32}	P_{01}=2^{-31}	P_{02}=2^{-31}	P_{03}=2^{-32}	2^{-29.4}
I_1	P_{10}=2^{-31}	P_{11}=2^{-30}	P_{12}=2^{-30}	P_{13}=2^{-31}	2^{-28.4}
I_2	P_{20}=2^{-31}	P_{21}=2^{-30}	P_{22}=2^{-30}	P_{23}=2^{-31}	2^{-28.4}
I_3	P_{30}=2^{-32}	P_{31}=2^{-31}	P_{32}=2^{-31}	P_{33}=2^{-32}	2^{-29.4}

(표 3-4) 3라운드의 characteristic 및 truncated differential 확률

따라서 원하는 3라운드 truncated differential의 확률 중 가장 좋은 것은 I_2에서 출발하는 것으로 (I_1도 거의 같으나, 실제로 소수점 이하에서 I_2가 조금 좋음) $P_{3R} = \sum_{i=0}^{3} P_{3i} = 2^{-28.4}$ 임을 알 수 있다.

3의 배수 라운드에서의 확률은 출발점의 input difference를 선택하여 원하는 라운드 output difference의 오른쪽 두 워드가 모두 0이고

왼쪽 워드의 각 상위 두 비트만이 nonzero인 모든 가능한 characteristic 확률의 합을 구하면 된다. (표 3-5)를 이용하여 이런 방법으로 I_2에서 출발하는 6, 9, 12, 15라운드의 truncated differential을 다음과 같이 구할 수 있다:

$$
\begin{aligned}
P_{6R} &= \sum_{i=0}^{3} P_{2i}\Big(\sum_{j=0}^{3} P_{ij}\Big) = 2^{-57.1},\\
P_{9R} &= \sum_{i=0}^{3} P_{2i}\Big(\sum_{j=0}^{3} P_{ij}\Big(\sum_{k=0}^{3} P_{jk}\Big)\Big) = 2^{-85.7},\\
P_{12R} &= \sum_{i=0}^{3} P_{2i}\Big(\sum_{j=0}^{3} P_{ij}\Big(\sum_{k=0}^{3} P_{jk}\Big(\sum_{l=0}^{3} P_{kl}\Big)\Big)\Big) = 2^{-114.4},\\
P_{15R} &= \sum_{i=0}^{3} P_{2i}\Big(\sum_{j=0}^{3} P_{ij}\Big(\sum_{k=0}^{3} P_{jk}\Big(\sum_{l=0}^{3} P_{kl}\Big(\sum_{m=0}^{3} P_{lm}\Big)\Big)\Big)\Big) = 2^{-143.1}
\end{aligned}
$$

이렇게 이론적으로 계산한 확률들의 신뢰성을 확인하기 위해 실제로 충분히 많은 랜덤 데이터에 대해 F 함수의 입/출력 특성을 시뮬레이션해 보았다. 즉, 주어진 input difference를 갖는 2^{28}개 정도의 랜덤하게 생성된 64비트 데이터 쌍들로 F 함수를 돌린 후 각각의 output difference를 주는 데이터의 수를 카운트하여 결과를 (표 3-5)에 정리하였다. 대부분 중요한 확률들이 2^{-14} - 2^{-16} 정도이므로 2^{28}개 정도의 데이터면 충분히 신뢰성 있는 실측 확률을 구할 수 있을 것이다.

입력 \ 출력	00-40	00-80	00-c0	40-00	40-40	40-80	40-c0	80-00	80-40	80-80	80-c0	c0-00	c0-40	c0-80	c0-c0
00-40	0	0	0	0	77a	334	17cc	41	153	0	138	299	fd9	267	10c
00-80	3f03	0	415c	1dc	3cfc	94	0	0	bc	401f	1a8	152	4044	0	f1
00-c0	1815	0	189c	1b1	1a83	3f	28bc	84a	4c0	0	c7	0	17b	c6	26b1
40-00	df3	0	1212	3f1	71	1c5	1b2	136	bc	0	ca	0	0	11c	0
40-40	0	0	0	200c	0	5cd5	0	0	0	0	0	0	0	2814	0
40-80	40b0	0	4021	2bf	0	0	0	c5	68	0	0	0	c0	198	82
40-c0	2b0f	0	235d	17f	2933	100	1141	197	50c	0	1b7	427	1ca5	c3	d6
80-00	1c47	3f98	24b4	0	0	107	0	293	18c	0	ff	0	1c2	1a9	0
80-40	29bb	0	24aa	64	16cd	3e4	2881	c5	203	0	4c	200	273	102	1a57
80-80	0	0	0	4033	2003	4002	2005	3f6d	0	0	0	0	40f1	3f63	0
80-c0	1cd5	0	2290	0	9d8	35	1709	0	205	0	2d7	224	1009	2cd	0
c0-00	1c3c	0	2101	c2	0	0	0	717	599	0	26b	0	b2	1a7	0
c0-40	1200	0	1c4a	235	27b2	548	1c2c	10c	218	0	44	4ac	261c	1ac	137
c0-80	26ab	0	19c0	436	af	0	0	0	3ba	0	14b	0	170	171	0
c0-c0	0	0	0	3faf	0	230	0	0	0	0	0	0	0	381b	0

(표 3-5) Simulation에 의한 F 함수의 DC 특성
(확률 = 0xn × 2^{-28})

위의 (표 3-5)를 이론적으로 계산한 (표 3-3)과 비교해 보면, 정도의 차이는 있으나, 대부분 이론적인 계산치와 맞아떨어지지만 중요하게 차이가 나는 부분들도 꽤 있다. 예를 들어 이론적으로는 (00, 40) → (00, 40)의 확률이 2^{-16} 정도의 확률로 일어나지만, 위의 (표 3-5)에서 보듯이 실제로는 이러한 characteristic은 일어나지 않는 것으로 나온다. 비록 약간의 오차는 있겠지만 simulation 결과가 실제 상황이므로 이론적으로 계산한 확률들에 약간의 오류가 있음을 알 수 있다. 그 원인을 분석해 본다면 (표 3-1)과 (표 3-2)의 S-box 및 덧셈에 대한 확률은 모든 값들이 균일한 확률 분포를 가지고 일어난다는 가정하

에서 계산한 것이나, 실제로는 F 함수 내의 각 단계를 통과한 값들이 균일한 확률로 일어나지 않기 때문일 것이다. 즉, S-box나 덧셈의 입력으로 들어가는 값들이 전 단계에서 계산된 값들이므로 어떤 제한된 집합 내에서만 일어날 것이다.

비록 개별적인 characteristic의 확률에서는 약간의 차이가 나지만 아래에서 보듯이 이 실측치를 이용하여 계산한 전체적인 differential 확률은 거의 이론치와 일치하므로, 위의 이론적인 분석이 아주 틀리지는 않음을 알 수 있다. 아래 (표 3-6)에서 실측 확률은 위의 (표 3-5)를 이용하여 전과 같은 방법으로 계산한 특정 입력차에서 출력의 오른쪽 각 워드 상위 두 비트가 nonzero인 truncated differential에 대한 확률이다. (표 3-6)은 이론치와 실측치가 거의 차이가 없음을 보여준다. 실제로 출력의 양쪽 모두의 각 워드들에서 상위 두 비트가 nonzero인 truncated differential에 대한 확률은 아래 표의 값들보다 훨씬 크다. 예를 들어 이렇게 계산된 15라운드의 확률은 2^{-137.6} (실측: 2^{-137.9})이다. 그러나 이 확률로는 16라운드의 SEED에 대한 공격이 불가능하고, 또한 이런 truncated differential로는 1R attack을 적용해야 하므로 비실용적인 counter-keeping이 필요하며, 마지막으로 더 많은 nonzero비트들을 고려하면 이보다 훨씬 좋은 확률을 구할 수 있으므로 이런 종류의 공격은 더 이상 고려하지 않는다.

라운드 수	differential prob.(이론)	differential prob.(실측)
3	2^{-28.4}	2^{-28.2}
6	2^{-57.1}	2^{-57.1}
9	2^{-85.7}	2^{-85.6}
12	2^{-114.4}	2^{-114.3}
15	2^{-143.1}	2^{-142.9}

(표 3-6) 이론 및 simulation에 의한 differential 확률 비교

이상에서는 각 워드의 상위 두 비트만이 nonzero인 truncated differential만을 생각하였으나, 실제로 4비트나 8비트가 nonzero인 characteristic들이 훨씬 많으므로 differential 확률 역시 훨씬 높아질 것임은 자명하다. 아래 (표 3-7)에 상위 4비트가 nonzero인 truncated differential에 대한 확률을 정리하였다. 이런 확률을 구하기 위해서는 오른쪽 두 워드의 difference가 모두 0이고 왼쪽 두 워드의 상위 4비트만이 nonzero인 모든 가능한 각각의 input difference에 대해 3라운드 후에 같은 형태의 output difference를 주는 모든 확률들을 계산해야 한다. 물론 우선은 상위 4비트를 고려한 S-box 및 덧셈에 대한 확률들을 구해야 하고, 이들을 이용해 앞의 두 비트에 대한 경우와 마찬가지 방법으로 F 함수에 대한 모든 가능한 characteristic 확률들을 구하고, 다시 이를 이용해 3라운드의 truncated differential을 구하는 복잡한 계산을 해야하므로 computer program을 사용해야만 가능하다.

라운드 수	differential prob.(이론)	differential prob.(실측)
3	2^{-23.5}	2^{-23.4}
6	2^{-52.0}	2^{-49.8}
9	2^{-76.6}	2^{-75.1}
12	2^{-105.1}	2^{-101.5}
15	2^{-129.8}	2^{-127.0}

(표 3-7) 상위 4비트를 고려한 이론 및 simulation에 의한 differential 확률 비교

(표 3-5)의 상위 두 비트만을 고려한 확률과 비교해 보면 약 2^{13}배 이상이 좋아졌음을 알 수 있다. 위의 확률은 3라운드의 iterative truncated differential만을 고려하여 얻은 것이므로, 실제로는 15라운드의 확률은 2^{-128}보다 훨씬 클 것으로 예상된다. 참고로 위의 15

라운드 확률은 input difference가 (0x80000000, 0x80000000; 0x00000000, 0x00000000)이고, 15라운드 output difference가 (0xp0000000, 0xq0000000; 0x00000000, 0x00000000) 형태인 truncated differential에 대한 확률이다 (p, q는 임의의 16진수 수).

3. SEED 초안의 Differential Cryptanalysis

위에서 살펴보았듯이 상위 4비트만 nonzero인 3라운드의 iterative truncated differential을 이용한 15라운드의 differential 확률은 실제로 2^{-128}보다 크다. 상위 8비트 정도가 nonzero인 truncated differential은 대략 2^{-110} - 2^{-120} 사이가 될 것으로 추정되고, 모든 가능성을 고려한 최상의 공격이면, 2^{-110} 이상이 될 가능성도 충분한 것 같다. 편의상 여기서는 15라운드 differential 확률을 2^{-127}(실측치)으로 가정하고 16라운드의 SEED를 공격하는 방법을 기술한다.

우선 앞에서 언급했듯이 15라운드 output difference의 오른쪽 두 워드가 모두 0이므로 15라운드 differential은 곧 16라운드 differential과 같다. 상위 4비트만이 nonzero인 최상의 truncated differential의 경우 16라운드의 최종 output difference는

(0xp0000000, 0q00000000; 0x00000000, 0x00000000)

형태가 될 것이다. 이제 시작점의 input difference가 (0x80000000, 0x80000000; 0x00000000, 0x00000000)인 2^{128}개의 128비트 입력쌍을 암호화하여 출력쌍의 차이가 예상치와 다른 모든 ciphertext pair는 버린다. 이 과정에서 살아 남은 ciphertext pair는 약 2^{8}개 정도가 될 것이고, 이 중 right pair는 확률적으로 2개 정도가 있을 것이

다. 이 정도의 right pair candidate이면 counting이 필요없이 전수 조사에 의해 마지막 라운드 키를 찾아낼 수 있다. 즉, 모든 가능한 마지막 라운드 키 값으로 2^{8}개의 candidate들을 한 라운드 복호화하여 F 함수 출력 difference가 모두 0이 되는 라운드 키를 찾는 것이다. Difference 0인 랜덤 쌍이 difference 0인 출력을 낳을 확률은 2^{-64} 정도이므로 2^{8}개의 candidate들 중 하나라도 제대로 된 출력쌍을 주는 키가 거의 확실하게 올바른 라운드 키가 될 것이다. 이 과정에서 필요한 계산량은 2^{72} 정도의 1-round 복호화로 이전 단계의 계산량에 비하면 무시할 수 있을 정도이다.

위와 같은 공격으로 15라운드 differential 확률이 2^{-n}인 경우 약 2^{n} 정도의 암호화면 마지막 라운드 키를 찾을 수 있다. 이 과정에서 필요한 메모리 요구량은 거의 없으므로 순수히 2^{n}번의 암호화에 필요한 계산만 하면 된다. 예를 들어 상위 8비트만이 nonzero인 truncated differential의 15라운드 확률이 2^{-110}이라면, filtering 과정을 통과하는 pair가 거의 확실한 right pair가 될 것이다 (filtering 과정에서의 오류 확률이 2^{-112} 정도이므로).

제 2 절 선형해독법

1. 개요

SEED 초안의 선형근사식을 얻기 위해 먼저 3라운드 반복 선형근사식을 찾는다. 우리가 찾은 3라운드 반복 선형근사식의 확률은 1/2+189/2^26이며, 이것을 5번 반복 사용하면 15라운드 선형근사식을 얻을 수 있다. SEED 초안은 16라운드이므로 15라운드 선형근사식을 이용하여 키를 찾을 수 있다. 15라운드 선형근사식의 확률이 1/2+1/2^88.2이므로 SEED 초안의 LC 복잡도는 2^176.4이다. 따라서 SEED 초안은 LC 공격에 안전한 것으로 예측된다.

2. 덧셈의 선형근사

SEED 초안의 F함수 내에는 32비트끼리의 덧셈 연산이 3개 있다. x, y 가 32비트일 때, $x \boxplus y = x + y \pmod{2^{32}}$ 로 정의된다. 좀더 일반적으로 x, y 가 n 비트일 때, $x \boxplus y = x + y \pmod{2^n}$ 의 성질을 살펴보자. $x=(x_1, \cdots, x_n), y=(y_1, \cdots, y_n)$, 그리고 $c=(c_1, \cdots, c_n)$ 를 캐리라고 하자. 그러면 $x \boxplus y = (x_1 \oplus y_1 \oplus c_1, \cdots, x_n \oplus y_n \oplus c_n)$ 이다. 분명히 $c_1=0$이며, c_i에 대한 좀 더 많은 성질을 살펴보자. x, y 가 독립이고 일양분포를 가지면 다음 성질이 성립한다.

성질 (1) $P(c_{i+1}=0 | c_i=0) = P(c_{i+1}=1 | c_i=1) = 3/4$

(2) $P(c_{i+1}=1 | c_i=0) = P(c_{i+1}=0 | c_i=1) = 1/4$

(3) $P(c_1=0)=1, \quad P(c_1=1)=0$

(4) $P(c_2=0)=3/4, \quad P(c_2=1)=1/4$

(5) $P(c_3=0)=10/16,\ P(c_3=1)=6/16$

(6) $P(c_i \oplus c_{i+1}=0)=3/4$

이미 언급하였듯이 성질 (3)은 자명하며, 성질 (1)도 비교적 쉽게 증명할 수 있다. 성질 (2)는 성질 (1)로부터 나온다. 성질 (4)는 성질 (1), (2), (3)을 이용하여 계산할 수 있다.

즉,

$$\begin{aligned} P(c_2=0) &= P(c_2=0|c_1=0)P(c_1=0)+P(c_2=0|c_1=1)\,P(c_1=1) \\ &= P(c_2=0|c_1=0) \\ &= \frac{3}{4} \end{aligned}$$

이다.

같은 방법으로 성질 (5)도 계산할 수 있다. 성질 (6)도 성질(1)과 (2)를 이용하여 계산할 수 있다.

$$\begin{aligned} P(c_i \oplus c_{i+1}=0) &= P(c_i=c_{i+1}=0)+P(c_i=c_{i+1}=1) \\ &= P(c_{i+1}=0|c_i=0)P(c_i=0) \\ &\quad + P(c_{i+1}=1|c_i=1)P(c_i=1) \\ &= \frac{3}{4}P(c_i=0)+\frac{3}{4}P(c_i=1) \\ &= \frac{3}{4} \end{aligned}$$

위의 성질을 이용하여 x, y 의 덧셈을 z , 즉, $z=x \boxplus y$ 라고 할 때 $z=(z_1,\cdots,z_n)$ 의 비트를 x, y 의 비트로 근사시킬 수 있다.

$z_1=x_1 \oplus y_1$ (확률1)

$z_2=x_2 \oplus y_2$ (확률 3/4)

$z_i \oplus z_{i+1}=x_i \oplus y_i \oplus x_{i+1} \oplus y_{i+1}$ (확률 3/4)

3. SEED 초안 F 함수의 선형근사

SEED 초안의 3라운드 반복 선형근사를 설계하기 위해 다음과 같은 F함수의 선형근사를 찾으면 된다.

$$(b,0) \xrightarrow{F} (0,a) \qquad (0,a) \xrightarrow{F} (b,0),$$

여기서 a, b 는 32비트 벡터이며, 0은 32비트 0벡터를 나타낸다. F 함수 내에는 3개의 G 함수와 3개의 덧셈 연산이 있다. 덧셈 연산에서 출력의 최하위 비트는 입력의 최하위 비트들로 선형근사 시키면 좋으나(선형근사 확률 1) 덧셈 연산은 G 함수와 연결되어 있다. G 함수내에는 4개의 S-box(2개의 S1-box와 2개의 S2-box)가 작용하며 S-box들의 출력 32비트가 4비트 왼쪽으로 순환이동하여 G 함수의 출력이 된다. 따라서 G 함수의 출력 최하위 비트는 G 함수의 입력 최하위 비트로 선형근사시킬 수 없다. 다음으로 덧셈 연산에서 출력을 입력으로 선형근사시킬 수 있는 선형근사 확률은 3/4이다. 출력의 둘째 하위비트를 입력의 둘째 하위비트로 선형근사할 때 확률은 3/4이며, 인접하는 2개의 출력비트를 대응되는 2개의 입력비트로 선형근사할 때 확률도 3/4이다. 둘째 하위비트를 선형근사할 경우는 최하위 비트를 선형근사할 때와 같이 F 함수로까지 선형근사시킬 수 없다.

이제 덧셈 연산에서 출력의 인접하는 2개의 비트를 대응되는 2개의 입력비트로 선형근사시키는 경우에 대해 고려해 보자. G함수는 S-box의 출력 32비트를 4비트 왼쪽으로 순환이동시키므로 다음과 같은 S-box의 선형근사를 찾아야 한다.

$$0x30 \xrightarrow{S} 0x03 \ , \ 0x60 \xrightarrow{S} 0x06 \ , \ 0xc0 \xrightarrow{S} 0x0c$$

S1-box와 S2-box에 대해 위의 형태의 선형근사 확률은 다음 (표 3-8)과 같다.

S-Box	S-box입력 --> S-box 출력	선형근사 확률
S1-box	0x30 --> 0x03	1/2+ (-4)/2^8
	0x60 --> 0x06	1/2+ (-2)/2^8
	0xc0 --> 0x0c	1/2+ 2/2^8
S2-box	0x30 --> 0x03	1/2+ 12/2^8
	0x60 --> 0x06	1/2+ (-4)/2^8
	0xc0 --> 0x0c	1/2+ (-12)/2^8

(표 3 8) S Box의 선형근사확률

위의 S-box의 선형근사를 이용하여 다음 (표 3-9)와 같이 G함수의 선형근사 확률을 구할 수 있다.

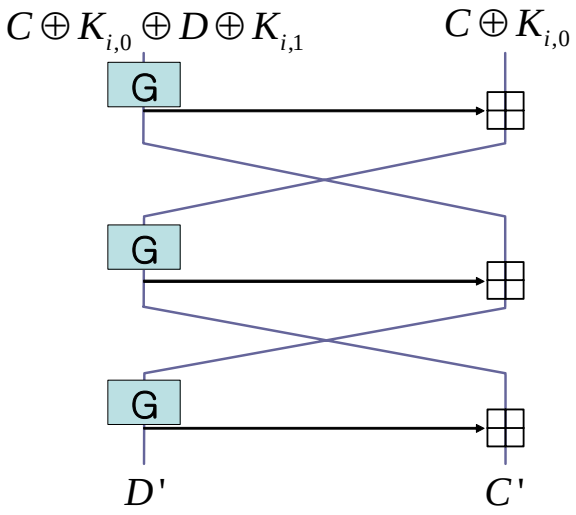
G함수 입력 → G함수 출력	선형근사 확률
0x00000030 → 0x00000030	1/2+ (-4)/2^8
0x00000060 → 0x00000060	1/2+ (-2)/2^8
0x000000c0 → 0x000000c0	1/2+ 2/2^8
0x00003000 → 0x00003000	1/2+ 12/2^8
0x00006000 → 0x00006000	1/2+ (-4)/2^8
0x0000c000 → 0x0000c000	1/2+ (-12)/2^8
0x00300000 → 0x00300000	1/2+ (-4)/2^8
0x00600000 → 0x00600000	1/2+ (-2)/2^8
0x00c00000 → 0x00c00000	1/2+ 2/2^8
0x30000000 → 0x30000000	1/2+ 12/2^8
0x60000000 → 0x60000000	1/2+ (-4)/2^8
0xc0000000 → 0xc0000000	1/2+ (-12)/2^8

(표 3 9) G함수의 선형근사확률

위의 G 함수의 선형근사에서 입력과 출력은 같으며, 표현을 간단히 하기 위해 이런 형태의 입력, 출력을 간단히 a로 쓰기로 하자. 이젠 위의 G의 선형근사를 이용하여 다음과 같은 형태의 F 함수의 선형근사를 구해보자.

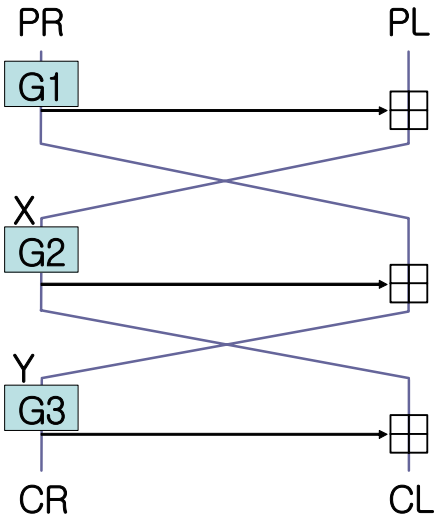
$$(0,a)\xrightarrow{F}(b,(\quad(b,0)\xrightarrow{F}(0,c$$

F 함수는 다음 (그림 3-3)과 같이 표현할 수 있다.



(그림 3-3) F 함수의 다른 표현

$C\oplus K_{i,0}\oplus D\oplus K_{i,1}$을 PR, $C\oplus K_{i,0}$을 PL이라 두고, C′을 CL, D′을 CR이라 두고, G2의 입력을 X, G3의 입력을 Y라고 두자. 그러면 F 함수는 다음 (그림3-4)와 같이 표현된다.



(그림 3-4) F 함수 중간 입출력값 정의

$a \xrightarrow{G} c$의 선형근사 확률을 1/2+p1, $a \xrightarrow{G} b$의 선형근사 확률을 1/2+p2, $b \xrightarrow{G} a$의 선형근사 확률을 1/2+p3이라고 쓰자. 여기서 b는 32비트 벡터이다. 벡터 a는 Hamming weight가 2이며 2개의 1비트가 인접해 있다. 한편 $z = x \boxplus y$ 라고 할 때, $z_i \oplus z_{i+1} = x_i \oplus y_i \oplus x_{i+1} \oplus y_{i+1}$ (확률 3/4)이므로

X[a]=G1[a]⊕PL[a] (확률 1/2+1/4)

Y[a]=G2[a]⊕G1[a] (확률 1/2+1/4)

이다. 또 $a \xrightarrow{G} c$의 선형근사 확률이 1/2+p1, $a \xrightarrow{G} b$의 선형근사 확률이 1/2 +p2이므로

X[a]=G2[a] (확률 1/2+ p1)

Y[a]=CR[b] (확률 1/2 +p2)

이다. 위의 4개의 선형근사식을 XOR하면 F 함수의 선형근사식을 얻을 수 있다.

PL[a]=CR[b] (확률 1/2 +(2^3)*p1*p2*1/4*1/4)

즉, $(0,a) \xrightarrow{F} (b,($ (확률 1/2 +(2^3)*p1*p2*1/4*1/4)이다.

위와 같은 방법으로 $(b,0) \xrightarrow{F} (0,$인 F 함수의 선형근사식도 얻을 수 있다.

Y[a]=G1[a]⊕G2[a] (확률 1/2+1/4)

CL[a]=G2[a]⊕CR[a] (확률 1/2+1/4)

Y[a]=CR[a] (확률 1/2+ p1)

PR[b]=G1[a] (확률 1/2+ p3)

위의 4개의 선형근사식을 XOR하면, F 함수의 선형근사식을 얻을 수 있다.

PR[b]=CL[a] (확률 1/2 +(2^3)*p1*p3*1/4*1/4)

즉, $(b,0) \xrightarrow{F} (0,$ (확률 1/2 +(2^3)*p1*p3*1/4*1/4)이다. 위의 2가지 형태의 F 함수의 선형근사 $(0,a) \xrightarrow{F} (b,($ $(b,0) \xrightarrow{F} (0,$ 를 결합하면 3라운드 반복 선형근사를 얻을 수 있다. 이 때 반복 선형근사의 확률은 1/2+ (2^7)*p1*p1*p2*p3*1/4*1/4*1/4*1/4*1/4이다. 좋은 반복 선형근사(근사 확률이 0 또는 1에 가까울수록 좋음)를 찾기 위해 p1*p1*p2*p3의 절대값이 큰 것을 찾으면 된다. $a \xrightarrow{G} b$의 선형근사 확률이 1/2 +p2, $b \xrightarrow{G} a$의 선형근사 확률이 1/2+p3이므로, p2*p3의 절대값이 크기 위해서는 b의 Hamming weight는 4이하이어야 하며, 1비트의 위치는 베터 a의 비트가 1이 될 수 있는 곳이어야 한다. 다음 표는 a=0x00000030, a=0x00300000일 때 가능한 b를 찾아 2^32|p1*p1*p2*p3|의 값을 계산한 것이다.

사 확률이 1/2+p3이므로, p2×p3의 절대값이 크기 위해서는 b의 Hamming weight는 4이하이어야 하며, 1비트의 위치는 벡터 a의 비트가 1이 될 수 있는 곳이어야 한다. (표 3-10)은 a=0x00000030, a=0x00300000일 때 가능한 b를 찾아 2^32|p1×p1×p2×p3|의 값을 계산한 것이다.

a	b	a	b	2^32×|p1×p1×p2×p3|
	0x00000010		0x00100000	**4×4×160**
	0x00000020		0x00200000	4×4×56
	0x00000030		0x00300000	4×4×16
	0x00000040		0x00400000	4×4×24
	0x00000050		0x00500000	4×4×0
	0x00000060		0x00600000	4×4×0
	0x00000070		0x00700000	4×4×0
0x00000030	0x00000080	0x00300000	0x00800000	4×4×0
	0x00000090		0x00900000	4×4×36
	0x000000a0		0x00a00000	4×4×4
	0x000000b0		0x00b00000	4×4×8
	0x000000c0		0x00c00000	4×4×4
	0x000000d0		0x00d00000	4×4×8
	0x000000e0		0x00e00000	4×4×8
	0x000000f0		0x00f00000	4×4×140

(표 3-10) 2^32|p1×p1×p2×p3|의 값

위의 (표 3-10)에서 a가 0x00000030일 때 2^32×|p1×p1×p2×p3|이 가장 큰 b는 0x00000010, a가 0x00300000일 때 2^32×|p1×p1×p2×p3|가 가장 큰 b는 0x00100000이며, 이 때 2^32×|p1×p1×p2×p3|의 값은 4×4×160이다. 같은 방법으로 다른 a에 대해서 2^32×|p1×p1×p2×p3|이 최대가 되는 b를 다음 (표 3-11)과 같이 구할 수 있다.

a	b	2^32× |p1×p1×p2×p3|
0x00000030	0x00000010	4×4×160
0x00000060	0x000000c0	2×2×56
0x000000c0	0x00000080	2×2×120
0x00003000	**0x00001000**	**12×12×168**
0x00006000	0x00007000	4×4×100
0x0000c000	0x0000c000	12×12×144
0x00300000	0x00100000	4×4×160
0x00600000	0x00c00000	2×2×56
0x00c00000	0x00800000	2×2×120
0x30000000	**0x10000000**	**12×12×168**
0x60000000	0x70000000	4×4×100
0xc0000000	0xc0000000	12×12×144

(표 3-11) 2^32× |p1×p1×p2×p3|이 최대가 되는 b

따라서 p1×p1×p2×p3의 절대값이 가장 큰 a, b는 다음과 같다.

a=0x00003000, b=0x00001000, 또는 a=0x30000000, b=0x10000000

이 때, 2^32× |p1×p1×p2×p3|의 값은 12×12×168이다.

이 경우 $(0,a) \xrightarrow{F} (b,0)$ 의 선형근사 확률은

1/2+2^3×p1×p2×1/4×1/4

= 1/2+2^3×12/2^8×12/2^8×1/4×1/4

= 1/2+9/2^13이며,

$(b,0) \xrightarrow{F} (0,a)$ 의 선형근사 확률은

1/2+2^3×p1×p3×1/4×1/4

= 1/2+2^3×12/2^8×14/2^8×1/4×1/4

= 1/2+21/2^14이다.

4. SEED 초안의 선형근사

위에서 구한 F 함수의 선형근사를 이용하여, 다음과 같이 2개의 SEED에 대한 3라운드 반복선형근사를 얻을 수 있다.

(0x00001000, 0x00000000) ← (0x00000000, 0x00003000)
(확률 1/2+9/2^13)
(0x00000000, 0x00003000) ← (0x00001000, 0x00000000)
(확률 1/2+21/2^14)
(0x10000000, 0x00000000) ← (0x00000000, 0x30000000)
(확률 1/2+9/2^13)
(0x00000000, 0x30000000) ← (0x10000000, 0x00000000)
(확률 1/2+21/2^14)

위의 3라운드 반복선형근사의 확률은

$1/2+2\times 9/2^{13}\times 21/2^{14}$

$= 1/2+189/2^{26}$

$= 1/2+1/2^{18.44}$ 이다.

이 3라운드 반복선형근사를 이용하여, 다음 (표 3-12)와 같이 n라운드 선형근사확률을 구할 수 있다.

라운드 수	선형근사확률	LC 복잡도
4	1/2+1/2^18.44(=1/2+189/2^26)	2^36.88
5	1/2+1/2^27.05(=1/2+3,969/2^39)	2^54.1
6		2^71.76
7	1/2+1/2^35.88	2^71.76
8	1/2+1/2^44.49	2^88.98
9	1/2+1/2^53.32	2^106.64
10	1/2+1/2^53.32	2^106.64
11	1/2+1/2^61.93	2^123.86
12	1/2+1/2^70.76	2^141.52
13	1/2+1/2^70.76	2^141.52
14	1/2+1/2^79.37	2^158.74
15	1/2+1/2^88.2	2^176.4
16	1/2+1/2^88.2	2^176.4

(표 3 12) SEED 초안의 n라운드 선형근사확률

SEED 초안은 16라운드이므로, LC를 이용하여 공격하는데 필요한 복잡도는 2^176.4이다. 즉, SEED 초안은 LC 공격에 안전한 것으로 예측된다.

제 3 절 SEED 수정안에 대한 DC 및 LC 분석

이제 개선된 G 함수를 사용할 때의 DC 및 LC에 대한 복잡도의 대략적인 bound를 구해 보자. 우선 F 함수의 구조상 임의의 F 함수 input difference에 대해서도 3개의 G 함수 중 최소한 2개의 G 함수가 active하게 된다(즉, nonzero input difference가 최소한 2개의 G 함수로 들어가게 된다). 그런데 G 함수는 diffusion order가 최소한 4이므로 두 개의 G 함수를 연속해서 통과하면 최소한 4개의 S-box가 nonzero 입력을 받게 된다. 예를 들어 첫 G 함수의 32비트 입력 중 임의의 한 비트라도 nonzero이면 S-box에 의해 최소 한 바이트가 nonzero가 되고, 다시 연속된 bit-permutation에 의해 이 한 바이트가 최소 3바이트로 확산된다. 따라서 두 번째 G 함수의 입력 바이트는 최소 3바이트가 nonzero가 되어 3개의 S-box가 active하게 된다.

결론적으로 가장 rough한 estimation으로도 F 함수의 DC 및 LC 확률은 최소 2^{-6×4}=2^{-24}을 갖게 된다. 그러나 실제로 S-box의 characteristic 및 bit-permutation의 성질상 이 확률을 갖는 characteristic이 생길 가능성은 매우 희박하며, 설령 존재하더라도 덧셈에 의한 carry propagation으로 인해 그런 characteristic을 찾는 것은 거의 불가능하다. 즉, F 함수의 최대 확률 2^{-24}은 4개의 active S-box들이 모두 2^{-6}의 최대 확률을 가지며, 각 G 함수의 S-box output difference의 nonzero byte값들이 모두 동일한 값을 가지며 이 값이 bit-permutation의 diffusion order가 4인 30개의 입력 값 중에 하나가 되어야 하고, 첫 G 함수 출력이 32비트 덧셈을 통과한 후 다음 G 함수의 S-box 입력에서도 difference가 다른 바이트로 확산되지 않고 동일한 값으로 유지되는 경우에 국한된다(이 모두에서 덧셈에 의한 확률은 고려되지 않았슴).

다음으로 F 함수 확률을 이용한 n 라운드 확률을 추정해 보자. 최대 2^{-24}의 확률을 갖는 F 함수 characteristic이 최상의 3라운드 iterative characteristic을 주는 (A, B) → (A, B) 혹은 (A, B) → (C, D) & (C, D) → (A, B) 형태를 가질 가능성 역시 매우 희박하며, 마찬가지로 설령 존재하더라도 이를 찾는 것은 거의 불가능하다. 설령 이런 characteristic이 존재하고 또한 찾았다고 가정하더라도 10라운드 characteristic 확률은 $2^{-24\times 6}=2^{-144}$ 이하이다. G 함수의 우수한 diffusion 성질상 다수의 characteristic들이 모여 최상의 differential을 형성할 가능성 역시 매우 희박하다. 결론적으로, F 함수의 자세한 특성을 고려하지 않더라도 (사실 자세히 분석하는 것이 매우 어려울 것임) S-box, bit-permutation 및 addition 등에 대한 최악의 경우를 가정한 10라운드 characteristic이 2^{-144} 이하의 확률을 가지므로, 여기에 4라운드 정도의 safety margin을 고려하더라도 16라운드면 충분한 안전도를 제공할 수 있을 것으로 생각된다.

제 4 절 난수 통계특성

1. 개요

여기에서는 난수생성기(random number generator)의 성질을 조사하기 위한 몇몇 테스트를 소개한다. 이러한 테스트는 난수생성기가 가지는 약점은 알려주지만 그것이 좋은 난수생성기라는 수학적인 증명을 의미하지는 않는다. 이것은 생성된 표본출력수열(sample output sequence)에 몇몇 테스트를 적용하여 이루어진다. 각각의 통계적 테스트는 난수수열이 가져야 하는 특정한 속성을, 표본출력수열이 가지는 지의 여부를 판별하는 것이며, 그 결과는 결정적이기보다 확률적이다. 특정한 속성에 대한 테스트를 만족하지 못한다면 그 난수생성기는 비-난수(non-random)로 판별되거나 더 많은 다른 테스트를 거쳐야 한다. 한편 모든 테스트를 통과한다면 그 생성기(generator)는 난수로 받아들일 수 있다. 엄밀하게 말하면 "받아들인다(accepted)"는 말은 "거부하지 않는다(not rejected)"를 뜻한다. 왜냐하면 이러한 테스트를 통과한다는 것은 생성된 수열이 특정한 몇 개의 난수 수열의 성질은 만족한다는 사실만을 의미하기 때문이다.

가. 정규분포 (Normal distribution)

1) 확률변수 X의 밀도함수 $f(X)$가 $f(X)=\frac{1}{\sigma\sqrt{2\pi}}\exp(\frac{-(x-\mu)^2}{2\sigma^2})$ 일 때 이를 정규분포라 하고, 이 분포는 평균 μ, 분산 σ^2을 가지며 $N(\mu,\sigma^2)$로 표현한다. 특히, $N(0,1)$을 표준정규분포라고 하고, 임의의 $N(\mu,\sigma^2)$에 대해 $Z=\frac{X-\mu}{\sigma}$는 표준정규분포가 된다.

2) X가 $N(0,1)$일 때 $P(X>x)=\alpha$을 만족하는 α, x 표.

α	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
χ	1.2816	1.6449	1.9600	2.3263	2.5758	2.8070	3.0902	3.2905

(표 3 13) X가 $N(0,1)$일 때,
$P(X>x)=\alpha$을 만족하는 α, x

나. 카이제곱 분포 (Chi-square distribution : χ^2)

1) 확률밀도함수가 다음 식으로 주어지는 분포를 자유도 v인 카이제곱 분포라 한다.

$$f(X)=\begin{cases} \frac{1}{\Gamma(v/2)2^{v/2}}x^{(v/2)-1}e^{-x/2}, & 0\le x<\infty, \\ 0 & x<0 \end{cases}$$

이 분포는 평균 v, 분산 $2v$를 가진다. 위에서 $\Gamma()$는 gamma function이다.

$$\Gamma(t)=\int_0^\infty x^{t-1}e^{-x}dx \quad \text{for } t>0$$

2) $X_1, X_2, \cdots, X_n$ 이 표준정규분포이면, $Z=\sum_{i=1}^{n}X_i^2$ 은 자유도 n인 χ^2분포가 된다.

다. 통계적 가설검정

1) 주어진 표본출력수열이 난수생성기에 의해 생성되었다는 가설을 H_0라고 하자. 이 때 H_0에 대한 유의수준 α가 너무 크면 주어진 수열이 난수임에도 불구하고 테스트가 그 수열을 "거부하는" 오류

가 생긴다. 이러한 오류를 Type I error라고 한다. 한편, 유의수준 α가 너무 작으면 주어진 수열이 난수가 아닌 경우에도 테스트가 그 수열을 "받아들이는" 오류가 생긴다. 이러한 오류를 Type II error라고 한다. 그러므로 가설검정에서는 유의수준을 신중하게 정해야 하며, 일반적으로 유의수준 α는 $0.001 \leq \alpha \leq 0.05$이 되도록 선택한다.

2) 임의의 통계치 X가 $N(0,1)$을 만족하고, 유의수준 α가 주어졌다고 하자.

먼저 (표 3-13) 등을 이용하여 $P(X > x_\alpha) = P(X < -x_\alpha) = \alpha/2$ 을 만족하는 x_α을 찾는다. 주어진 표본출력수열 S의 통계치 X_S 가 $X_S > x_\alpha$ 또는 $X_S < -x_\alpha$ 을 만족하면, 이 테스트는 S를 거부한다. 한편, X_S 가 $-x_\alpha \leq X_S \leq x_\alpha$ 를 만족하면, 이 테스트는 S를 받아들인다. 이러한 테스트를 양측검정이라고 한다.

일례로, $\alpha = 0.05$ 일 경우 $x_\alpha = 1.96$ 이고 5%의 확률로 난수 수열을 거부할 것이 예상된다.

3) 임의의 통계치 X가 자유도 v인 χ^2이고, 유의수준 α가 주어졌다고 하자.

먼저 $P(X > x_\alpha) = \alpha$ 를 만족하는 x_α를 찾는다. 주어진 표본출력수열 S의 통계치 X_S가 $X_S > x_\alpha$ 를 만족하면, 이 테스트는 S를 거부한다. 한편 X_S가 $X_S \leq x_\alpha$ 를 만족하면, 이 테스트는 S를 받아들인다. 이러한 테스트를 단측검정이라고 한다.

일례로, $v = 5, \alpha = 0.025$ 일 경우 $x_\alpha = 12.8325$ 이고 2.5%의 확률로 난수 수열을 거부할 것이 예상된다.

2. 난수통계테스트의 방법

가. 일반적인 통계테스트의 종류

○ 먼저 $s = s_0, s_1, \cdots, s_{n-1}$ 을 길이 n인 2진 수열이라 하자.

1) Frequency test (mono-bit test)

- 0과 1의 개수가 같다는 가정에 근거한 테스트.
- 0과 1의 개수를 각각 n_0, n_1 이라고 하면, $n \geq 10$인 경우

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

은 자유도(degree of freedom)가 1인 χ^2 분포를 따른다.

- 이 때, $n_0 + n_1 = n$ 을 만족한다.

※ Uniformity Test, Equidistribution Test라고도 불림

2) Runs distribution test

(방법 1)

길이가 i인 run에 대해 1의 run(block)의 개수와 0의 run(gap)의 개수를 각각 Bi, Gi라고 하면, 길이 i인 block(gap)의 개수의 기대값 e_i는 $e_i = \frac{\frac{n-i-1}{2}+2}{2^{i+1}}$ 이고,

$$\chi^2 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$ 은

자유도 2k-2인 χ^2 분포를 따른다.

(방법 2)

방법 1과 같으나, 기대값으로 $e_i = \frac{n}{2^{i+2}}$를 사용한다.

3) Serial test (two-bit test)

- 연속된 2비트 00, 01, 10, 11의 개수가 같다는 가정에 근거한 테스트.
- 00, 01, 10, 11의 개수를 각각 $n_{00}, n_{01}, n_{10}, n_{11}$이라고 하면, $n \geq 21$인 경우

$$X_2 = \frac{4(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2)}{n-1} - \frac{2(n_0^2 + n_1^2)}{n} + 1$$

은 자유도(degree of freedom)가 2인 χ^2분포를 따른다.

- 이 때, $n_{00} + n_{01} + n_{10} + n_{11} = n-1$, $n_0 + n_1 = n$을 만족한다.

4) Poker test

- 길이가 n인 전체 비트 스트림을 길이가 b인 subblock들로 나누었을 때, 각 subblock에 포함된 1의 개수가 균일하게 분포되었는가를 테스트.
- 양의 정수 m이 $\lfloor n/m \rfloor \geq 5 \cdot 2^m$을 만족하면, 양의 정수 $k = \lfloor n/m \rfloor$에 대해 수열 s를 각 m비트인 k개의 부분수열로 둘 수 있고, 각각의 부분수열 α는 m비트 수이므로 $0 \leq \alpha < 2^m$을 만족한다. 이 때, k개의 부분수열이 $1, 2, \cdots, 2^m - 1$ 을 나타내는 빈도를 n_i 라 하자. 그러면

$$X_3 = \frac{2^m}{k}\left(\sum_{i=0}^{2^m} n_i^2\right) - k$$

는 자유도 $2m-1$인 χ^2 분포를 따른다.

- $\lfloor n/m \rfloor \geq 5 \cdot 2^m$를 만족하기 위해서는 $m=4,8,12$ 일 때, n은 각각 320=40바이트, 10240=1255바이트, 485760=60720바이트 이상이 되어야 한다.

※ Hamming weight Test라고도 불림

5) Change point test

비트 스트림에서 t번째 비트에 대해 그 이전 1의 비율과 그 이후 1의 비율을 비교하여 그 차이가 최대인 점을 찾는 방법이다. 이 테스트의 가정은 전체 스트림에서 1의 비율에 변화가 없다는 것이다. 이 가설을 검정하기 위한 통계량은 $U[t] = nS[t] - tS[n]$이다. 여기서, t = 1, ... , n-1이고, n은 전체 스트림의 비트 수이고, S[n]은 스트림의 전체 1의 개수이고, S[t]는 t비트까지의 1의 개수이다.

U[t]의 절대값 중 최대인 것을 M이라고 하면 즉, $M = Max|U[t]|$이면, tail area probability α는 $\alpha = e^{-\frac{2M^2}{nS[n](n-S[n])}}$ 가 된다.

이 α의 값은 단측 검정에 적용되는 값이므로, 양측 검정에 적용할 때는 2α를 사용한다.

6) Binary derivative test

길이 n인 비트 스트림의 binary derivative란 연속하는 두 비트를 modulo-two addition하여 생성되는 새로운 길이 n-1의 비트 스트림을 말한다. 예를 들면 비트 스트림 S에서, 첫번째 binary derivative d1(S)는 다음과 같다.

S = 0 1 1 0 1 0 0 0 1 0 0 1 1 1

d1(S) =1 0 1 1 1 0 0 1 1 0 1 0 0

여기서, dk(S)를 S의 k번째 binary derivative라고 한다.

각각의 Binary derivative와 원래의 비트 스트림에 대해 frequency test를 적용한다.

6) Runs test

- 0 또는 1이 연속하여 나타나는 정도가 추정치에 근접한지를 확인하는 테스트.
- n비트의 수열에서 0 또는 1이 i개 연속하여 나타날 확률은 $e_i=(n-i+3)/2^{i+2}$ 이다. 먼저 $e_{k-1}<5\leq e_k$ 를 만족하는 k를 결정하고, 길이가 i인 Block(1이 연속된 것)과 Gap(0이 연속된 것)의 개수를 B_i, G_i 라고 하자. 그러면

$$X_1=\sum_{i=1}^{k}\frac{(B_i-e_i)^2+(G_i-e_i)^2}{e_i}$$

는 자유도 $2k-2$인 χ^2 분포를 따른다.

7) Autocorrelation test

- 일정한 간격차이의 두 비트들 간의 관계를 테스트, 즉, 비트 스트림에서 일정 거리만큼 떨어진 비트들관의 상관성을 검증하는 테스트.
- $n/2$미만의 양의 정수 d에 대해 $A(d)=\sum_{i=0}^{n-d-1}s_i\veebar s_{i+d}$ 라 두자. 이때, 연산 $\veebar$는 배타적 논리합 XOR이다. 그러면 $n-d\geq 10$ 인 경우

$$X_5=\frac{2A(d)-(n-d)}{\sqrt{n-d}}$$

는 정규분포 $N(0,1)$을 따른다. 이 경우는 양방향 테스트를 한다.

8) Maurer's universal statistical test

- 기본 아이디어는 주어진 비트 수열이 난수라면 그 수열을 압축하였을 때 그 압축률이 너무 크지는 않을 것이라는 사실이다. 즉, 어떤 수열을 압축하였을 때 큰 압축률을 보인다면 그 수열을 거부한다. 이 테스트는 실제로 주어진 수열을 압축하는 대신 압축률에 관련된 값만을 계산하므로 그 실행속도는 비교적 빠르다. 그러나 매우 큰 비트 수열을 필요로 하기 때문에 난수생성기가 표준출력수열을 생성하는데 비교적 많은 시간이 소요된다.
- 임의의 $L \in [6,7,\cdots,16]$ 에 대해 주어진 2진 수열 s를 $Q+K$ 개의 L-비트 블록으로 분리하여 각 블록을 $b_1, b_2, \cdots, b_{Q+K}$라 하자. $0 \le b_i < 2^L$을 만족하고, Q, K는 $Q \ge 10 \cdot 2^L$, $K \ge 1000 \cdot 2^L$이 되도록 한다. 먼저 $k = 1, \cdots, Q$에 대해 $T[j]$, $0 \le j < 2^L$, 에 $b_k = j$ 를 만족하는 최대의 k값을 저장한다. 즉, $T[j]$는 가장 최근에 j값이 나타난 블록의 위치를 나타낸다. 이후로도 $T[j]$는 j값이 나타난 최근의 블록위치를 나타낸다. 그리고 $A_i = i - T[b_i], i = Q+1, \cdots, Q+K$ 라고 하면, A_i는 b_i값이 몇 블록 이전에 나타났는 지를 의미한다. 그러면,

$$X_u = \frac{1}{K} \sum_{i=Q+1}^{Q+K} \ln A_i$$

는 L에 따라 아래 표의 μ, σ_1에 대해 평균 μ와 분산 σ^2을 가지는 정규분포를 따른다. 이때

$$\sigma^2 = (0.7 - (0.8/L) + (1.6 + 12.8/L) \cdot K^{-4/L})^2 \cdot \sigma_1^2 / K$$

이다.

L	μ	σ_1^2	L	μ	σ_1^2	L	μ	σ_1^2	L	μ	σ_1^2
1	0.7326495	0.690	5	4.2534266	2.705	9	8.1764248	3.311	13	12.168070	3.410
2	1.5374383	1.338	6	5.2177052	2.954	10	9.1723243	3.356	14	13.167693	3.416
3	2.4016068	1.901	7	6.1962507	3.125	11	10.170032	3.384	15	14.167488	3.419
4	3.3112247	2.358	8	7.1836656	3.238	12	11.168765	3.401	16	15.167379	3.421

- 이 테스트는 비교적 큰 표본출력수열을 필요로 하는데, 표본출력수열의 비트수 n은

$$n = L \cdot (Q+K) \geq L \cdot (10 \cdot 2^L + 1000 \cdot 2^L) = L \cdot 1010 \cdot 2^L$$

을 만족하여야 한다. 이는 $L = 6,7,8$일 때, n은 각각 387840 = 48480바이트, 904960 = 113120바이트 그리고 2068480 = 258560 바이트 이상이 되어야 한다.

나. FIPS 140-1 난수성 테스트

1) FIPS 140-1(Security requirements for cryptographic modules)은 난수성을 테스트하기 위한 4개의 통계테스트를 언급한다. 여기서는 유의수준 대신 통계치의 상하 한계값을 주어 이를 만족할 것을 요구한다.

2) 20000비트의 표본출력수열이 다음 4개의 테스트를 통과하여야 한다.

- Mono-bit test : (2.1.가)의 테스트에서 $9654 < n_1 < 10346$을 만족한다.
- poker test : (2.1.다)의 테스트에서 $m=4$인 경우

  $1.03 < X_3 < 57.4$을 만족한다.
- runs test : (2.1.라)의 테스트에서 B_i, G_i, $1 \le i \le 6$, 값이 아래 표의 범위에 있어야 한다.

  (단, B_6, G_6 각각은 i가 6이상인 모든 B_i, G_i의 합이다.)

i	B_i, G_i의 범위
1	2267 - 2733
2	1079 - 1421
3	502 - 748
4	223 - 402
5	90 - 223
6	90 - 223

- long run test : (2.1.라) 테스트에서 $B_i=0$, $G_i=0$, $i \ge 34$을 만족해야 한다. 즉, 0이나 1이 34개 이상 연속되지 않아야 한다.
- FIPS 140-1은 고도의 안전성을 보장하기 위해 난수생성기의 개선시 위의 4개의 통계테스트를 적용하도록 규정하고 있다. 또한 FIPS 140-1은 위의 통계테스트를 또 다른 통계테스트로 대치하는

것을 허용하지만, 이는 더욱 강력한 난수성 통계조사임이 보장되어야 한다.

다. Avalanche Effect, SAC

블록 암호 알고리즘에서 평문의 한 비트가 바뀌었을 때 평균적으로 암호문의 비트들의 반이 바뀌었을 때, 블록 암호 알고리즘은 avalanche 특성을 만족한다고 한다. 또, 평문의 한 비트가 바뀌었을 때 암호문의 각각의 비트가 바뀔 확률이 1/2이 된다면 블록 암호 알고리즘은 strict avalanche 특성을 만족한다고 하며, 이러한 성질을 블록 암호 알고리즘이 만족하는 가를 테스트하는 것은 중요하다고 할 수 있다.

예를 들어, 블록 암호 알고리즘의 블록 크기가 128비트이고 평문의 한 비트가 바뀌었을 때 암호문에서 바뀌어진 비트들의 개수를 random variable X라고 하자. 만일 좋은 블록암호알고리즘이라면 X는 Bernoulli random variable을 따르게 된다. 따라서 X의 평균은 64가 되며, 표준편차는 $\frac{4\sqrt{2}}{\sqrt{N}}$가 된다. 여기서 N은 X의 표본의 수가 된다.

만일, 유의수준 0.3%로 검정을 하면 $|X-64| \leq \frac{4\sqrt{2}}{\sqrt{N}} \times 3$일 때, avalanche 성질을 만족한다고 할 수 있다.

Strict avalanche 특성을 측정하기 위해 v_{ij}를 평문 i번째 비트가 바뀌었을 때, 암호문의 j번째 비트가 바뀌면 1의 값을 가지고 그렇지 않을 경우 0의 값을 가지는 random variable이라고 하자. 전체 샘플의 개수가 S일 경우, 통계량 $z_{ij} = 2\sqrt{S}\,(a_{ij} - \frac{1}{2})$는 표준정규분포 N(0, 1)을 따르게 된다(단, $a_{ij} = \frac{1}{S} \sum_{p=1}^{S} v_{ij,p}$).

3. 난수통계특성 테스트 결과

길이가 n인 비트 스트림이 각각의 비트들이 0과 1이 나올 확률이 같은 n번의 독립적인 Bernoulli trial로 구성되어 있는 가를 검정하는 방법인 frequency, runs, runs distribution, autocorrelation, poker, universal, serial, binary derivative test에 대해 alternative hypothesis H1과 H2를 따르는 길이 n = 10,240인 비트 스트림 10,000개를 사용하여 유의 수준 5%와 1%에 대해 power function을 실험적으로 구하였다.

비트 스트림을 일정한 길이의 subblock단위로 나누어서 테스트하는 경우인 frequency test를 subblock에 적용하는 방법, poker test, universal test의 경우에는 subblock의 길이 b = 4, 6에 대해 각각 universal test를 적용하기 위한 최소 비트 스트림의 길이인 n = 102,400, 387,840인 비트 스트림 1,000개를 사용하였다. Autocorrelation test의 경우에는 frequency test와의 비교를 위해 n = 20,480이고, 거리 d = 10,240인 비트 스트림을 사용하였다.

실험의 결과를 (표 3-14 ~ 3-15) 에 제시하였다.

가. 실험 방법

블록 암호 알고리즘은 스트림 암호 알고리즘과 같이 키 스트림의 randomness를 테스트할 필요가 없으므로 위에서 설명한 randomness test방법을 그대로 적용할 수가 없다. 따라서, 블록 암호 알고리즘의 평문과 암호문 사이의 독립성을 검정하기 위한 다음과 같은 방법에 randomness test방법을 사용한다.

1) 방법 1

Non-random한 비트 스트림은 알고리즘 평문으로 입력할 경우, 만일 평문과 암호문이 독립성을 가지고 있다면, 암호문은 random한 비트 스트림이 될 것이다. 따라서 이 암호문에 대해 난수통계특성을 수행한다.

2) 방법 2

random한 비트 스트림은 알고리즘 평문으로 입력할 경우 생성되는 평문과 암호문을 bitwise-Xor한 결과는 만일 평문과 암호문이 독립성을 가지고 있다면, random한 비트 스트림이 될 것이다. 따라서 이 비트 스트림에 대해 난수통계특성을 수행한다.

나. 실험 결과

	SEED 초안	SEED 수정안
Avalanche Test	4라운드 이후 만족	2라운드 이후 만족
SAC Test	4라운드 이후 만족	2라운드 이후 만족
난수성 테스트	2라운드 이후 만족	2라운드 이후 만족

SEED 초안(유의 수준 0.3%)		SEED 수정안(유의 수준 0.3%)	
라운드 수	63.9933≤X≤64.0067	라운드 수	63.9933≤X≤64.0067
2	63.3396	2	63.9962
4	63.9987	4	64.0028
6	64.0059	6	63.9996
8	63.9995	8	64.0024
10	64.0014	10	63.9962
12	64.0011	12	64.0015
14	63.9993	14	63.9968
16	63.9957	16	63.9975

(표 3-14) SEED의 Avalanche effect

SEED 초안(유의 수준 0.3%)		SEED 수정안(유의 수준 0.3%)	
라운드 수	threshold = 49.152	라운드 수	threshold = 49.152
2	678	2	29
4	45	4	46
6	38	6	47
8	50	8	48
10	48	10	33
12	34	12	44
14	44	14	46
16	54	16	43

(표 3-15) SEED의 SAC test

	라운드수	Frequency Test		Run Distribution Test		Frequency Test on subblock(b=2)	
		% of failed sample		% of failed sample		% of failed sample	
		유의수준		유의수준		유의수준	
		5%	1%	5%	1%	5%	1%
방법 1	2	2.82	0.58	7.92	1.84	3.03	0.45
	4	4.26	1.02	7.15	1.68	5.15	0.99
	6	4.29	1.20	7.40	1.73	5.15	1.02
	8	4.05	1.06	7.16	1.71	4.92	1.05
	10	4.13	0.92	7.13	1.82	4.94	1.02
	12	4.06	0.90	7.57	1.82	4.94	1.02
	14	4.12	1.03	7.10	1.68	5.09	1.01
	16	3.91	0.96	7.20	1.65	4.96	0.91
방법 2	2	4.30	0.92	7.28	1.64	4.99	0.89
	4	4.33	1.03	7.29	1.88	4.93	1.10
	6	5.93	1.30	7.28	1.54	4.95	0.92
	8	4.10	0.86	7.72	1.93	4.89	0.94
	10	4.03	0.99	7.39	1.86	5.09	0.87
	12	4.20	1.20	7.52	1.87	5.03	0.96
	14	4.17	1.05	7.30	1.82	5.31	1.11
	16	4.53	1.26	7.81	2.08	5.61	1.29

(표 3-16) SEED 초안의 통계적 특성 분석 I

	라운드수	Poker Test(b=4)		Change point Test		1-Binary derivative Test	
		% of failed sample		% of failed sample		% of failed sample	
		유의수준		유의수준		유의수준	
		5%	1%	5%	1%	5%	1%
방법 1	2	3.43	0.82	3.54	0.64	3.42	0.63
	4	5.14	1.28	4.24	0.82	4.61	0.89
	6	5.13	1.23	4.02	0.77	4.90	1.15
	8	5.01	1.18	3.72	0.65	4.62	1.05
	10	5.12	1.07	4.26	0.78	4.44	1.02
	12	5.06	1.30	4.03	0.64	4.55	0.90
	14	5.19	1.45	3.95	0.65	4.60	1.02
	16	4.94	1.25	3.97	0.87	4.52	1.02
방법 2	2	5.60	0.15	3.87	0.73	4.51	0.89
	4	5.47	1.37	4.00	0.64	4.93	1.18
	6	5.09	1.31	4.02	0.67	4.70	1.06
	8	4.88	1.28	3.91	0.69	4.66	0.98
	10	5.14	1.26	3.84	0.79	4.69	1.00
	12	4.83	1.22	4.23	0.82	4.85	1.09
	14	5.27	1.33	3.91	0.82	4.67	1.10
	16	5.36	1.35	4.03	0.79	5.25	1.25

(표 3-17) SEED 초안의 통계적 특성 분석 II

	라운드수	Frequency Test		Run Distribution Test		Frequency Test on subblock(b=2)	
		% of failed sample		% of failed sample		% of failed sample	
		유의수준		유의수준		유의수준	
		5%	1%	5%	1%	5%	1%
방법 1	2	3.84	0.88	6.72	1.70	4.33	0.65
	4	4.36	1.01	7.17	1.74	5.07	0.87
	6	4.38	0.92	7.36	1.79	4.81	0.95
	8	4.02	0.97	6.96	1.79	4.81	1.04
	10	4.00	1.00	7.33	1.64	4.87	0.88
	12	3.72	0.94	7.06	1.57	4.84	0.97
	14	4.30	1.10	7.29	1.83	5.18	1.07
	16	4.29	1.07	7.18	1.87	5.15	1.02
방법 2	2	4.07	0.95	7.68	1.84	4.94	0.99
	4	3.96	0.87	7.12	1.61	5.20	0.83
	6	4.46	1.13	7.35	1.86	5.36	0.89
	8	4.77	1.26	7.23	1.66	5.08	1.03
	10	4.11	1.00	7.64	1.82	4.86	0.95
	12	4.27	0.89	7.18	1.72	4.97	0.87
	14	4.05	1.02	7.40	1.72	4.87	0.88
	16	4.23	0.93	6.91	1.85	4.77	0.87

(표 3-18) SEED 수정안의 통계적 특성 분석 I

	라운드수	Poker Test(b=4)		Change point Test		1-Binary derivative Test	
		% of failed sample		% of failed sample		% of failed sample	
		유의수준		유의수준		유의수준	
		5%	1%	5%	1%	5%	1%
방법 1	2	4.58	1.04	3.71	0.61	4.18	0.77
	4	5.17	1.18	3.86	0.65	4.97	1.09
	6	4.96	1.22	3.71	0.61	4.52	0.88
	8	5.30	1.34	4.24	0.85	4.64	1.10
	10	4.94	1.13	4.01	0.80	4.31	0.97
	12	4.81	1.17	3.79	0.62	4.56	0.97
	14	5.06	1.40	4.28	0.73	4.77	0.97
	16	5.08	1.19	4.02	0.82	4.68	1.03
방법 2	2	5.16	1.44	4.10	0.70	4.78	1.03
	4	4.95	1.17	3.84	0.82	4.57	0.89
	6	5.44	1.47	4.04	0.71	4.91	1.09
	8	5.06	1.43	4.39	0.85	4.87	1.08
	10	4.98	1.17	3.97	0.84	4.70	0.96
	12	5.37	1.48	3.78	0.65	4.49	0.93
	14	5.19	1.32	4.09	0.71	4.59	1.01
	16	4.83	1.27	3.88	0.73	4.51	1.01

(표 3-19) SEED 수정안의 통계적 특성 분석 II

제 5 절 키 생성 알고리즘 분석

암호키는 키생성 알고리즘에 의해 암/복호화 과정에서 필요로 하는 서브키의 형태로 변환된다. 이 변환알고리즘을 분석함으로써 암호알고리즘 해독에 소요되는 노력을 줄일 수 있다. 주로 weak key, semi-weak key, equivalent key, linear factor, complementation property 등의 성질을 이용하여 exhaustive key search attack의 복잡도를 감소시키거나 서브키 간의 규칙성(related-key)을 이용하여 DC 등에 필요한 평문의 갯수를 줄일 수 있다. 키생성 알고리즘에 의한 공격방법은 대개 비실제적이지만 경우에 따라 암호의 응용범위에 제약을 주는 요인이 될 수 있다. 가령, related key나 weak key가 많은 암호를 해쉬함수의 구성에 사용한다면 해쉬함수의 충돌쌍을 쉽게 찾을 수 있다.

본 절에서는 SEED 수정안의 키생성 알고리즘에서 weak key, semi-weak key, complementation property, equivalent key등의 특성과 differential related-key attack, related-key slide attack등의 적용가능성을 살펴본다.

SEED 수정안의 키생성 알고리즘은 라운드 함수에서 사용하는 G함수와 64비트 회전이동을 사용하여 64비트의 서브키를 생성한다. G 함수는 비선형성이 높은 S-box와 충분한 diffusion을 가지는 bit-wise 연산으로 이루어져 있으며, 128비트의 각 비트가 모든 라운드의 서브키에 영향을 미치도록 설계되었다. SEED 수정안의 키생성 알고리즘은 라운드 상수, S-box, 32비트 덧셈/뺄셈 연산을 사용하여 현재까지 알려져 있는 여러 가지 공격에 대해 안전하도록 설계되었다. 또한 B||A, D||C의 회전이동과 덧셈과 뺄셈을 각 1회만 수행하고 나면, 라운드 함수의 구현에서 사용된 모듈만으로 서브키를 생성할 수 있으므로 효

율적인 구현이 가능하다. 또한 라운드 키 상수의 생성이나 라운드별 서브키 생성방법이 매 라운드마다 동일하므로 자원이 제한적인 스마트카드 등에서도 충분히 구현될 수 있다.

분석에 의하면 SEED 수정안의 키생성 알고리즘에서는 weak key, semi-weak key, complementation property, equivalent key가 존재하지 않는다. 또한 related key slide attack이 불가능하며 라운드 수가 5라운드 이상이 되면 differential related-key attack에 안전하다.

SEED 수정안의 키생성 알고리즘에 사용되는 32비트 A, B, C, D를 $A=a_3a_2a_1a_0$, $B=b_3b_2b_1b_0$, $C=c_3c_2c_1c_0$, $D=d_3d_2d_1d_0$와 같이 16개의 byte로 나타내면 각 라운드에서 서브키를 생성하기 위한 G함수의 입력은 (표 3-20)과 같다. SEED 수정안의 키생성 알고리즘은 2라운드마다 일정한 규칙으로 A, B, C, D를 생성하여 서브키를 내며 8+i 라운드의 A, B, C, D는 i 라운드의 B, A, D, C와 동일하나(표 3-20 참조) $K_{i,0}$, $K_{i,1}$에 대한 G함수 입력조합이 다르므로 서브키가 다르게 생성됨을 알 수 있다. 즉, 1라운드와 9 라운드에서 A, B, C, D값이 그대로 사용되지만 각각 A+C, B-D, B+D, B-D의 형태이므로 생성되는 서브키 $K_{1,0}$, $K_{1,1}$, $K_{9,0}$, $K_{9,1}$의 값은 모두 다르게 된다.

	$K_{i,0}$	$K_{i,1}$
라운드 수	A+C-KCi	B-D+KCi
1	$a_3a_2a_1a_0+c_3c_2c_1c_0-KC_1$	$b_3b_2b_1b_0-d_3d_2d_1d_0+KC_1$
2	$b_0a_3a_2a_1+c_3c_2c_1c_0-KC_2$	$a_0b_3b_2b_1-d_3d_2d_1d_0+KC_2$
3	$b_0a_3a_2a_1+c_2c_1c_0d_3-KC_3$	$a_0b_3b_2b_1-d_2d_1d_0c_3+KC_3$
4	$b_1b_0a_3a_2+c_2c_1c_0d_3-KC_4$	$a_1a_0b_3b_2-d_2d_1d_0c_3+KC_4$
5	$b_1b_0a_3a_2+c_1c_0d_3d_2-KC_5$	$a_1a_0b_3b_2-d_1d_0c_3c_2+KC_5$
6	$b_2b_1b_0a_3+c_1c_0d_3d_2-KC_6$	$a_2a_1a_0b_3-d_1d_0c_3c_2+KC_6$
7	$b_2b_1b_0a_3+c_0d_3d_2d_1-KC_7$	$a_2a_1a_0b_3-d_0c_3c_2c_1+KC_7$
8	$b_3b_2b_1b_0+c_0d_3d_2d_1-KC_8$	$a_3a_2a_1a_0-d_0c_3c_2c_1+KC_8$
9	$b_3b_2b_1b_0+d_3d_2d_1d_0-KC_9$	$a_3a_2a_1a_0-c_3c_2c_1c_0+KC_9$
10	$a_0b_3b_2b_1+d_3d_2d_1d_0-KC_{10}$	$b_0a_3a_2a_1-c_3c_2c_1c_0+KC_{10}$
11	$a_0b_3b_2b_1+d_2d_1d_0c_3-KC_{11}$	$b_0a_3a_2a_1-c_2c_1c_0d_3+KC_{11}$
12	$a_1a_0b_3b_2+d_2d_1d_0c_3-KC_{12}$	$b_1b_0a_3a_2-c_2c_1c_0d_3+KC_{12}$
13	$a_1a_0b_3b_2+d_1d_0c_3c_2-KC_{13}$	$b_1b_0a_3a_2-c_1c_0d_3d_2+KC_{13}$
14	$a_2a_1a_0b_3+d_1d_0c_3c_2-KC_{14}$	$b_2b_1b_0a_3-c_1c_0d_3d_2+KC_{14}$
15	$a_2a_1a_0b_3+d_0c_3c_2c_1-KC_{15}$	$b_2b_1b_0a_3-c_0d_3d_2d_1+KC_{15}$
16	$a_3a_2a_1a_0+d_0c_3c_2c_1-KC_{16}$	$b_3b_2b_1b_0-c_0d_3d_2d_1+KC_{16}$

(표 3-20) 각 라운드에서 G함수의 입력 값

가정 : 평문을 P, 암호키를 K, 암호문을 C라 하자. 암호화 알고리즘을 E, 복호화 알고리즘을 D라 하자. C = E, P = D(C, K).

※ 참고로 A_i, B_i, C_i, D_i는 master key K를 사용하여 i번째 라운드의 서브키를 생성하는 데 사용되는 레지스터 변수 A, B, C, D로 정의하고, master key를 K로 표기하기로 한다.

1. 라운드 키에 대한 암호키의 영향

SEED 수정안 키 생성은 전체 암호키가 균등하게 모든 라운드 키에 영향을 주도록 설계되었다. 암호키 전체가 모든 라운드 키의 생성에 사용되며, 암호키의 모든 바이트가 균등한 영향을 주도록 로테이션 양이 결정되어 있다. 암호키의 각 바이트는 32 비트 연산의 모든 바이트 위치에 2번씩 사용되기 때문에 연산의 비트 위치에 따른 상관관계 정도도 매우 균등하다.

2. Weak Key

Weak Key는 같은 키를 이용하여 두 번 암호화를 수행하면 다시 평문을 얻을 수 있는 키를 말한다. ($P = E_K(E_K(P))$) 즉, i 라운드 키는 17-i 라운드 키와 동일해야 한다. SEED 수정안의 weak key가 존재한다면 1 라운드와 16 라운드 키, 8 라운드와 9 라운드 키가 동일해야 한다. G 함수는 일대일이기 때문에 weak key는 다음 조건을 만족한다.

$$(a3|a2|a1|a0) + (c3|c2|c1|c0) - KC1$$

$$= (a3|a2|a1|a0) + (d0|c3|c2|c1) - KC16$$

(a3|a2|a1|a0) - (d0|c3|c2|c1) + KC8

= (a3|a2|a1|a0) - (c3|c2|c1|c0) + KC9

이 식을 정리하면 다음과 같다.

(c3|c2|c1|c0) - (d0|c3|c2|c1) = KC1 - KC16

(c3|c2|c1|c0) - (d0|c3|c2|c1) = KC9 - KC8

여기서 KC1 - KC16 ≠ KC9 - KC8 이므로, weak key는 존재하지 않는다.

3. Semi-weak Key

Semi-weak Key는 서로 다른 키를 이용하여 두 번의 암호화를 수행하면 다시 평문을 얻을 수 있는 키를 말한다. ($P = E_{K_1}(E_{K_2}(P))$) 즉, K1을 이용한 i 라운드 키와 K2를 이용한 17-i 라운드 키가 동일해야 한다.

(a3|a2|a1|a0) + (c3|c2|c1|c0) - KC1

= (a3′|a2′|a1′|a0′) + (d0′|c3′|c2′|c1′) - KC16

(a3|a2|a1|a0) + (d0|c3|c2|c1) - KC16

= (a3′|a2′|a1′|a0′) + (c3′|c2′|c1′|c0′) - KC1

(a3|a2|a1|a0) - (d0|c3|c2|c1) + KC8

= (a3′|a2′|a1′|a0′) - (c3′|c2′|c1′|c0′) + KC9

(a3|a2|a1|a0) - (c3|c2|c1|c0) + KC9

$$= (a3' | a2' | a1' | a0') - (d0' | c3' | c2' | c1') + KC8$$

1 라운드와 16 라운드, 8 라운드와 9 라운드 키를 비교하면 다음을 계산할 수 있다.

$$(c3 | c2 | c1 | c0) - (d0 | c3 | c2 | c1) + 2KC16$$

$$= (d0' | c3' | c2' | c1') - (c3' | c2' | c1' | c0') + 2KC1$$

$$(c3 | c2 | c1 | c0) - (d0 | c3 | c2 | c1) + 2KC8$$

$$= (d0' | c3' | c2' | c1') - (c3' | c2' | c1' | c0') + 2KC9$$

이 식을 간단히 정리하면 다음과 같다.

$$2KC16 - 2KC8 = 2KC1 - 2KC9$$

여기서 KC16 - KC8 \ KC1 - KC9 이므로 모순이다. 즉, semi-weak key는 존재하지 않는다.

4. Complementation Property

Complementation property란 키 K에 대한 평문 P의 암호문이 K의 bitwise complement인 $\overline{K}$에 의한 $\overline{P}$의 암호문의 bitwise complement와 값이 같은 성질을 말한다. 즉, 암호알고리즘 E가 complementation property를 가지는 경우, $C=E_K(P)$이면 $\overline{C} = E_{\overline{K}}(\overline{P})$ 이다. 서브키와 평문입력이 XOR되는 SEED 수정안과 같은 구조의 암호에서

complementation property가 성립하기 위해서는 서브키 생성과정과 암호함수가 bitwise shifting, bitwise permutation, exclusive-OR 등의 비트별 연산으로 구성되어 있어야 한다. 반면에 S-box, modulo-addition과 같은 연산들을 사용한 경우에는 일반적으로 complement property를 만족하지 않는다.

SEED 수정안의 complement property의 성립여부를 조사하기 위해서는 1 라운드 함수의 서브키 입력의 형태와 함수의 입력을 살펴봐야 한다. 1 라운드의 서브키 64비트는 평문입력 64비트와 XOR되며 $a \oplus b = \overline{a} \oplus \overline{b}$, $a \oplus \overline{b} = \overline{a} \oplus b = \overline{a \oplus b}$이므로 complementation property를 만족하기 위해서는 우선 1라운드 서브키가 complement property를 만족하여야 한다. 그러나 SEED 수정안에서는 서브키를 비선형적인 출력을 내는 S-box를 사용하므로 $\overline{K}$에 의해 생성된 서브키들은 K에 의해 생성된 서브키들과는 무관하다. 따라서 complementation property를 만족하지 않는다. 한편, 서브키 생성방법이 complementation property를 만족한다 하더라도 F함수 내에서 modulo-addition이 사용되므로 complementation property를 만족하지 않게 된다.

5. Equivalent Key

Equivalent key란 평문(P)를 서로 다른 두 개의 키로 암호화하였을 때 생성되는 두 개의 암호문이 동일한 경우 사용되는 두 개의 키를 말한다. 즉, 두 개의 서로 다른 키 K, K*에 대해 $C = E_{K(P)} = E_{K^*}(P)$ 이다. K와 K*가 equivalent key가 되기 위해서는 각각에 의해 생성된 모든 서브키가 동일해야 한다. 만약 K, K*가 equivalent key라고 가정하면

$K_1=K^*_1$, $K_9=K^*_9$, (A, B, C, D)=(B_9, A_9, D_9, C_9)이므로 다음 식을 만족한다.

$$A + C = A^* + C^*, \qquad B - D = B^* - D^*.$$
$$B + D = B^* + D^*, \qquad A - C = A^* - C^*.$$

위 식을 A′=(A-A^*), B′=(B-B^*), C′=(C-C^*), D′=(D-D^*)라 놓고 풀면 다음과 같다.

$$A' + C' = 0, \qquad A' - C' = 0.$$
$$B' + D' = 0, \qquad B' - D' = 0.$$

위 식의 해는 A′=B′=C′=D′=0이므로(즉, K=K*이므로) 서로 다른 equivalent key pair는 존재하지 않는다. 참고로 위 식에서 사용된 연산이 modulo 2^{32}에 대한 연산이므로 A′=C′=2^{31}인 경우의 해도 나올 수 있다. 그러나 2라운드와 10라운드의 식을 살펴보면 역시 A′과 마찬가지로 MSB(A_2')을 제외한 A_2'의 모든 비트는 0이 된다. 그러나 $A_2'=(b_0'a_3'a_2'a_1')$, $A'=(a_3'a_2'a_1'a_0')$이므로(즉, a_3'=0이어야 하므로) A′=2^{31}이 될 수 없다. 결과적으로 두 개의 다른 키를 가지고 모든 라운드에 대해 동일한 라운드 키를 생성할 수 없으며, Equivalent key는 존재하지 않는다.

6. Related Key Attack

Related-key Cryptanalysis는 key scheduling으로 구한 related-key를 사용하여 암호를 공격하는 방법이다. 이 공격은 대개 키 생성 알고리즘이 비교적 단순하거나 규칙적인 경우에 서브키 간의 관계를 이용하여 키를 찾는 것이다. 즉, 이 공격은 두 개의 키인 K 와 K′ 간의 일정한 관계

(K′=f(K))를 찾음으로써 가능하고 암호의 라운드 수와 무관하게 적용될 수 있다. Related-key Cryptanalysis는 DC 등의 다른 암호해독법과 같이 적용되기도 하며 related-key slide attack(rotational subkey related-key attack)과 differential related-key attack 등이 있다.

가. related-key slide attack

SEED 수정안과 같은 구조에 대한 related-key slide attack은 임의의 평문(P)에 대한 1라운드부터 n라운드까지의 암호화 결과가 다른 평문(P^*)에 대한 s라운드부터 s+n라운드까지의 암호화 결과와 동일할 때 가능하다. Biham 등은 LOKI89, LOKI91, Lucifer 등에 이 공격을 적용하였으며, DES의 key scheduling에서 shift량이 동일하도록 변형한 DES에도 적용 가능함을 보였다. 주로 key scheduling이 rotation, bitwise permutation, XOR 등과 같이 단순한 연산만을 사용할 때 related-key slide attack이 가능하다.

SEED 수정안의 경우, 2라운드마다 동일한 규칙으로 B||A, D||C를 한 번씩 rotation하여 update한 D, C, B, A로서 서브키를 생성하므로 rotational related-key가 되려면 키 K에 대한 i+2 라운드 서브키가 K^*에 대한 i 라운드 서브키와 같아야 한다. 실제로 SEED 수정안은 매 라운드마다 일정한 rotation으로 A, B, C, D를 G 함수의 입력으로 서브키를 생성하지만 라운드 키 상수가 모두 다르고 A, B, C, D의 조합이 모두 다르기 때문에 related-key가 발생할 확률이 거의 없다. 만약 SEED 수정안의 키생성 알고리즘에서 related-key slide attack에 대한 related-key가 존재한다면, related-key K, K^*에 대해서, $K^*_i = K_{i+2}$ (i=1, ..., 14)이어야 한다. 그러면, $K^*_{1,0} = K_{3,0}$, $K^*_{9,1} = K_{11,1}$ 이므로 두 식을 더하면 다음과 같다.

$A^*+B^*+C^*-D^* = A_3+B_3+C_3-D_3.$ (1)

또한 두 식 $K^*_{1,1} = K_{3,1}$, $K^*_{9,0} = K_{11,0}$ 을 더하면,

$A^*_9+B^*_9+C^*_9-D^*_9 = A_{11}+B_{11}+C_{11}-D_{11}.$ (2)

한편, (A, B, C, D)=(B_9, A_9, D_9, C_9), (A_3, B_3, C_3, D_3)=(B_{11}, A_{11}, D_{11}, C_{11})이므로,

$A^*+B^*-C^*+D^* = A_3+B_3-C_3+D_3.$ (2′)

그러면 식 (1)과 식 (2′)에 의해,

$A^*+B^* = A_3+B_3, \quad C^*-D^* = C_3-D_3.$ (3)

식 (3)에 의해, 일반성을 잃지 않고, 다음과 같이 가정할 수 있다.

$A^* = A_3 + s, \quad B^* = B_3 - s, \quad C^* = C_3 + t, \quad D^* = D_3 + t.$

그런데, $K^*_{1,0} = K_{3,0}$, $K^*_{9,0} = K_{11,0}$ 이므로, s 와 t는 다음을 만족하여야 한다.

$s + t = KC_1 - KC_3, \quad s - t = KC_{11} - KC_9.$

즉,

$s = (KC_1 - KC_3 - KC_9 + KC_{11})/2$

또는 $(KC_1 - KC_3 - KC_9 + KC_{11})/2 + 2^{31}$,

$t = (KC_1 - KC_3 + KC_9 - KC_{11})/2$

또는 $(KC_1 - KC_3 + KC_9 - KC_{11})/2 + 2^{31}$.

그러나, (KC_1 - KC_3 - KC_9 + KC_{11}), (KC_1 - KC_3 + KC_9 - KC_{11})가 모두 홀수이므로 위 두 식을 모두 만족하는 s, t를 찾을 수 없다. 즉, SEED 수정안의 related-key를 찾을 수 없다. 그러므로 SEED 수정안에 대한 related-key slide attack이 불가능하다. 만약 모든 라운드 키 상수가 동일하다면 related-key를 찾을 수 있다. 가령, s = t = 0 인 경우 (A,B,C,D)=(0xiiii, 0xjjjj, 0xkkkk, 0xhhhh), (i,j,h,k : byte hexacode)가 related key(또한 equivalent key임)가 된다. 그러므로 SEED 수정안의 key scheduling에서 각 라운드마다 값이 다르게 정의된 라운드 키 상수에 의해 각 라운드의 서브키 생성방법간의 규칙성이 깨어지므로 related-key slide attack이 불가능하다.

나. Differential related-key attack

Differential related-key attack은 주어진 master key간의 차이와 평문 쌍의 차이에 대한 differential characteristic을 구성함으로써 related-key slide attack에 강하도록 설계된 블록암호를 공격하거나 DC공격의 성공확률을 높이고자 하는 공격방법이다. 주로 IDEA 등과 같이 linear factor가 존재하는 암호나 SAFER-64K와 같이 master key의 각 비트가 모든 서브키에 고루 영향을 미치지 못하고 특정 서브키에만 영향을 미칠 때 적용 가능하다. SEED 수정안의 경우, master key의 각 비트가 모든 라운드의 서브키에 고루 영향을 미치며 32비트 연산인 덧셈과 뺄셈을 사용하였으므로 각 서브키의 생성에 대한 master key의 비트 조합이 모두 다르다고 할 수 있다. 또한 diffusion 효과가 큰 bitwise permutation의 사용하므로 효과적인 differential related-key attack이 어렵다.

먼저, SEED 수정안의 differential related-key attack에 대한 공격의 복잡도를 살펴보기 위해 key scheduling과 F함수에 쓰인 S-box와 G함수의 XOR 연산에 대한 입출력 difference의 특성을 살펴본다.

S-box의 상위 두 비트에 대한 XOR difference의 분포는 다음과 같다.

S_1:	0x40 →0x40, 0x80	$p=2^{-7}$
	0xc0	p=0
	0x80 → 0x80	$p=2^{-7}$
	0x40, 0xc0	p=0
	0xc0 → 0x40, 0x80, 0xc0	$p=2^{-7}$
S_2:	0x40 →0x80	$p=2^{-7}$
	0x40,0xc0	p=0
	0x80 →0x80, 0xc0	$p=2^{-7}$
	0x40	p=0
	0xc0 → 0x40, 0x80, 0xc0	p=0

여기서 S_2가 0xc0을 입력차로 가지는 경우 원하는 패턴인 0x40, 0x80, 0xc0이 나오지 않으므로, 계산의 편의상 상위 한 비트에 대한 difference만 고려하여 분석하기로 한다. 그러면 두 가지 S-box에 의해 입력차 0x80은 출력차 0x80에 확률 2^{-7}의 확률로 대응된다.

G 함수에 대한 XOR difference의 분포는 다음과 같다. active S-box의 개수를 줄이기 위해 bitwise permutation의 diffusion 효과가 최소가 되도록 출력차를 정했다.

0x80000000 → 0x80808000, $p=2^{-7}$

0x00800000 → 0x80800080, $p=2^{-7}$

0x00008000 → 0x80008080, $p=2^{-7}$

0x00000080 → 0x00808080, $p=2^{-7}$

0x80800000 → 0x00008080, $p=2^{-14}$

0x80008000 → 0x00800080, $p=2^{-14}$

0x80000080 → 0x80000080, $p=2^{-14}$

0x00808000 → 0x00808000, $p=2^{-14}$

0x00800080 → 0x80008000, $p=2^{-14}$

0x00008080 → 0x80800000, $p=2^{-14}$

0x80808000 → 0x80000000, $p=2^{-21}$

0x80800080 → 0x00800000, $p=2^{-21}$

0x80008080 → 0x00008000, $p=2^{-21}$

0x00808080 → 0x00000080, $p=2^{-21}$

0x80808080 → 0x80808080, $p=2^{-28}$

다음에 XOR difference들의 덧셈에 대한 확률은 대략적으로 다음과 같이 쓸 수 있다.

(0x00000000, 0x80000000) → 0x80000000 : 1

(0x00000000, 0x00800000) → 0x00800000 : 1/2

(0x00000000, 0x00008000) → 0x00008000 : 1/2

(0x00000000, 0x00000080) → 0x00000080 : 1/2

(0x00000000, 0x80800000) → 0x80800000 : 1/2

(0x00000000, 0x80008000) → 0x80008000 : 1/2

(0x80000000, 0x80000000) → 0x00000000 : 1

(0x80000000, 0x00800000) → 0x80800000 : 1/2

(0x80000000, 0x00008000) → 0x80008000 : 1/2

(0x80000000, 0x00000080) → 0x80000080 : 1/2

...

(0x80008000, 0x80800000) → 0x00808000 : 1/4

(0x00808080, 0x80808080) → 0x80000000 : 1/8

(0x80808080, 0x80808080) → 0x00000000 : 1/8

위에서 나열한 G에 대한 XOR difference와 덧셈에 대한 XOR difference를 이용하여 key scheduling과 F함수의 characteristic을 계산할 수 있다. 먼저, 임의의 키 K에 대해, K*를 'K의 하위 8비트와 64비트만을 toggle한 키'로 정의하고 K'을 K와 K^*의 XOR difference라고 하자. 즉, K^*는 K의 a_1과 c_0(표 3-20 참조)의 최상위 비트를 각각 toggle한 값이 된다.

주어진 K와 K^*에 대해 각 서브키를 구하는 과정에서 G의 입력차/출력차와 그 때의 확률을 각각 구해보면 (표 3-21)와 같다.

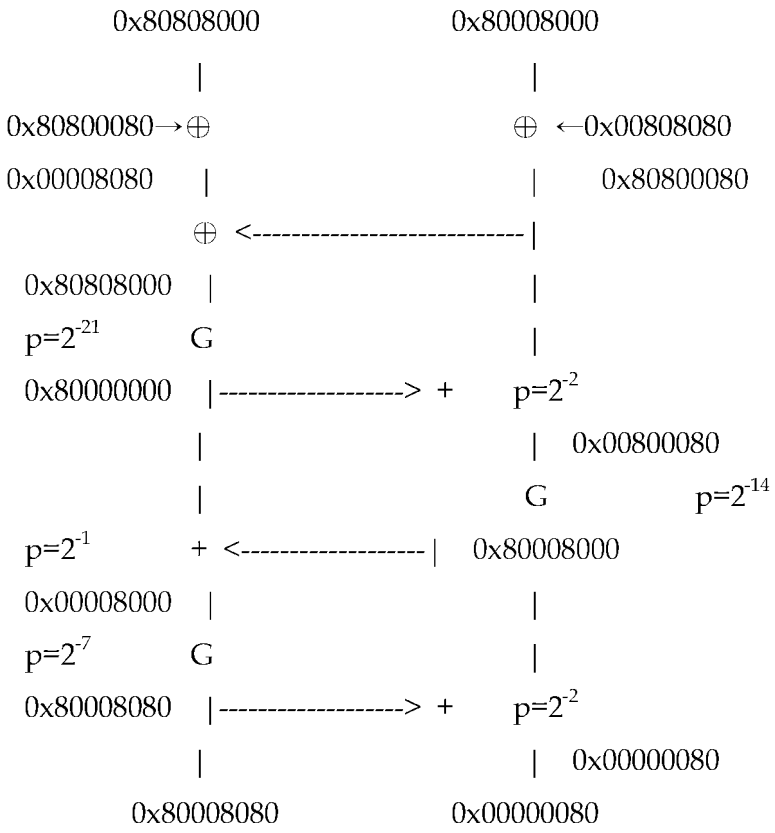
	$K_{i,0}$의 차			$K_{i,1}$의 차		
i	G 입력차(p_1)	G 출력차(p_2)	$p=p_1p_2$	G 입력차(p_1)	G 출력차(p_2)	$p=p_1p_2$
1	0 0 80 80 (1/4)	80 80 0 0 (2^{-14})	2^{-16}	0 (1)	0 (1)	1
2	0 (1/2)	0 (1)	1/2	0 (1)	0 (1)	1
3	0 0 80 80 (1/4)	80 80 0 0 (2^{-14})	2^{-16}	0 (1)	0 (1)	1
4	0 0 80 0 (1/2)	80 0 80 80 (2^{-7})	2^{-8}	80 0 0 0 (1)	80 80 80 0 (2^{-7})	2^{-7}
5	0 80 0 0 (1/2)	80 80 0 80 (2^{-7})	2^{-8}	80 0 0 0 (1)	80 80 80 0 (2^{-7})	2^{-7}
6	0 80 0 0 (1/2)	80 80 0 80 (2^{-7})	2^{-8}	0 80 0 0 (1/2)	80 80 0 80 (2^{-7})	2^{-8}
7	80 0 0 0 (1)	80 80 80 0 (2^{-7})	2^{-7}	0 80 0 0 (1/2)	80 80 0 80 (2^{-7})	2^{-8}
8	80 0 0 0 (1)	80 80 80 0 (2^{-7})	2^{-7}	0 0 80 0 (1/2)	80 0 80 80 (2^{-7})	2^{-8}
9	0 (1)	0 (1)	1	0 0 80 80 (1/4)	80 80 0 0 (2^{-14})	2^{-16}
10	0 (1)	0 (1)	1	0 (1/2)	0 (1)	1/2

(표 3-21) $K'=(2^8 + 2^{64})$인 경우 각 서브키의 입출력차 표

Key scheduling에서 서브키의 difference를 구하는 방법으로 K'_1을 구하는 과정을 살펴본다. a1과 c0(표 3-20 참조)의 최상위 비트만이 바뀌었으므로 $K'_{1,0}$에서 G의 입력차는 하위 8비트와 16비트에서 carry가 발생하지 않았다고 가정할 때(즉, 확률 $p_1=1/4$) (A+C-KC_1)의 difference는 0x00008080이 된다. G에 의해 0x00008080은 확률 $p_2=2^{-14}$로서 0x80800000이 된다(두개의 active S-box와 bitwise permutation의 성질에 의해). 한편 $K'_{1,1}$에서 대한 G의 입력차는 0이므로 출력차도 0이 된다. 그러므로 $K'_{1,1}=0$(확률=1)이 된다.

또한 F함수의 입력으로 들어가는 평문과 서브키가 둘 다 바뀔 때 평문 입력차 = 0x80808000 || 0x80008000 와 서브키 입력차 =

0x80800080 || 0x00808080 에 대해 F의 differential을 구하는 과정을 다음과 같이 나타낼 수 있다. 이 때 differential의 성립확률은 $p=2^{-47}$이 된다.



이제 위에서 구한 두 가지 키인 K와 K^*의 difference K′에 대해 differential characteristic을 계산해 보자. Feistel 구조의 특성을 효과적으로 이용하기 위해 평문차를 (K'_2, K'_1)로 선택한다. 그러면 1라운드와 2라운드에서는 F함수의 평문입력과 키 입력이 상쇄되어 모든 G함수의 입력차가 0이므로 F함수의 출력차는 0이다. 또한 $K'_3=K'_1$이므로 3라운드에서도 1, 2 라운드와 같은 동일한 효과를 가진다. 또한 1, 2, 3 라운드로 이루어진 characteristic의 확률은 주어진 키 difference에 대

해 K'_1, K'_2, K'_3가 발생할 확률만으로 결정된다. 평문간의 차가 (K'_2, K'_1)이고 두 개의 키의 차가 앞서 정의한 K′일 때, 5라운드까지의 characteristic을 순서대로 표현하면 다음과 같다.

평문 차 : (K′2, K′1)

	F의 입력차	F의 출력차	확률
1라운드	(K'_1, K'_1)	0	2^{-16}
2라운드	(K'_2, K'_2)	0	1/2
3라운드	(K'_1, K'_3)	0	2^{-16}
4라운드	(K'_2, K'_4)	0x8080800080	$2^{-16-49}= 2^{-64}$
5라운드	($K'_1 \oplus$8080800080, K'_5)	0x8000808000008000	$2^{-15-34}= 2^{-49}$

4라운드 characteristic의 확률=2^{-97}, 5라운드 characteristic의 확률=2^{-146}이 되므로 5라운드 이상이 되면 related-key differential attack이 거의 불가능하다고 볼 수 있다. 위 특성은 $K'_3=K'_1$이므로 3 라운드가 성립할 확률이 다소 높지만 4 라운드와 5 라운드에서 알 수 있듯이 G 함수의 diffusion효과가 매우 크므로 평문과 키의 특정 비트만을 변화시킨다 해도 확률이 충분히 큰 characteristic을 찾기가 매우 어렵다. $K'_3=K'_1$를 만족하도록 하기 위해 키 K에 대해 K^*을 K의 b_0과 c_3의 최상위비트를 toggle한 키로 설정하거나 K의 a_3와 c_2의 최상위비트를 toggle하도록 해서 각 characteristic의 확률을 살펴보았으나 위의 characteristic보다 확률이 훨씬 큰 characteristic을 찾지는 못했다.

K의 b_0과 c_3의 최상위비트를 toggle했을 때 특성확률 :
(2^{-15}, 1, 2^{-15}, 2^{-16-50}, 2^{-16-34}),
K의 a_3와 c_2의 최상위비트를 toggle했을 때 특성확률 :

$(2^{-15}, 1, 2^{-15}, 2^{-16-65}, 2^{-16-?})$.

또한 K'_3, K'_1이 다르므로 3라운드의 확률은 낮지만 bitwise permutation에 의한 diffusion이 적어지도록 K와 K^*이 한 비트(carry의 효과를 최소화하기 위해 각 byte의 최상위비트만 고려함)만이 다르도록 하여 그 중 몇 가지의 특성확률을 살펴보았다.

K의 a_1의 최상위비트만을 toggle했을 때 특성확률 :
$(2^{-8}, 2^{-8}, 2^{-72}, 2^{-42})$,
K의 a_3의 최상위비트만을 toggle했을 때 특성확률 :
$(2^{-7}, 2^{-8}, 2^{-70}, 2^{-57})$,
K의 c_3의 최상위비트만을 toggle했을 때 특성확률 :
$(2^{-7}, 2^{-7}, 2^{-55}, 2^{-70})$.

그러나 이 경우에도 역시 G함수가 0이 아닌 입력차를 가지는 3 라운드부터 확률이 현저히 낮아짐을 알 수 있다. 그러므로 $K'_3=K'_1$이 되도록 K'을 선택하는 경우의 특성이 상대적으로 더 효과적임을 알 수 있다.

이상으로 SEED 수정안이 5라운드 이상이 되면 differential related-key attack은 거의 불가능함을 보였다. carry의 효과와 0x80을 포함한 다른 입력차/출력차를 같이 고려한 확률, 그리고 bitwise permutation의 diffusion 등을 종합적으로 고려한다면 다소 높은 확률을 가지는 특성을 찾을 수는 있을 것이지만 일단 찾는다고 하더라도 모든 경우를 고려한 종합적인 분석은 매우 어려울 것이며 새로 변형된 bitwise permutation의 diffusion 특성상 제시된 특성보다 현저히 높은 확률을 가지는 특성을 구하는 것도 역시 어려울 것이다.

제 4 장 수정된 SEED의 효율성 분석

아래의 결과는 MSVC/C++ 5.0으로 구현하여 Pentium Pro 200MHz에서 수행한 결과이다. 속도를 cycle 수로 나타내면 같은 architecture하에서 clock에 비례하는 속도를 쉽게 구할 수 있으므로 편리한 경우가 많다. 수정된 SEED의 속도는 대략 DES와 같다. 그러나 key scheduling은 상당히 빠른 편으로 MAC이나 hash function의 building block으로 이용시 상당히 효율적일 것이다.

알고리즘명	라운드 키 생성	암복호화
DES	1606 cycles = 8.03 μsec	432 cycles/ 8 bytes = 28.97 Mbps
3DES	4704 cycles = 23.52 μsec	1133 cycles/ 8 bytes = 9.38 Mbps
RC5	1594 cycles = 7.97 μsec	140 cycles/ 8 bytes = 89.40 Mbps
SAFER	8150 cycles = 40.75 μsec	571 cycles/ 8 bytes = 21.90 Mbps
Blowfish	142770 cycles = 713.85 μsec	262 cycles/ 8 bytes = 47.66 Mbps
CAST	1124 cycles = 5.62 μsec	349 cycles/ 8 bytes = 35.82 Mbps
IDEA	13266 cycles = 66.33 μsec	516 cycles/ 8 bytes = 24.25 Mbps
SEED	411 cycles = 2.06 μsec	870 cycles/16 bytes = 28.00 Mbps

(표 4-1) 블록 암호 알고리듬들의 성능 비교

- RC5는 32bit word, 16 round 버전임.
- 라운드 키 생성은 랜덤한 비밀키에 대해 라운드 키 생성 과정을 반복적으로 수행.
- 암복호화는 랜덤한 단위블록(8bytes, 단 SEED는 16bytes)을 반복적(암호화한 결과를 다시 암호화)으로 암호화하는 과정과 반복적으로 복호화하는 과정을 수행. 암호화와 복호화의 수행속도는 거의 비슷하고, 위의 결과는 이를 평균한 것임.

- 각 알고리듬에 대하여 라운드 키 생성과 암호화, 복호화 과정을 약 10초간 수행하여 그 속도를 측정하였고 암복호화는 암호화와 복호화 속도의 평균을 취하였다. 그리고 이 결과를 수행기종 (Pentium Pro 200MHz)의 속도를 고려하여 cycles로 환산하였다.

제 5 장 결론

블록 암호알고리즘은 전자 데이터의 기밀성 기능을 제공하기 위한 핵심 기술이다. SEED는 안전성과 효율성을 고려하여 국내 전자상거래에서 활용가능한 암호알고리즘을 개발할 목적으로 개발되었고, 향후 국내 표준으로 제안할 예정이다.

본 블록 암호알고리즘 SEED는 초안에 약간의 수정을 통해 효율성 저하없이 안전성을 강화하여 128비트의 안전도를 충분히 지원하는 것으로 분석되었다.

개발된 SEED를 통하여 현재 정보사회 진입의 커다란 이슈로 부각되고 있는 정보보호 서비스를 제공함으로써 전자상거래 등에서 유통되는 개인정보 및 거래 정보의 안전성이 확보될 것으로 기대된다.