SEED 블록 암호 알고리즘의 파이프라인 하드웨어 설계

(A Pipelined Design of the Block Cipher Algorithm SEED)

엄 성 용 † 이 규 원 †† 박 선 화 †††

(Seong Yong Ohm) (Kyu-Won Lee) (Sun-Hwa Park)

요 약 최근 들어, 정보 보호의 필요성이 높아지면서, 암호화 및 복호화에 관한 관심이 커지고 있다. 특히, 대용량 정보의 실시간 고속 전송에 사용되기 위해서는 매우 빠른 암호화 및 복호화 기법이 요구되었다. 이를 위한 방안중의 하나로서 기존의 암호화 알고리즘을 하드웨어 회로로 구현하는 연구가 진행되어 왔다. 하지만, 기존 연구의 경우, 구현되는 회로 크기를 최소화하기 위해, 암호화 알고리즘들의 주요 특성인 병렬 수행 가능성을 무시한 채, 동일 회로를 여러 번 반복 수행시키는 방법으로 설계하였다.

이에 본 논문에서는 1998년 한국정보보호센터에서 개발한 국내 표준 암호화 알고리즘 SEED의 병렬 특성을 충분히 활용하는 새로운 회로 설계 방법을 제안한다. 이 방법에서는 암호 연산부의 획기적인 속도 개선을 위해 암호 블록의 16 라운드 각각을 하나의 단계로 하는 16 단계의 파이프라인 방식으로 회로를 구성한다. 설계된 회로 정보는 VHDL로 작성되었으며, VHDL 기능 시뮬레이션 검증 결과, 정확하게 동작함을 확인하였다. 또한 FPGA용 회로 합성 도구를 이용하여, 회로 구현시 필요한 회로 크기에 대한 검증을 실시한 결과, 하나의 FPGA 칩 안에 구현 가능함을 확인하였다. 이는 단일 FPGA 칩에 내장될 수 있는 고속, 고성능의 암호화 회로 구현이 가능함을 의미한다.

키워드 : SEED, 암호화 알고리즘, 하드웨어 설계, 파이프라인, 하드웨어 합성

Abstract The need for information security increases interests on cipher algorithms recently. Especially, a large volume of data transmission over high-band communication network requires faster encryption and decryption techniques for real-time processing. It would be a good solution for this problem that we implement the cipher algorithm in forms of hardware circuits. Though some previous researches use this approach, they focus only on repeatedly executing the core part of the algorithm to minimize the hardware chip size, while most cipher algorithms are inherently parallel.

In this paper, we propose a new design for the SEED block cipher algorithm developed by KISA (Korea Information Security Agency) in 1998 as Korean standard cipher algorithm. It exploits the parallelism of the algorithm basically and implements it in a pipelined fashion. We described the design in VHDL program and performed functional simulations on the program, and then found that it worked correctly. In addition, we synthesized it and verified that it could be implemented in a single FPGA chip, implying that the new design can be practically used for the actual hardware implementation of a high-speed and high-performance cipher system.

Key words : Block Cipher Algorithm, Hardware Design, Pipelined Design, SEED, Hardware Synthesis

· 본 연구는 2001학년도 서울여자대학교 자연과학연구소 학술연구비로 수행되었음.

† 종신회원 : 서울여자대학교 정보통신공학부 교수
osy@swu.ac.kr
†† 비 회 원 : 이랜드시스템 3G 연구소 연구원
leekw@eland.co.kr
††† 비 회 원 : 서울여자대학교 정보통신공학부
bban@swu.ac.kr

논문접수 : 2002년 1월 29일
심사완료 : 2002년 11월 29일

1. 서 론

일반적인 암호 시스템의 설계시 따르는 제약 조건은 크게 두 가지로 분류될 수 있다. 첫째는 암호화 및 복호화 알고리즘이 계산적으로 효율적이어야 한다는 것이다. 암호화와 복호화는 각각 데이타의 송신 및 수신 시점에서 이루어지기 때문에 이 작업에 과다한 계산이 요구된다면 통신 시스템의 효율성은 급격히 하락할 것이다. 둘

째는 암호시스템의 안정성은 키의 기밀성에만 의존해야 한다. 실용적인 암호 시스템이 되기 위해서는 암호화 및 복호화 알고리즘을 공개하여 누구나 사용할 수 있게 해야 한다. 따라서 암호화와 복호화에 적용되는 키가 암호 시스템의 안정성을 유지하는 유일한 수단이 되기 때문에 키에 대한 안전한 관리가 필수적이다. 하지만, 키의 길이가 매우 길어져야 한다는 제약 조건이 따른다[1]. 이는 특정 암호문을 만드는 데에 사용된 키를 찾아내기 위해서 가능한 모든 키를 시도해 보는 암호 공격을 방지하기 위해서이다.

대량의 정보를 고속으로 송수신하는 경우 정보의 실시간 암호화 및 복호화 구현이 필요하다. 복잡한 암호 알고리즘을 소프트웨어로 구현한다면 CPU의 부하로 전체 시스템에 영향을 미칠 것이고, 암호화 및 복호화의 수행 속도가 저하될 수 있다. 이러한 속도 문제를 개선하기 위해, 기존의 암호화 알고리즘을 하드웨어 회로로 구현하는 연구가 진행되어 왔다. 하지만, 기존 연구의 경우, 구현되는 회로 크기의 제한으로 인해, 암호화 알고리즘들의 주요 특성인 병렬 수행 가능성을 무시한 채, 동일 회로를 여러 번 반복 수행시키는 방법으로 설계하였다[2,3,4,5,6,7].

이에 본 연구에서는 한국형 대표적 대칭 알고리즘인 SEED[3,5,6,7]를 대상으로 암호 연산부 전체를 파이프라인 구조로 설계하는 방안을 제안한다. 일반적으로 파이프라인으로 하드웨어를 설계했을 때 면적이 증가되는 단점을 가진다. 반면, 하나의 프로세서를 서로 다른 기능을 가진 여러 개의 서브 프로세서로 나누어 각 서브프로세서가 동시에 서로 다른 데이타를 취급하도록 하면, 그 연산 속도가 월등히 빨라진다는 장점을 가진다[9]. SEED의 구조는 암호부의 연산이 16 라운드(round)로 구성되는 점을 착안하여, 각 라운드를 하나의 독립적인 단계(stage)로 하는 파이프라인 방식의 설계를 제안한다.

하지만, 이러한 파이프라인식 설계는 근본적으로 기존 방식에 비해 많은 회로 면적을 요구하게 된다. 본 연구에서는 이러한 면적 증가는 근본적으로 피할 수는 없더라도 최대한 최소화시켜 궁극적으로 단일 FPGA 칩 내에 구현 가능하도록 시도하였다. 제안된 설계는 VHDL[11]로 기술되었으며, 기능 시뮬레이션 및 상위단계 합성 결과, 기능적으로 정확하며, 또한 단일 FPGA 칩 내에 구현 가능함을 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서는 여러 가지 암호화 알고리즘과 이를 하드웨어로 구현하는 기존의 관련 연구에 대해 설명하며, 3장에서는 한국 표준 암호화 알고리즘인 SEED의 구조에 대해 설명한다. 4장에서는 암호화 알고리즘을 파이프라인 방식으로 설계하여 내용을 VHDL로 기술한 내용을 각 모듈별로 설명하였다. 마지막으로 5장에서는 설계에 대한 정확성 검사를 위해 수행한 기능 시뮬레이션 결과 및 단일 칩 내에 구성 가능한 지를 판단하기 위해 수행한 상위 단계 합성 결과를 보여준다.

2. 관련연구

기존의 암호 알고리즘들은 암호화가 적용되는 평문의 길이에 따라 스트림 암호(stream cipher)와 블록 암호(block cipher) 방식으로 분류될 수 있다. 암호화가 적용되는 평문의 길이가 한 개의 비트나 문자이면 스트림 암호 방식이고, 한 개 이상일 경우는 블록 암호 방식이 된다[11,12]. 스트림 암호 방식에서는 평문 비트들을 키 수열 비트들과 단순히 Exclusive-OR 연산시킨 암호문이 생성되기 때문에 스트림 암호 시스템의 설계는 결국 키 수열을 생성해내는 키 생성(key generation) 알고리즘의 설계라고 볼 수 있다. 블록 암호 방식에서는 한 개 이상의 비트 집합이 암호화의 단위가 된다. DES, SEED, RSA와 같은 암호 알고리즘은 이 범주에 속한다.

암호화 알고리즘의 또 다른 분류는 암호화에 사용되는 키와 복호화에 사용되는 키가 서로 동일하냐 또는 상이하냐에 따라서 이루어진다[13,14]. 암호화와 복호화에 사용되는 키가 동일한 암호 알고리즘을 관용 암호시스템(conventional cryptosystem) 또는 대칭형 암호시스템(symmetric cryptosystem)이라고 한다. 대표적인 대칭형 암호 알고리즘인 DES와 국산 대표 알고리즘인 SEED가 이 범주에 속한다. 반면 암호화와 복호화에 적용되는 암호가 서로 다른 알고리즘은 공개키 암호시스템(public key cryptosystem) 또는 비대칭형 암호시스템(asymmetric cryptosystem)으로 불린다. 암호화에 사용되는 공개키(public key)는 메시지를 암호화하고자 하는 송신자가 소지하게 되며, 복호화에 사용되는 개인키(private key)는 수신자만이 소유하게 된다[15,16].

각종 암호화 알고리즘을 하드웨어로 구현한 기존의 연구를 살펴보면, 구현한 알고리즘의 종류에 따라 DES를 구현한 연구[2]와 RSA를 구현한 연구[17], 그리고 SEED를 설계 구현한 연구[3,5,6,7]로 나눌 수 있는데, 특히 블록 암호 알고리즘인 DES와 SEED를 하드웨어로 구현한 기존 연구의 대부분에서는 하나의 라운드를 회로로 구성하고 이를 여러 번 반복 연산하는 방법을 사용하였다. 병렬 Feistel 구조를 가지는 DES를 제안한 연구[2]는 DES 자체의 구조적 문제(error의 propaga-

ton) 때문에 pipeline 방식을 사용할 수 없어 데이타 처리속도와 보안사이에서의 trade-off 관계를 가질 수밖에 없었던 DES의 성능을 향상시켰다. RSA를 구현한 연구[17]에서는 Montgomery 기법을 이용한 systolic array 방식의 설계를 제안하였다. SEED를 설계 구현한 연구[3,5,6,7]는 SEED가 구조상 많은 하드웨어 자원을 필요로 한다는 점을 감안하여 자원 제한에 의한 문제점을 최소화하기 위해 F 함수부와 라운드키 생성부에서 사용되는 G 함수를 각각 1개씩 구현하고 이를 순차적으로 사용하도록 구현하였다.

하지만, 이러한 연구들은 하드웨어 구현시 게이트 수를 최소화하는데 목적을 두고 있기 때문에 알고리즘이 갖는 고유의 병렬성을 충분히 활용하지 못하는 면이 있다. 반면에 본 논문에서는 단일 칩 구현이라는 면적 제한 조건을 만족하는 가운데 알고리즘의 병렬성을 최대한 살린 새로운 설계 방법을 제안한다.

3. SEED 알고리즘

3.1 전체 구조

SEED는 안정성과 효율성 측면을 고려한 암호화 알고리즘으로, 128 비트 안전도를 유지하기 위해 기존에 알려진 공격들에 대해 강하고 효율적으로 설계하였다. SEED 알고리즘의 데이타 처리 단위는 8, 16, 32비트 모두 가능하다. 입출력문의 크기는 128비트이며, 입력키의 크기 역시 128비트를 기준으로 설계되었다. 내부는 16라운드로 구성된 Feistel 구조를 가진다.

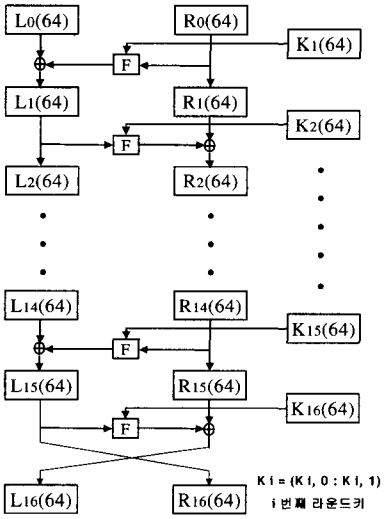

그림 1 SEED 알고리즘의 전체 구성도

그림 1은 SEED 알고리즘의 전체 구성도이다. 그림에서 보는 바와 같이, 128비트의 입력을 2개의 64비트 블록(L0(64), R0(64))으로 나누어, 16개의 라운드키(64비트)와 함께 16 라운드를 순차적으로 수행한 후, 최종적으로 128비트의 출력(L16(64), R16(64))을 낸다.

3.2 Feistel 구조의 암호 및 복호화 기능

SEED는 16개의 라운드를 가진 Feistel 구조를 갖는다. 그림 2는 SEED의 Feistel 구조를 보여준다. Feistel 구조란 각각 t 비트인 L0, R0 블록으로 이루어진 2*t 비트의 평문 블록 (L0, R0)을 r라운드(r≥1)를 거쳐 암호문 (Lr, Rr)을 내는 반복 구조로서, 평문 블록에 대해 몇 번의 라운드를 거쳐 암호화를 수행하는 것을 말하며, 보통 Feistel 구조는 3 라운드 이상의 짝수 라운드로 구성된다.

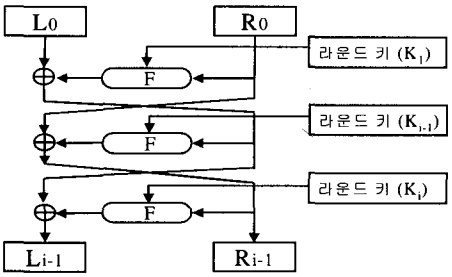

그림 2 Feistel 구조

3.3 F 함수

SEED의 F 함수는 그림 3에 나타난 바와 같이 수정된 64 비트의 Feistel 구조를 갖추고 있다.

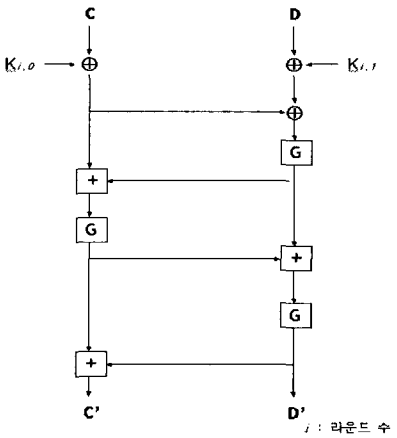

그림 3 F 함수의 구조

F 함수는 Feistel 구조의 블록 암호화 알고리즘의 특성을 구분하는 가장 큰 요소로서, 32 비트 단위의 2개의 블록(C, D)을 입력으로 받아, 32 비트 단위의 2개의 블록(C′, D′)을 각각 출력한다. 즉, 암호화 과정에서 64 비트의 데이타(C, D)와 64 비트의 라운드키 Ki (=(Ki,0 ; Ki,1))를 F 함수의 입력으로 받아 64 비트의 함수 결과(C′, D′)를 출력한다. 그림에서 ⊕는 Exclusive-OR 연산을 의미하고 ⊞는 두 피연산자를 더하되 carry를 무시하는 연산, 즉, a⊞b = (a+b) mod 2^{32}을 의미한다.

3.4 G 함수

G 함수는 F함수 및 라운드키 생성시에 사용되는 주요 함수이다. 표 1은 전체 G 함수의 기능을 수식으로 설명한 것이며, 그림 4는 이러한 기능을 수행하는 G 함수의 전체 구조를 보여준다. 표와 그림에서 ⊕ 기호는 Exclusive-OR 연산을 의미하며, & 기호는 비트 단위 AND 연산을 의미한다. G 함수는 기존의 G 함수와 마찬가지로 4개의 확장된 SS-box들의 exclusive-OR로 구현할 수 있는데, 32 비트 입력을 4 부분으로 나눈 후 각 블록에 대해 S-box (S1, S2)를 통과시켜, 각각 m_0에서 m_3까지의 값들을 생성한 후, 이를 다시 4개의 SS-box들에 저장하는 방식으로 수행된다.

표 1 G 함수의 기능

$y_0 = S_1(x_0),\ y_1 = S_2(x_1),\ y_2 = S_1(x_2),\ y_3 = S_2(x_3),$
$z_0 = (y_0 \,\&\, m_0) \oplus (y_1 \,\&\, m_1) \oplus (y_2 \,\&\, m_2) \oplus (y_3 \,\&\, m_3),$
$z_1 = (y_0 \,\&\, m_1) \oplus (y_1 \,\&\, m_2) \oplus (y_2 \,\&\, m_3) \oplus (y_3 \,\&\, m_0),$
$z_2 = (y_0 \,\&\, m_2) \oplus (y_1 \,\&\, m_3) \oplus (y_2 \,\&\, m_0) \oplus (y_3 \,\&\, m_1),$
$z_3 = (y_0 \,\&\, m_3) \oplus (y_1 \,\&\, m_0) \oplus (y_2 \,\&\, m_1) \oplus (y_3 \,\&\, m_2),$
$(m_0 = 0xfc,\ m_1 = 0xf3,\ m_2 = 0xcf,\ m_3 = 0x3f)$

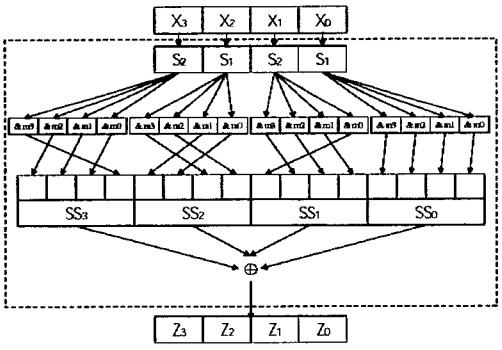

그림 4 G 함수의 구조

3.5 키 생성 알고리즘

SEED의 키 생성 기법에서는 다음에 설명한 바와 같이, 128 비트의 암호키를 64 비트씩 좌우로 나누어 이들을 교대로 8 비트씩 좌/우로 회전 이동 한 결과에 대해 간단한 산술 연산과 G 함수를 적용함으로써 라운드키를 생성한다. 각 라운드에 사용되는 라운드 키는 다음과 같은 방식으로 생성한다.

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

② $K_{1,0}$ = G(A+C-KC_0) ; $K_{1,1}$ = G(B-D+KC_0)
(단, KC_0 : 1 라운드 상수)로 제1라운드 키를 생성하고,

③ A||B = (A||B) >> 8

④ $K_{2,0}$ = G(A+C-KC_1) ; $K_{2,1}$ = G(B-D+KC_1)
(단, KC_1 : 2 라운드 상수)로 제2라운드 키를 생성하고,

⑤ C||D = (C||D) << 8

⑥ $K_{3,0}$ = G(A+C-KC_2) ; $K_{3,1}$ = G(B-D+KC_2)
(단, KC_2 : 3 라운드 상수)로 제3라운드 키를 생성하고,

⑦ 각 라운드에 대해 반복하여 해당 라운드의 키를 생성한다.

4. SEED 암호화 칩의 모듈별 설계

4.1 모듈 연결부의 설계

SEED 암호화 칩은 크게 키 생성부와 암호부로 나눌 수 있다. 그림 5는 SEED 회로의 전체 구조를 보여준다. 먼저 키 생성 블록은 주어진 128 비트의 암호키를 토대로 16개의 라운드키를 출력시킨다. 각 라운드키는 완전히 동시에 생성되지는 않지만, 키 생성 블록은 기본적으로 조합 회로이므로 약간의 지연 시간이 지나면, 모두 안정된 상태의 출력 값을 생성한다.

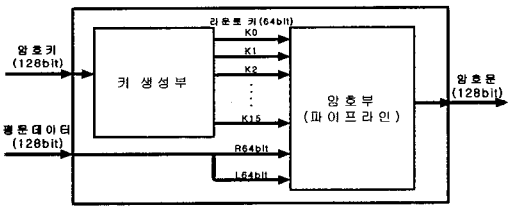

그림 5 전체 모듈

각 라운드별 키가 모두 생성되면, 파이프라인 방식으로 구현된 암호부는 주어진 암호키에 의해 생성된 16개

의 64 비트 라운드키들과 클록당 하나씩 입력되는 평문 데이타(128 비트)를 받아 들여, 일정 시간이 지난 후 한 클록당 하나의 암호문(128 비트)을 출력한다.

4.2 키 생성 블록의 설계

키 생성 블록은 128 비트의 암호키를 입력받아 16개의 64 비트 라운드키를 생성한다. 생성된 라운드키는 암호키가 변하지 않는 한, 일정한 값을 유지한다. 키 생성 블록은 기본적으로 클록이 없는 조합 회로로 구성된다. 따라서 모든 출력 값은 일정 지연 시간 후에는 모두 안정된 결과 값을 유지하는 특성이 있다.

한편 본 논문에서는 암호부만을 설명하였으나, 복호부 역시 키 생성부에서 설계되는 라운드키를 사용하여 생성된 라운드키의 순서를 역으로 사용하면 되므로, 자세한 설명은 생략한다. 키 생성부에서는 +, - 연산과 G 함수가 사용된다. 그림 6은 키 생성 블록에 대한 설계를 VHDL 코드로 작성한 예이다.

```
entity Key_Gen is
    port (              EncKey      : in std_logic_vector (127 downto 0);
                        Key_out0    : out std_logic_vector ( 63 downto 0);
                        Key_out1    : out std_logic_vector ( 63 downto 0);
                        Key_out2    : out std_logic_vector ( 63 downto 0);
                        Key_out3    : out std_logic_vector ( 63 downto 0);
                        Key_out4    : out std_logic_vector ( 63 downto 0);
                        Key_out5    : out std_logic_vector ( 63 downto 0);
                        Key_out6    : out std_logic_vector ( 63 downto 0);
                        Key_out7    : out std_logic_vector ( 63 downto 0);
                        Key_out8    : out std_logic_vector ( 63 downto 0);
                        Key_out9    : out std_logic_vector ( 63 downto 0);
                        Key_out10   : out std_logic_vector ( 63 downto 0);
                        Key_out11   : out std_logic_vector ( 63 downto 0);
                        Key_out12   : out std_logic_vector ( 63 downto 0);
                        Key_out13   : out std_logic_vector ( 63 downto 0);
                        Key_out14   : out std_logic_vector ( 63 downto 0);
                        Key_out15   : out std_logic_vector ( 63 downto 0));
end Key_Gen;

architecture main of Key_Gen is
-------------------------------------------------------
component G_Fun
    port (
                        G_in        : in std_logic_vector(31 downto 0);
                        G_out       : out std_logic_vector(31 downto 0)
          );
end component ;
-------------------------------------------------------
begin
  ROUND0:
        AB0(63 downto 0) <= EncKey(127 downto 64);
        CD0(63 downto 0) <= EncKey(63 downto 0);
                G00 <= AB0(63 downto 32) + CD0(63 downto 32) - "10011110001101110111100110111001";
                                --G(A+C-KC0=0x9e3779b9)
        G01 <= AB0(31 downto 0) - CD0(31 downto 0) + "10011110001101110111100110111001";
                                --G(B-D+KC0=0x9e3779b9)
        G00_block : G_Fun port map(G00, key_out0(31 downto 0)) ;
        G01_block : G_Fun port map(G01, key_out0(63 downto 32)) ;
        AB1 <= AB0(7 downto 0) & AB0(63 downto 8) ;   -- BA 8 bit right shift
        CD1 <= CD0 ;
  ROUND1:
        G10 <= AB1(63 downto 32) + CD1(63 downto 32) - "00111100011011101111001101110011";
                --0x3c6ef373
        G11 <= AB1(31 downto 0) - CD1(31 downto 0) + "00111100011011101111001101110011";
                --0x3c6ef373
        G10_block : G_Fun port map(G10, key_out1(31 downto 0)) ;
        G11_block : G_Fun port map(G11, key_out1(63 downto 32)) ;
        CD2 <= CD1(55 downto 0) & CD1(63 downto 56) ; -- DC 8 bit left shift
        AB2 <= AB1 ;
   .....

  ROUND15:
        G150 <= AB15(63 downto 32) + CD15(63 downto 32) -"10111100110111001100111100011011";
                --0xbcdccf1b
        G151 <= AB15(31 downto 0) - CD15(31 downto 0) + "10111100110111001100111100011011";
                --0xbcdccf1b
        G150_block : G_Fun port map(G150, key_out15(31 downto 0)) ;
        G151_block : G_Fun port map(G151, key_out15(63 downto 32)) ;
end main;
```

그림 6 키 생성 블록에 대한 VHDL 코드

4.3 암호부의 파이프라인 설계

각 라운드를 통과한 결과는 클록이 주어지면, 다음 라운드로 보내진다. 각 라운드를 거친 결과가 마지막 라운드를 통과하게 되면 최종적인 암호화 결과가 나오게 된다. 본 논문에서는 각 라운드를 각 단계(stage)로 하는 파이프라인 구조를 사용한다. 그림 7은 암호부의 파이프라인 설계를 VHDL 코드로 표현한 예이며, 그림 8은 그림 7의 파이프라인 설계에서 파이프라인 각 단계에 해당하는 각 라운드에 대한 VHDL 코드를 보여준다.

암호부에는 16개의 라운드 블록(Round0, …, Round15)이 있고, 각각의 라운드 블록은 F 함수부(F0_Fun … F15_Fun)를 사용하도록 설계하였다. 각각의 라운드 블록의 수행 후 다음 라운드 블록으로 진행하기 위해 데이타 저장을 위한 레지스터(Register0 … Register15)를 둔다. 이렇게 암호부를 파이프라인 방식으로 설계함에 있어 고려할 사항은 16개의 라운드키가 동시에 공급되어 유지되고 있어야 한다는 점이다. 파이프라인 단계간의 데이타의 흐름은 상위 64비트와 하위 64비트로 나뉘어서 동작하며, 각 파이프라인 단계 사이에는 레지스터를 두어 값을 전달한다. 최종 라운드(라운드 15)의 경우에는 상위 64비트와 하위 64비트의 데이타가 교환되어 출력된다.

```
entity SEED_Pipeline is
    port(       clk          : in std_logic;
                reset        : in std_logic;
                enable       : in std_logic;
                Round_key0 : in std_logic_vector(63 downto 0) ;--Round Keys below
                Round_key1 : in std_logic_vector(63 downto 0) ;
                Round_key2 : in std_logic_vector(63 downto 0) ;
                Round_key3 : in std_logic_vector(63 downto 0) ;
                Round_key4 : in std_logic_vector(63 downto 0) ;
                Round_key5 : in std_logic_vector(63 downto 0) ;
                Round_key6 : in std_logic_vector(63 downto 0) ;
                Round_key7 : in std_logic_vector(63 downto 0) ;
                Round_key8 : in std_logic_vector(63 downto 0) ;
                Round_key9 : in std_logic_vector(63 downto 0) ;
                Round_key10: in std_logic_vector(63 downto 0) ;
                Round_key11: in std_logic_vector(63 downto 0) ;
                Round_key12: in std_logic_vector(63 downto 0) ;
                Round_key13: in std_logic_vector(63 downto 0) ;
                Round_key14: in std_logic_vector(63 downto 0) ;
                Round_key15: in std_logic_vector(63 downto 0) ;
                Data_in      : in std_logic_vector(127 downto 0); --Plain Text
                Data_out     : out std_logic_vector(127 downto 0)--Ciphered Text);
end SEED_Pipeline;

architecture struct of SEED_Pipeline is
  .....
  begin
    Round_0: Round port map  (
                clk, reset, enable, Round_key0,
                Round_in => Data_in, Round_out => Data0_out);
    Round_1: Round port map  (
                clk, reset, enable, Round_key1,
                Round_in => Data0_out, Round_out => Data1_out);
    .....

    Round_15: Round port map  (
                clk, reset, enable, Round_key15,
                Round_in => Data14_out, Round_out => Data15_out);
    Data_out(127 downto 64) <= Data15_out(63 downto 0);
    Data_out(63 downto 0) <= Data15_out(127 downto 64);
  end struct;
```

그림 7 파이프라인 블록의 VHDL 코드

```
entity Round is
    port (      clk         : in std_logic;
                Reset       : in std_logic;
                enable      : in std_logic;
                Round_key   : in std_logic_vector ( 63 downto 0 );  round key
                Round_in    : in std_logic_vector ( 127 downto 0 );  plain text
                Round_out   : out std_logic_vector ( 127 downto 0)-- cipher text);
end Round;

architecture struct of Round is
    component f_fun
        port (
                Round_key   : in std_logic_vector ( 63 downto 0);
                F_in        : in std_logic_vector ( 63 downto 0);
                F_out       : out std_logic_vector ( 63 downto 0));
    end component;

    component reg
        port(   clk         : in std_logic;
                Reset       : in std_logic;
                enable      : in std_logic;
                reg_in      : in std_logic_vector(127 downto 0) ;-- Register In
                reg_out     : out std_logic_vector(127 downto 0)  Register Out);
    end component;
.....
begin
    ........
    F0 : f_fun port map (
                Round_key, F_in => Round_in(63 downto 0), F_out => F_out_s);

    reg_temp (63 downto 0) <= Round_in(127 downto 64) or F_out_s;
                --intermidiation of exclusive or

    reg_temp (127 downto 64) <= Round_in(63 downto 0);

    Reg_Storage : reg port map (
                clk, reset, enable, reg_in  => reg_temp, reg_out => Round_out);
end struct;
```

그림 8 라운드 블록의 VHDL 코드

4.4 파이프라인의 설계방법

기존 논문에서 제시된 암호부의 설계중 대표적인 방법은 하나의 라운드 블록을 설계하고 그 블록을 16회 반복 연산하는 방법이었다.[8] C 암호 알고리즘에서 사용되는 For Loop처럼, 암호부를 16회 반복 루프를 돌면서 암호화시키는 것이다. 이 방법으로 설계된 암호 모듈의 연산시간은 "한 라운드의 암호 연산시간(R)×16(회)×입력문의 크기(I) = 16RI" 가 소요된다. 예를 들어, 입력문의 크기가 26일 경우 라운드 반복설계 방법으로 소요되는 시간은 "16R×26 = 416R" 이다.

표 2는 암호부의 설계를 라운드 반복 구조로 설계 시 연산 속도를 나타낸 것이다. 입력마다 16R의 시간이 소요되므로 A에서 Z까지의 입력에 대한 암호연산은 416R의 시간이 소요됨을 예상할 수 있다.

본 논문에서 제시하는 암호부의 설계 방법은 16개의 라운드 블록을 동시에 사용할 수 있는 파이프라인 방법으로 설계하였다. 이 방법으로 설계된 암호 모듈의 연산시간은 "(한 라운드의 연산시간(R)×16(회))+(한 라운드의 연산시간(R)×입력문의 크기(I-1)) = 16R+(I-1)R = R(15+I)"의 시간이 소요된다.

예를 들어, 입력문의 크기가 26일 경우 파이프라인 설계로 소요되는 시간은 R× (15+26) = 41R 이다. 표 3은 암호부를 파이프라인 구조로 설계 시 연산속도를 나타낸 것이다. A에서 Z의 입력이 있을 경우 첫 입력의 결과는 표 2와 같이 16R이 소요되지만, 두 번째 B에서 Z의 입력은 A의 결과가 발생된 후 바로 R의 시간이 소

표 2 라운드 반복 구조의 연산속도

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
R	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

표 3 파이프라인 구조의 연산속도

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
R0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
R1		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
R2			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
R3				A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
R4					A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
R5						A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
R6							A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
R7								A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
R8									A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
R9										A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
R10											A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R11												A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
R12													A	B	C	D	E	F	G	H	I	J	K	L	M	N
R13														A	B	C	D	E	F	G	H	I	J	K	L	M
R14															A	B	C	D	E	F	G	H	I	J	K	L
R15																A	B	C	D	E	F	G	H	I	J	K

요된 후 결과가 발생된다. 그러므로 16R+25R = 41R의 시간이 소요됨을 예상할 수 있다.

4.5 파이프라인 방식과 16 라운드 공유 방식 비교분석

그림 9는 블록 암호화 알고리즘을 연산하는 방법에 따라 동일한 기능을 하는 라운드의 반복 연산을 공유하는 방법을 채택했을 때와 파이프라인 기능을 사용했을 경우의 총 소요되는 게이트 수를 나타낸다.

공유 방식을 사용했을 때 게이트 수는 라운드 수와 무관하며, 파이프라인 기능을 사용할 경우 사용하는 라운드 수에 따라 요구되는 게이트 수가 선형적으로 증가됨을 알 수 있다.

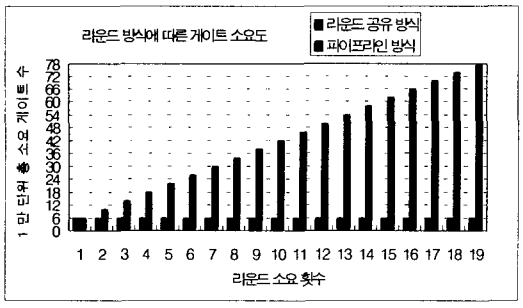

그림 9 라운드 방식에 따른 게이트 소요도

라운드 공유 방식은 16라운드 후에 결과가 출력되는데 반해, 파이프라인을 사용한 경우 16라운드까지는 라운드 공유방식과 동일하나 16라운드 이후에는 매 클럭마다 결과가 출력되고 있다.

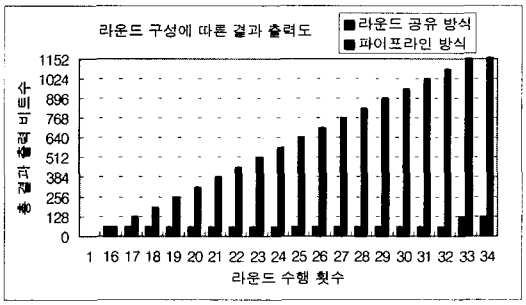

그림 10 라운드 구성에 따른 결과 출력도

파이프라인 방식으로 설계하면 라운드 수행 횟수와 관계없이 총 결과 비트 수는 선형적으로 증가하게 된다.

암호부의 Feistel 구조로 이뤄진 16라운드를 16단계 파이프라인 라운드 블록으로 구현하면, 데이타 처리속도가 향상되지만 그 라운드 수만큼의 게이트 수가 선형적으로 증가된다.

개선책으로 N라운드 혼합방식이 있는데, 이는 N개의 라운드를 하나의 블록으로 처리하는 것이다. 이는 라운드 공유 방식에 비해 속도와 면적의 trade off 관계에 있어서 합의점을 찾아 줄 수 있을 것이라 예상된다.

5. 기능 시뮬레이션 및 상위단계 합성

5.1 각 세부 모듈별 기능 시뮬레이션

설계의 정확성을 검증하기 위해, 먼저 각 세부모듈별로 독립된 기능 시뮬레이션을 수행하였다. 사용된 시뮬레이션 도구는 Cadence의 NCVHDL이다. 시뮬레이션 분석 결과, 각 세부 모듈은 독립적으로 작동할 경우, 정확하게 동작함을 확인할 수 있었다. 그림 11은 키 생성 블록에 대한 모듈별 기능 시뮬레이션 결과의 한 예를 보여 주며, 그림 12는 파이프라인 방식으로 동작하는 암호부에 대한 기능 시뮬레이션 결과의 한 예를 보여준다.

키 생성 블록은 조합 회로로 구성되어 있기 때문에 그림 12에서 결과 값인 16개의 라운드키 값들 Key_out0, ..., Key_out15는 동일 클록에서 생성됨을 확인할 수 있다. 분석 결과 생성된 키 값들은 정확하였다. 한편, 파이프라인 방식으로 구현된 암호부는 클록 주기가 20ns이고 enable 신호가 10ns에 주어진다고 가정할 경우, 40ns에 첫 라운드(round0)의 결과 값을 출력한 후,

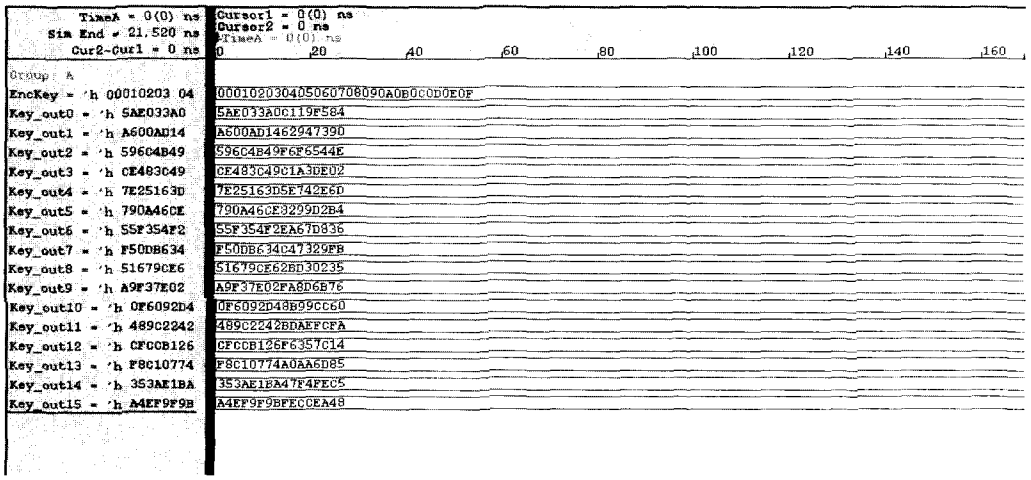

그림 11 키 생성 블록에 대한 기능 시뮬레이션 결과 예

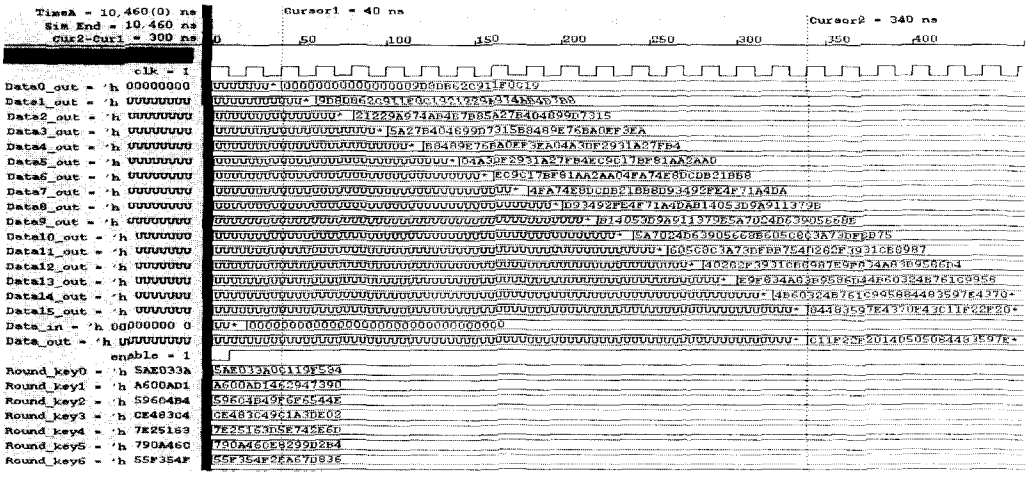

그림 12 파이프라인 블록에 대한 기능 시뮬레이션 결과 예

매 20ns마다 다음 라운드 결과 값을 차례로 출력하여, 결과적으로 340ns(=40ns+20ns×15) 후에는 최종 라운드(round15)인 15라운드 결과 값을 출력함을 알 수 있다. 또한 이러한 각 라운드별 결과 값들은 클록에 따라 다음 값들을 가졌고, 출력된 라운드별 결과 값 및 최종 출력 값은 모두 정확함을 확인 할 수 있었다.

5.2 전체 모듈(모듈 연결부)에 대한 기능 시뮬레이션

전체 회로를 구성하는 주요 세부 모듈에 대한 독립적인 기능 시뮬레이션을 통해 각 세부 모듈의 동작이 정확함을 확인한 후, 이들 모듈을 연결한 전체 시스템에 대한 기능 시뮬레이션을 수행하였다. 그림 13은 전체 모듈에 대한 기능 시뮬레이션 결과 예를 보여준다. 즉, 암호키가 "00000000000000000000000000000000"이고 데이타 입력 값이 "000102030405060708090A0B0C0D0E0F"인 경우, 암호화된 최종 결과가 "C11F201405050844835 97E4370F43"인 지를 검증하는 것이다. 분석 결과, 340ns 후에 올바른 암호화 결과 값이 출력됨을 확인할 수 있다.

5.3 합성 결과 및 분석

이러한 기능적인 정확성에도 불구하고, 앞에서 언급한 바와 같이 본 결과가 하나의 FPGA 칩 안에 구현할 수 없다면, 또 다른 문제를 낳게 된다. 따라서 본 논문에서는 상위 단계 합성 도구인 Synplify를 이용하여, 본 설계가 어느 정도의 게이트 수를 필요로 하는 지에 대한 분석을 시도하였다. 표 4는 Synplify에서 Virtex를 타겟 디바이스로 하여 합성한 결과를 보여준다. 실험 분석 결과, Xilinx의 Virtex-II Family의 XC2V400 이상의 칩이나 Virtex-E Family의 XCV2600E 이상의 칩을 사용할 경우, 모든 SEED 알고리즘이 단일 칩 내에 구현 가능함을 확인하였다.

표 4 합성 결과의 예

	셀종류	종류별 개수
합성된 셀의 종류 및 개수 (입출력 기본 소자 포함)	MUXCY_L	3,443
	XORCY	3,555
	MUXF5	22,307
	MUXF6	9,905
	VCC	1
	GND	1
	FDE	2,048
	BUF	5,118
	IBUF	257
	OBUF	128
	BUFGP	1
레지스터 비트수	2,048 비트	
필요한 전체 LUT의 갯수	54,803 개	

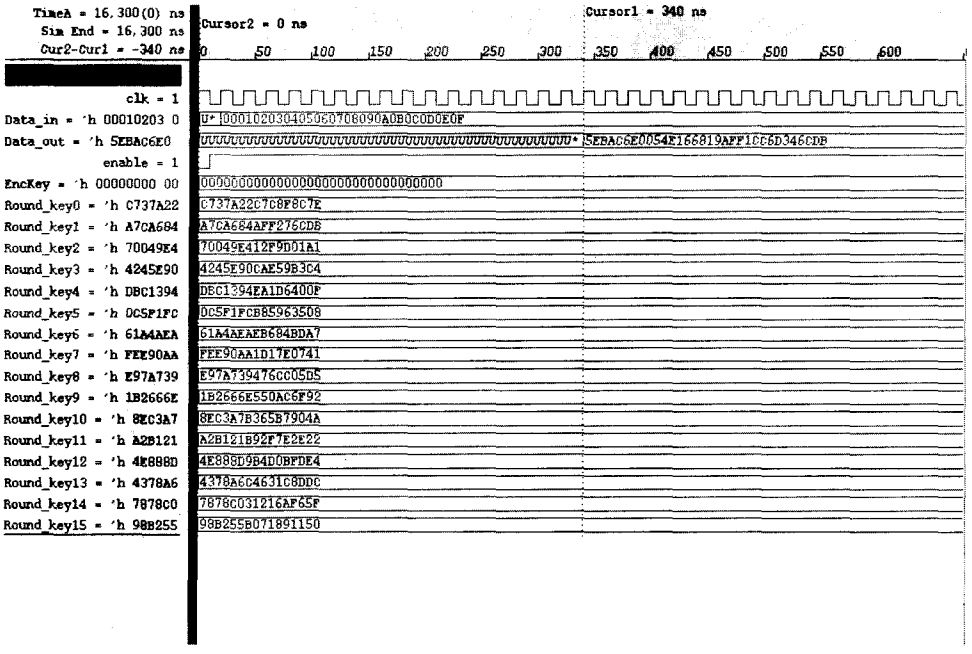

그림 13 전체 모듈에 대한 시뮬레이션 결과

6. 결 론

암호화 알고리즘을 하드웨어 칩으로 구현하는 방법은 암호화 성능 향상을 위한 좋은 대안이다. 그러나 기존의 연구에서는 암호화 알고리즘을 하드웨어로 구현함에 있어, 알고리즘내에 존재하는 병렬적인 요소를 모두 병렬 구현하기보다는 일부만을 하드웨어로 구현한 후 이를 하드웨어적으로 반복 수행하는 방법을 채택하였다. 이는 소프트웨어적인 수행 속도보다는 빠르면서도 하드웨어 자원 사용을 최소화하기 위해서라고 볼 수 있다. 하지만, 대용량의 정보를 고속으로 암호화하여 실시간 전송하려는 요구를 만족하기 위해서는 암호화 알고리즘의 병렬성을 최대한 활용하는 것이 필요하게 되었다. 단, 이러한 병렬화 시도는 하드웨어 자원 사용량을 늘려 단일 칩에 회로를 구현하는 것이 불가능할 수도 있는 문제점을 내포하게 된다.

본 연구에서는 한국형 암호화 알고리즘인 SEED의 병렬성을 파이프라인 방식으로 구현하는 방법을 제안하였다. SEED는 16개의 독립적인 라운드로 구성되어 있어, 이를 각 파이프라인 단계로 하는 설계 방식은 아주 자연스러운 면이 있었다. 기능 시뮬레이션 결과, 예상된 바와 같이 정확하고 좋은 성능을 확인하였다. 한편, 본 연구에서는 이러한 파이프라인 설계가 단일 칩에 구현 가능한 지가 매우 중요한 관건이었는데, 상위단계 합성 결과, 상용 FPGA칩 내에 구현 가능함을 확인하였다.

향후, 실제 암호 회로를 구현하는데 필요한 회로의 게이트 수와 동작 속도간의 상관 관계를 보다 정밀히 조사하여, 사용자가 원하는 다양한 제한 조건을 만족하는 회로를 설계하기 위한 연구가 필요하다.

참 고 문 헌

[1] 박창섭, 암호이론과 보안, p.20-23, 대영사, 1999.

[2] 이선근, "DES의 데이타 처리속도 향상을 위한 변형된 Feistel 구조에 관한 연구", 전자공학회논문지, 제37권, 제12호, pp. 91-97, 2000.

[3] 김종현, "블록 암호 알고리즘 SEED의 면적 효율성을 고려한 FPGA 구현", 정보과학회논문지, 제7권, 제4호, pp. 372-381, 2001.

[4] 염동복, "블록 암호화 프로세서 및 인터페이스 설계 및 구현", 석사학위논문, 한국 항공대학교, 2000.

[5] 송문빈, 고명관, 정연모, "SEED 암호화 알고리즘의 하드웨어 구현", 한국정보처리학회 추계학술발표논문집, 제7권, 제2호, 경희대학교, pp. 1453-1456, 2000.

[6] Young-Ho Seo, Jong-Hyeon Kim, Yong-Jin Jung, and Dong-Wook Kim, "Area Efficient Implementation of 128-bit Block Cipher, SEED", ITC-CSCC 2000, KwangWoon Univ, Korea, 2000.

[7] 신종호, 강준우, "SEED 블록 암호 알고리즘의 단일 칩 연구", 대한전자공학회 하계종합학술대회 논문집, 제23권, 제1호, 한국외국어대학교 전자제어공, pp. 165-168, 2000.

[8] 정진욱, 최병윤, "SEED와 TDES 암호 알고리즘을 구현하는 암호 프로세서의 VLSI 설계", 대한전자공학회 하계 종합 학술대회 논문집, 제23권, 제1호, 동의대학교, 2000. 6 pp.166-172.

[9] Mano, M. Morris, Computer System Architecture, p.260, Prentice Hall(Sd), 1996.

[10] 박현철, VHDL 회로설계와 응용, 한성 출판사, 1995.

[11] Bruce Schneuer, Applied Cryptography, Wiley, 1996.

[12] William Stalings, Network and Internetwork Security, Prentice Hall International Edition, 1995.

[13] <http://fn2.freenet.edmonton.ab.ca/~jsavard/crypto/co0409.htm>

[14] <http://msdn.microsoft.com/library/psdk/security/secglos_62nt.htm>

[15] <http://www.counterpane.com>

[16] <http://www.cryptogate.com>

[17] 반도체교육센터, 암호화 칩 설계, pp. 97-136, 2000.

엄 성 용

1985년 서울대학교 컴퓨터공학과 졸업(학사). 1987년 서울대학교 대학원 컴퓨터공학과 졸업(석사). 1992년 서울대학교 대학원 컴퓨터공학과 졸업(박사). 1992년 ~ 1993년 컴퓨터신기술공동연구소 특별연구원. 1993년 ~ 1995년 University of California, Irvine에서 Post-Doc. 1996년 ~ 현재 서울여자대학교 정보통신공학부 부교수. 관심분야는 컴퓨터그래픽스, CAD 소프트웨어, 홈네트워킹, IEEE1394, etc

이 규 원

1999년 서울여자대학교 컴퓨터학과 졸업(학사). 2001년 서울여자대학교 컴퓨터학과 졸업(석사). 2001년 1월 ~ 현재 이랜드시스템 3G 연구소 연구원. 관심분야는 암호화 알고리즘, PKI

박 선 화

1998년 서울여자대학교 컴퓨터학과 졸업(학사). 2000년 서울여자대학교 컴퓨터학과 졸업(석사). 2000년 ~ 2001년 서울여자대학교 컴퓨터학과 인턴연구원. 2001년 ~ 현재 서울여자대학교 정보통신공학부 박사과정. 관심분야는 CAD 소프트웨어, 모바일 컴퓨팅, 임베디드 시스템