하드웨어 암호화 기법의 설계 및 성능분석
(Design and Performance Evaluation of Hardware Cryptography Method)

아 재 용 † 고 영 웅 † 홍 철 호 † 유 혁 ††
(Jae-Yong Ah) (Young-Woong Ko) (Cheol-Ho Hong) (Hyuck Yoo)

요 약 암호화는 송수신자 사이에 메시지 전달이 비밀스럽게 이루어 질 수 있도록 보장해주는 기법이다. 이러한 암호화 알고리즘은 높은 계산량을 필요로 하며, 결과적으로 프로세서 자원을 과도하게 사용하는 문제를 가지고 있다. 이러한 문제점을 해결하기 위하여 암호화 알고리즘을 하드웨어 방식으로 구현함으로써 시스템의 부하를 줄여주는 기법이 제시되고 있다. 본 논문에서는 하드웨어 암호화 기법에 대한 설계 및 구현에 대해서 언급하고 있으며, 하드웨어 암호화 알고리즘과 소프트웨어 암호화 알고리즘에 대한 성능을 비교 분석하였다. 실험 결과에서, 계산 복잡도가 낮은 DES 알고리즘은 하드웨어 방식을 적용하여도 높은 입출력 오버헤드에 의해서 성능이 향상되지 않지만, 계산 복잡도가 높은 Triple DES는 하드웨어 방식을 적용하였을 때, 대략 2-4배 성능이 향상됨을 볼 수 있었다.

키워드 : 암호화 알고리즘, DES, Triple DES

Abstract Cryptography is the methods of making and using secret writing that is necessary to keep messages private between two parties. Cryptography is compute-intensive algorithm and needs cpu resource excessively. To solve these problems, there exists hardware approach that implements cryptographic algorithm with hardware chip. In this paper, we presents the design and implementation of cryptographic hardware and compares its performance with software cryptographic algorithms. The experimental result shows that the hardware approach causes high I/O overheads when it transmits data between cryptographic board and host cpu. Hence, low complexity cryptographic algorithms such as DES does not improve the performance. But high complexity cryptographic algorithms such as Triple DES improve the performance with a high rate, roughly from two times to four times.

Key words : cryptographic algorithm, DES, Triple DES

1. 서 론

최근에는 인터넷을 통한 전자상거래가 활성화되면서, 네트워크를 통해 전송되는 데이타에 대한 보안이 요구되고 있다. 암호화 메커니즘(cryptography mechanism)[1,2,3]은 데이타를 기밀성 있게 보호하기 위하여 사용하는 메커니즘으로, 데이타를 전송하기 전에 암호화시키고, 수신한 후에 복호화 시키는 방식으로 사용된다. 따라서 전송된 데이타에 대하여 권한이 없는 사용자의 접근을 무효화시킴으로써 기밀성을 부여할 수 있다. 현재 잘 알려진 대부분의 암호화 메커니즘은 소프트웨어적인 방법을 사용하고 있으나, 암호화 메커니즘이 복잡해질수록 점차 많은 계산량을 요구하게 되어 시스템에 많은 부하를 주게 되었다. 따라서 이러한 문제점을 해결하기 위한 방법으로 암호화 기능을 하드웨어에 전담시킴으로써, 시스템의 부하를 줄이고자하는 암호화 방법[4,5,6,7,8,9,10,11,12,13,14]이 나타나게 되었다.

암호화 프로세서(cryptography processor)는 암호화 기능을 하드웨어적으로 구현한 연산 프로세서이며, 시스템 프로세서가 처리하던 암호화 기능을 하드웨어적으로 수행함으로써 시스템 프로세서의 부하를 줄이기 위해 고안된 장치를 가리킨다. 이러한 암호화 프로세서를 최적으로 사용하기 위해서는 시스템 내부 버스(internal

† 학생회원 : 고려대학교 컴퓨터학과
jyah@os.korea.ac.kr
yuko@os.korea.ac.kr
chhong@os.korea.ac.kr
†† 종신회원 : 고려대학교 컴퓨터학과 교수
hxy@os.korea.ac.kr
논문접수 : 2002년 1월 25일
심사완료 : 2002년 8월 6일

bus)에 장착시켜 처리량을 극대화하는 것이 효율적이겠지만, 그러한 시스템을 구성하는 것은 범용 시스템이 아닌 특성화된 시스템을 설계 및 구현해야 하므로 가격이 비싸게 된다. 따라서 대부분의 경우 암호화 프로세서를 외부 버스에 장착이 가능한 카드 형태로 제작하게 되고, 네트워크 카드나 비디오 카드를 제어하는 것과 비슷한 방식으로 제어하여 시스템에 암호화 기능을 제공해주게 된다. 본 논문에서는 이러한 암호화 보드를 PCI (peripheral component interconnect)[15] 버스에 장착할 수 있는 형태로 구현하여 성능을 측정하고 문제점을 분석해보고자 한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 암호화 프로세서에 하드웨어적으로 구현된 암호화 알고리즘들에 대하여 간단히 기술하였다. 3장에서는 암호화 프로세서를 장착한 PCI 보드의 하드웨어 구성에 대해서 기술하며, 4장에서는 하드웨어를 제어하기 위한 소프트웨어 구조에 대하여 언급한다. 5장에서는 실험을 통하여, 하드웨어적인 암호화 기법에 대한 성능을 비교 및 분석하였으며, 6장에서는 결론을 맺고 있다.

2. 암호화 알고리즘

암호화 알고리즘은 크게 대칭형(symmetric) 알고리즘과 비대칭형(asymmetric) 알고리즘으로 나뉜다. 대칭형 알고리즘은 평문을 암호화하는데 사용되는 암호화 키값과 암호화된 데이타를 다시 평문으로 복호화 하는데 사용되는 복호화 키값이 같은 경우 대칭형 알고리즘이라고 한다. 그리고 암호화 키값과 복호화 키값이 다른 경우를 비대칭형 알고리즘이라고 한다. 또한 고정된 크기의 입력을 통해서 고정된 크기의 암호화 데이타를 생산하는 알고리즘을 블록 알고리즘(block algorithm)이라 하고, 입력 크기에 제한이 없이 스트림(stream)형태로 데이타를 받아들여서 스트림 형태로 암호화된 데이타를 생산하는 알고리즘을 스트림 알고리즘(stream algorithm) [1]이라고 한다.

2.1 DES(Data Encryption Standard)

본 절에서 설명하고자 하는 DES 암호화 알고리즘은 대칭형 알고리즘이면서, 블록 알고리즘이라고 할 수 있다. DES는 64 bit의 평문을 64 bit의 암호문으로 만드는 블록 암호화 알고리즘으로 64 bit의 키값을 사용한다. 이 64 bit의 키 중 56 bit는 실제 키값이 되고 나머지 8 bit는 검사용 비트로 사용된다. DES의 암호화 과정을 개략적으로 살펴보면 다음과 같다.

1) 암호화에 사용할 키값 K0를 받아들이고, 암호화하고자 하는 평문을 64 bit의 블록 단위로 블록화한다.

2) 입력으로 들어온 64 bit의 블록을 초기 치환한다. 즉, DES에서 규정하고 있는 표에 따라서 입력으로 들어온 64 bit들의 bit 배열을 뒤섞어 놓는다.

3) 치환된 결과의 64 bit 중에서 상위 32 bit와 하위 32 bit를 각각 L0와 R0라 하고 입력으로 들어온 키값을 변형시킨 K1과 R0를 사용하여 F 알고리즘을 적용시킨 후 나온 결과 32 bit를 L0에 배타적 논리합(exclusive OR)을 수행하여 R1을 만든다. L1은 R0를 그대로 사용한다. 이런 식의 연산을 16번 수행시키는데, 그 각각을 하나의 라운드(round)라 하고 16라운드를 거쳐서 다시 64 bit 값을 만들어 낸다. 각 라운드마다 전 라운드에 사용된 키값을 변형시킨 새로운 키값이 F 알고리즘에 적용되어 새로운 R값을 만들어 내는데 사용되게 된다. F 알고리즘의 개략적인 설명은 다음과 같다.

F 알고리즘의 입력으로는 이전 라운드의 R값 32 bit와 이전 라운드에서 사용된 키값을 변형시킨 새로운 키값 K가 들어오게 된다. 입력으로 들어온 R값은 DES에서 정한 F 알고리즘의 확장표에 의해서 32 bit를 48 bit로 확장하게 된다. 즉, 32 bit중에서 16개의 bit를 중복하여 사용하여 48 bit로 확장한다. 이렇게 확장된 48 bit 값과 64 bit에서 변형되어 48 bit로 압축된 키값으로 논리합 연산을 수행하여 48 bit를 만들게 된다. 이렇게 만들어진 48 bit를 8개의 6 bit로 나누어서 DES에서 정한 8개의 S-box라는 표에 적용하여 각 S-box마다 4 bit씩을 만들어내게 된다. S-box는 입력으로 들어온 6 bit의 MSB(most significant bit)와 LSB(least significant bit)를 가지고 만든 2bit를 표의 행값으로 사용하고, 나머지 4 bit를 표의 열값으로 사용하여 S-box안에서 해당하는 위치의 값을 얻어오게 된다. S-box 표 안에 정의된 값들은 모두 16보다 작아서 이렇게 얻어온 값은 4 bit로 나타내는 게 가능하다. 이렇게 만들어진 32 bit를 F의 치환표에 의해서 다시 bit 배열을 뒤섞은 후 F 알고리즘의 출력값으로 내보내진다.

4) 이렇게 만들어진 64 bit 값을 DES에서 규정하고 있는 역 초기 치환 표에 의해서 치환하여 암호화된 64 bit를 만들어 내게 된다. 수행된 2) - 4)의 과정은 하나의 평문 블록을 암호화하는데 수행되는 과정을 나타내는 것이고, Feistel Network[2]이라 부른다. 전체적인 개념도는 그림 1과 같다.

5) 평문이 64 bit이상의 크기라면 2)에서 4)까지의 과정을 연속된 블록들에 적용하여 암호화된 블록들을 만들어 내게 된다.

6) 복호화의 경우 암호화된 문장과 키값을 받아들인 후, 암호화와 같은 과정을 수행하게 된다. 다만, 입력으

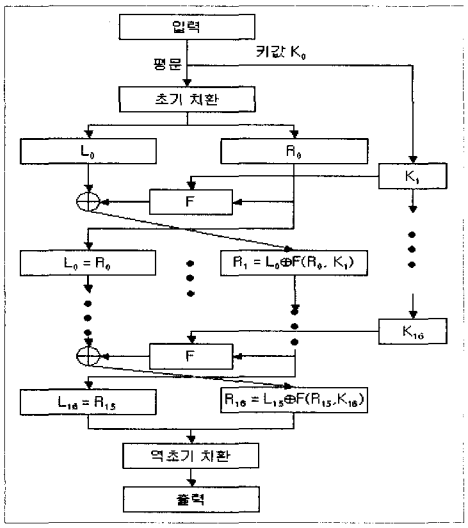


그림 1 DES의 암호화 과정

로 받아들인 키값을 내부에서 변형하여 사용하게 되는데, 변형된 키값들을 적용시키는 순서가 암호화의 경우와 역순일 뿐이다.

2.2 Triple DES

기본적으로 암호화 방법은 DES와 같다. 다만, DES를 3번 사용해서 암호화 과정에서의 복잡성을 증가시킨 것이므로 DES에 비해서 3배정도 느리고, 그만큼 복잡도도 증가하게 된다. Triple DES의 암호화 과정에 대한 개략도는 그림 2와 같다.

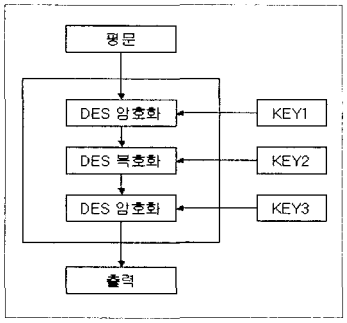


그림 2 Triple DES의 암호화 과정

3. 암호화 하드웨어의 구성

본 논문에서는 범용 시스템에서 사용가능한 암호화 보드를 대상으로 하고 있으며, 이를 위해서 암호화 프로세서를 시스템의 내부버스에 장착하는 방식은 고려하고 있지 않다. 왜냐하면 내부 버스에 장착하기 위해서는 시스템 보드를 새롭게 설계 및 구현해야 하는 문제점이 발생하기 때문이다. 따라서 암호화 프로세서를 장착한 PCI 보드 형태로 설계 및 구현하였으며, 암호화 PCI 보드는 범용 시스템에 PCI 확장 슬롯이 있는 경우에 장착하여 사용 가능하다. 본 연구에서 외부버스 규격으로 PCI를 선택한 이유는 특정 시스템에 국한되기 보다는 PCI가 널리 받아들여지고 있는 외부버스 규격이기 때문이다. 암호화 프로세서를 장착한 PCI 보드는 PCI 사양서 Rev 2.1을 기준으로 설계되었으며, PCI 버스에 대해서 Target 기능을 수행한다[4]. 암호화 PCI 보드에는 HI/FN, SCC, 및 SEED 칩이 집적되어 있으며, 보드의 버스 클럭은 33MHz, 신호선의 전압은 3.3V를 사용한다. 암호화 프로세서를 장착한 PCI 보드의 블록 다이어그램은 그림 3과 같다.

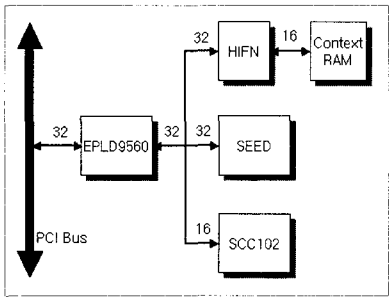


그림 3 암호화 PCI 보드의 블록 다이어그램

각 암호화 칩이 제공하는 기능을 살펴보면 다음과 같다.

1) HI/FN 칩은 데이타 압축, 데이타 암호화, 데이타 인증 알고리즘을 제공하고 있다. 구현한 알고리즘은 LZS, MPPC와 같은 압축 알고리즘이 있으며, DES, Triple DES, RC4와 같은 암호화 알고리즘이 있으며, SHA-1, MD5와 같은 인증 알고리즘이 있다. 내부적으로 RAM을 이용하여 키값과 같은 데이타를 RAM에 저장하여 암호화시에 같은 세션(session)을 이용하는 경우, RAM에 키값을 저장하거나 가져오게 하여 데이타의 입출력을 줄이고 있다.

2) SEED[17] 칩은 다양한 응용 시스템 분야에서 사용되는 데이타를 고속으로 암호화 및 복호화가 가능하도록 개발된 한국 표준 알고리즘을 구현한 칩으로 TTA(telecommunications technology association)의 규격을 만족하고 있다. 입출력 데이타 크기는 8 bit, 16 bit, 32 bit, 64 bit 및 128 bit 와 같이 다양한 블록 크기를 지원하고 있으며, 키 값은 128 bit로 구성되어 있다.

3) SCC[18] 칩은 1024 bit RSA 알고리즘을 구현한 칩으로, 일반적으로 사용되는 공개키 방식의 암호와 서명에 사용하기 위해서 구현되었다.

SCC 칩과 SEED 칩의 경우 PCI 버스의 클럭이 아닌 독자적인 클럭으로 움직이기 때문에 비동기적이며 HI/FN 칩의 경우는 PCI 클럭을 이용하기 때문에 동기적으로 동작한다

4. 암호화 보드 제어 소프트웨어의 설계 및 구현

4.1 솔라리스 디바이스 드라이버 개요

암호화 프로세서를 장착한 PCI 보드를 사용하기 위해서는 암호화 PCI 보드를 제어하는 소프트웨어가 필요하다. 이러한 제어 소프트웨어는 디바이스 드라이버의 형태로 구현되며, 암호화 기능을 필요로 하는 응용 프로그램에서는 직접적으로 암호화 PCI 보드를 제어할 필요없이 디바이스 드라이버에서 제공해주는 인터페이스를 이용하면 된다. 본 연구에서 대상으로 하는 시스템은 솔라리스 7 운영체제가 설치되어 있는 Ultra Sparc 10이며, 솔라리스 7 운영체제에서 암호화 PCI 보드를 제어하는 디바이스 드라이버의 설계 및 구현과정에 대해서 기술한다. 구현하는 디바이스 드라이버[19]는 문자형 디바이스 드라이버에 해당되며, 솔라리스에서 제공하고 있는 디바이스 드라이버 로드맵(roadmap)은 그림 4와 같다.

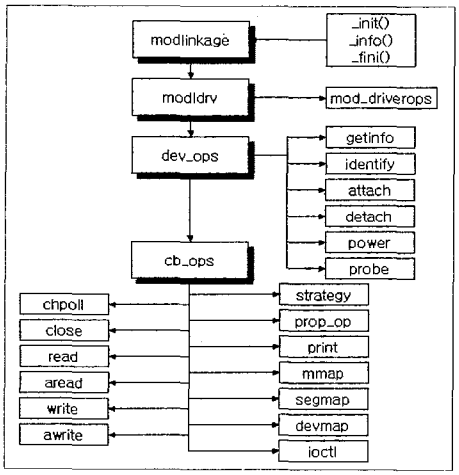


그림 4 디바이스 드라이버 로드맵

디바이스 드라이버를 구현하는데 있어서, modlinkage, modldrv, dev_ops 그리고 cb_ops에 해당되는 네 가지의 기본적인 구조체를 구현해야 하며, 각각에 대해서 개략적으로 살펴보면 다음과 같다.

1) modlinkage 구조체는 _init 함수에서 mod_install 함수를 호출하는데, 인자로 넘겨주어야 하는 구조체로 modldrv 구조체 변수를 포함하고 있어야 한다.

2) modldrv 구조체는 modlinkage 구조체의 멤버로 dev_ops와 cb_ops 구조체를 멤버로 포함하고 있어야 한다. modlinkage와 modldrv 구조체는 디바이스 드라이버를 시스템으로 적재하는데 사용되는 구조체라고 할 수 있다.

3) dev_ops 구조체는 attach, detach, getinfo 와 같은 함수 포인터를 멤버로 가지고 있으면서, 각각의 실제 구현 함수를 멤버로 가지고 있어야 하는 구조체이면서, cb_ops와 같이 실제 디바이스를 접근하는 함수들을 가지고 있는 구조체를 멤버로 포함해야 하는 구조체이다. 디바이스 드라이버가 시스템에 적재되어 초기설정을 하기 위한 작업들이 이 함수들에서 이루어져야 한다. 이 중에서 attach와 detach는 시스템에 디바이스 드라이버가 적재되면서 바로 불리는 함수로 attach에서의 주요한 작업은 디바이스의 레지스터를 커널 안의 주소공간에 맵핑하는 작업, 디바이스의 인터럽트를 시스템에 등록하는 등의 작업이 수행된다. detach에서는 attach에서의 수행되었던 작업들을 무효화시키기 위한 작업들이 수행되는데, 디바이스의 레지스터에 대해 맵핑이 되어있던 커널의 주소공간을 무효화 시키는 일이나, 시스템에 등록했던 인터럽트를 해제하는 등의 작업이 이루어진다.

4) cb_ops 구조체는 open, close, read, write 및 ioctl과 같이 실제로 디바이스를 접근하여 데이타를 읽거나 쓰는 함수들에 대한 포인터를 멤버로 가지고 있는 구조체로서 디바이스에 접근하는 기능들을 포함하고 있어야 한다. 이 중에서 open과 close 함수는 애플리케이션에서 디바이스를 호출하기 위해서 디바이스 노드를 열 때, 또는 닫을 때 불리는 함수로 open에서는 보통 디바이스를 초기화하는 작업과 필요한 메모리를 할당하는 등의 작업이 이루어지며, close에서는 사용한 메모리를 반환하는 등의 작업이 이루어진다. 또 read, write의 경우에는 애플리케이션에서 파일을 열고, 파일에 대해서 읽고 쓰는 것처럼 디바이스를 호출하는데 사용되는 함수로써, read는 디바이스로부터 데이타를 읽어서 애플리케이션으로 데이타를 복사하는 작업을 수행하며, write는 디바이스로 데이타를 전송하여 처리한 후 작업결과를 애플리케이션으로 전송하는 등의 작업을 수행한다. 마지막으로 ioctl은 디바이스마다 가지고 있는 특정한 기능을 제어하기 위해서 사용되는 함수로, ioctl 호출을 통해서 디바이스에게 필요한 데이타를 전송할 수도 있고, 필요한 데이타를 받아올 수도 있는 유연한 기능을 제공하는 함수이다. 본 논문에서 구현한 디바이스 드라이버를 호출

하는 작업은 모두 ioctl을 통해서 이루어진다.

솔라리스는 언급한바와 같이 디바이스 드라이버를 작성하기 위한 프레임워크를 제공해주는 것 뿐만 아니라, 솔라리스 DDI(device driver interface)/DKI(driver kernel interface)를 통해서 디바이스 드라이버와 솔라리스 커널의 나머지 부분을 인터페이스 하는데 있어서의 표준화와 안전성을 제공해주고 있다.

4.2 암호화 작업의 처리 흐름

디바이스 드라이버에서 암호화 작업이 처리되는 단계는 크게 6단계로 구분 지을 수 있다. 각 단계에 대한 설명은 다음과 같다.

4.2.1 디바이스 드라이버에 대한 open

디바이스 드라이버를 설치하고 나서, 시스템의 /dev 디렉터리 밑에 CryptoXpress라는 디바이스에 해당하는 디바이스 파일을 만들면 디바이스를 사용하기 위한 준비가 끝난다. 그 후, open 시스템 콜을 이용하여 /dev/CryptoXpress를 열게 되면, 구현한 디바이스 드라이버의 open 함수인 CryptoXpress_open이 호출된다. PCI 보드에 장착된 3 개의 칩에 대한 기본적인 초기화 과정을 수행하는 함수들을 호출한다. open 호출과정은 그림 5와 같다.

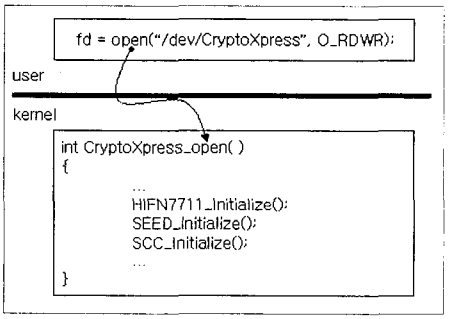


그림 5 open 호출

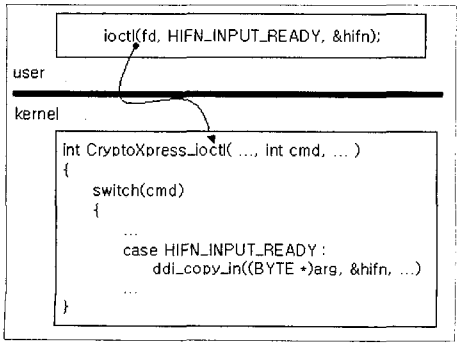


그림 6 ioctl을 통한 평문 크기와 키값의 전달

4.2.2 ioctl 함수를 통한 평문 크기와 키 값의 전달

ioctl 호출을 통하여, 미리 정의된 구조체를 이용해 암호화에 사용될 키값과 평문 길이, 세션 값 등을 디바이스 드라이버에 넘겨준다. 그림 6은 이러한 과정을 설명하고 있다.

4.2.3 ioctl을 통한 평문의 전달 및 암호화 함수 호출

ioctl 호출을 통해서 실제로 암호화에 사용될 평문 데이타를 디바이스 드라이버에게 넘겨준다. 그림 7은 이러한 과정을 설명하고 있다.

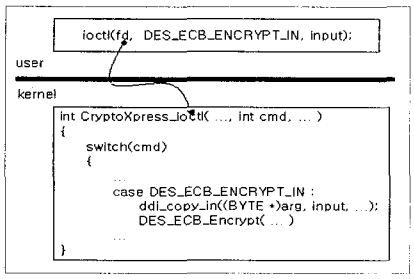


그림 7 ioctl을 통한 평문의 전달 및 암호화 함수 호출

4.2.4 암호화 함수에서의 명령어 구성 및 디바이스 호출

실제로 디바이스에 접근하는 함수로써, 디바이스가 적절한 암호화 작업을 수행할 수 있도록 명령어를 만들고, 그렇게 만들어진 명령어를 디바이스의 레지스터에 써서 디바이스가 암호화 하고자 하는 데이타의 입력을 기다리도록 만든다. 이제 실제로 데이타를 디바이스로 전송하게 되고, 데이타를 전송하면서 바로 디바이스에서 작업이 끝난 데이타에 대해서는 결과값을 가져갈 수 있도록 폴링(polling)을 하게 된다. 그림 8은 이와 같이 디바이스의 레지스터에 접근하는 함수를 부르는 과정을 보여주고 있다.

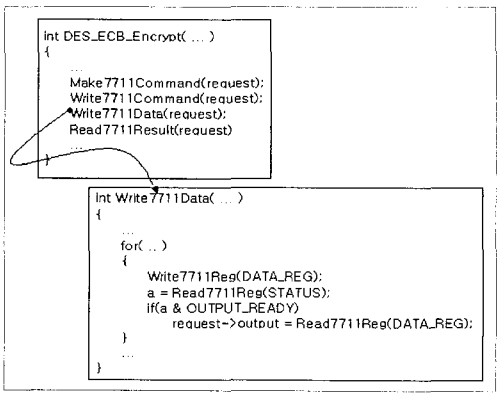


그림 8 암호화 함수의 명령어 구성 및 디바이스 호출

본 논문에서 구현하고 있는 암호화 프로세서를 장착한 PCI 보드의 경우, 인터럽트(interrupt)나 DMA(direct memory access) 기능이 추가되어 있지 않다. 따라서 시스템 프로세서는 디바이스의 상태 레지스터를 계속적으로 폴링함으로 상태를 확인하고 데이타를 입력하거나, 작업이 완료된 데이타를 가져오게 된다.

4.2.5 ioctl을 통한 암호화 처리된 데이타의 반환

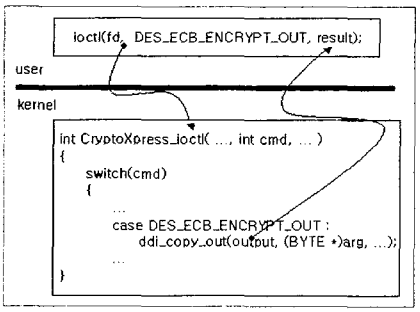


그림 9 ioctl을 통한 암호화 처리된 데이타의 반환

ioctl 호출을 통해서 앞에서 언급한 4.2.1에서 4.2.4까지의 과정을 거치면서 암호화 처리가 된 데이타를 반환한다. 여기에서는 4.2.3의 과정에서 ioctl을 통해서 애플리케이션에서 커널 안으로 데이타를 복사하는데 사용되었던 ddi_copy_in의 반대 역할을 수행하는 ddi_copy_out이라는 함수가 사용된다. 이 두 가지 함수는 앞에서 언급한 솔라리스 DDI/DKI에서 제공해주는 함수이며 커널과 유저 영역간의 데이타 복사 작업을 안정적으로 수행해준다.

4.2.6 복호화 과정

암호화 과정과 동일한 과정을 수행하게 된다. 다만, ioctl 호출에 사용되는 명령어가 DES_ECB_ENCRYPT_IN, DES_ECB_ENCRYPT_OUT에서 DES_ECB_DECRYPT_IN과 DES_ECB_DECRYPT_OUT이 사용된다.

4.3 암호화 라이브러리의 설계 및 구현

암호화 기능을 필요로 하는 애플리케이션을 제작하기 위해서는 앞에서 언급한 암호화 PCI 보드를 제어하는 디바이스 드라이버를 이용해야 한다. 하지만 애플리케이션을 제작하는 사람의 경우 시스템에 암호화 보드가 존재하는지 또는 소프트웨어적으로 구현된 암호화 함수 등이 존재하는지를 신경 쓰지 않고 애플리케이션을 개발해야 한다. 즉 암호화 PCI 보드의 존재유무에 상관없이 애플리케이션에서는 필요한 암호화 기능을 제공받을 수 있는 투명성이 제공되어야 한다. 이를 위해서 암호화 PCI 보드를 사용하는 시스템에 대해서는 PCI 보드를 활용하고, 그렇지 않은 시스템에 대해서는 소프트웨어로 구현된 암호화 기능을 제공할 수 있는 암호화 라이브러리를 개발하였다. 본 연구에서는 암호화 PCI 보드가 없는 경우에, 소프트웨어로 구현된 암호화 함수를 제공하고 있으며, 라이센스가 공개된 코드를 사용하여 라이브러리를 구축하였다. 이렇게 애플리케이션에게 투명한 서비스를 제공해주기 위해 라이브러리에서는 Crypto_Init 함수를 우선적으로 호출 한다. Crypto_Init 함수는 디바이스 드라이버에 대해 open을 시도하며, 시스템에 디바이스가 존재하는 경우 open의 반환 값으로 0보다 크거나 같은 수를 반환하고, 디바이스가 없는 경우는 -1을 반환한다. 그리고 디바이스가 존재하는 경우에 암호화 기능을 디바이스 드라이버에게 요청하고, 그렇지 않는 경우는 소프트웨어로 작성된 함수를 호출하여 암호화 기능을 처리하게 된다.

5. 실험 및 결과 분석

본 장에서는 실험을 통하여 하드웨어적으로 구현된 암호화 기법에 대한 성능을 분석하였다. 하드웨어 암호화 기법의 정확한 특성을 파악하기 위해서 크게 두 가지 범주로 나누어서 실험을 하고 결과를 기술하였다. 첫 번째 단계에서는 하드웨어 암호화 기능이 수행되는 각 구간에 대한 마이크로 벤치마크(micro benchmark)를 수행하였으며, 이를 통하여 하드웨어 암호화 기법이 수행됨에 있어서 성능에 영향을 주는 부분을 찾을 수 있다. 두 번째는 소프트웨어적으로 구현된 암호화 알고리즘과 하드웨어 암호화 기법을 상호 비교함으로, 하드웨어적인 접근 방식이 전체 시스템 성능에 어느 정도 영향을 주는지 분석하는 매크로 벤치마크(macro benchmark)를 수행하였다. 실험에서 사용된 하드웨어 플랫폼은 Sun Ultra10 워크스테이션이며, 333 Mhz CPU와 메인 메모리 512MB으로 구성되었고, 운영체제는 솔라리스 7을 사용하고 있다.

5.1 마이크로 벤치마크

하드웨어 보드를 사용한 암호화 연산과정은 여러 단계를 거쳐서 수행되며, 각 단계에서 어느 정도의 시간을 소모하는지 분석하는 것은 암호화 시스템의 성능을 검증하기 위해서 필수적인 일이다. 따라서 사용자 영역에서 수행되는 임의의 프로세스가 일정한 크기의 입력 데이타를 암호화 하드웨어 보드에 전송하고 암호화된 데이타가 암호화를 요청한 프로세스로 재전송되는 각 과정에 대한 세부적인 프로파일링(profiling)을 수행하였다. 이를 통하여 하드웨어적인 암호화 기법을 구현하는데 있어서 가장 많은 비용을 차지하는 구간이 어느 곳인지를 밝혀내어

암호화 보드의 성능상의 문제점을 분석한다.

본 실험에서 프로파일링을 하기 위한 구간을 크게 다섯 단계로 나누었다. 첫 번째 단계는 애플리케이션에서 라이브러리의 첫 ioctl 호출이 있기 전까지 걸린 시간이며, 두 번째 단계는 ioctl 호출 후, 디바이스에 명령어를 쓰기 전까지 걸린 시간 그리고 세 번째 단계는 디바이스의 레지스터에 첫 번째 데이타를 쓰기 직전까지 걸린 시간이다. 네 번째 단계는 디바이스에 모든 데이타를 쓰고, 암호화 된 모든 데이타를 받기까지 걸린 시간이며, 마지막으로 다섯 번째 단계는 그 이후부터 암호화 연산을 마치기까지 걸린 시간을 의미한다. 언급한 각 단계를 차례대로 A, B, C, D, E로 표시하였으며, 입력 데이타는 8바이트, 256바이트 그리고 8096바이트를 사용하여 DES 암호화 연산을 수행하였다. 실험은 10000 번의 암호화 작업을 수행한 후 그 평균값을 계산하는 방법을 사용하였으며, 이를 통하여 실험에서 발생하는 오차를 최소화하였다. 표 1은 실험 결과를 나타내고 있으며, 그림 10은 이를 그래프화 시킨 것이다. 표 1에서 나타내고 있는 수치들은 각각의 구간에 걸린 시간을 마이크로세컨드 단위로 표시한 것이다.

표 1 DES의 구간별 수행시간

	A	B	C	D	E
8	2.1	7.5	13.9	6.7	10.6
256	2.0	9.3	17.6	88.9	12.0
8092	2.6	34.8	138.2	2826.9	210.7

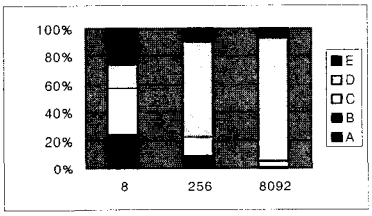


그림 10 DES의 구간별 프로파일링 결과

그림 10을 보면 입력 데이타의 크기가 커질수록 전체 시간에서 구간 D가 차지하는 폭이 커지는 것을 볼 수 있다. 구간 D의 작업을 살펴보면, 입력 데이타를 디바이스에 쓰고, 상태 레지스터를 확인한 후 암호화 작업이 끝났으면 암호화 된 데이타를 받아오고, 다시 입력 데이타를 쓰는 작업이 수행되는 곳이다. 즉 하드웨어 보드를 이용한 암호화 과정에 있어서 대부분의 암호화 처리 시간이 입력 데이타를 전송하는 과정에서 소모하고 있으며, 실제적으로 하드웨어 보드에서 암호화를 처리하는 연산에서는 거의 시간이 소모되고 있지 않음을 보이는 것이다. 예를 들어서 입력 데이타의 크기가 8 Kbyte인 경우 입력 데이타를 전송하는데 걸리는 시간이 전체시간의 90% 가까이를 차지하고 있는 것이며 따라서, 암호화 보드의 성능 개선을 위해서는 이 부분의 성능 개선이 필수적이라고 할 수 있다. 이것은 대부분의 암호화 디바이스가 PIO(Programmed IO) 방식으로 데이타를 교환하기 때문에 생기는 문제점이다.

5.2 매크로 벤치마크

매크로 벤치마크에서는 하드웨어 암호화 기법을 사용했을 때, 어느 정도의 성능 향상을 기대할 수 있는지 분석하였다. 첫 번째 실험 방법은 입력 데이타의 크기를 달리하여 DES와 Triple DES 연산을 하드웨어적인 기법과 소프트웨어적인 기법으로 각각 수행한 후, 수행 시간을 비교하여 보았다. 이러한 방법을 통하여 입력 데이타의 크기가 하드웨어적인 또는 소프트웨어적인 암호화 기법의 성능에 어떤 영향을 주는지 알 수 있다. 두 번째 실험 방법은 시스템 프로세서에 부하를 변화시키면서 입력 데이타의 크기를 달리하여 DES와 Triple DES 연산을 하드웨어적인 기법과 소프트웨어적인 기법으로 수행시키고 수행 시간을 비교하여 보았다. 이것은 시스템 프로세서의 부하가 하드웨어와 소프트웨어적인 암호화 연산에 미치는 영향을 분석하고자 한 것이다.

5.2.1 입력 데이타의 크기에 따른 수행 시간 비교

표 2는 시스템의 유휴 상태에서 암호화 수행 시간을 측정한 것이며, 입력 데이타의 크기를 변화시키면서 하드웨어적으로 수행한 DES(DES-HW) 및 Triple DES (3DES-HW), 그리고 소프트웨어적으로 수행한 DES(DES-SW)와 Triple DES(3DES-SW)의 측정치를 보이고 있다. 그림 11과 12는 표 2의 데이타를 그래프화 시킨 것이며, X축은 입력 데이타의 크기를 바이트 단위로 나타낸 것이며, Y축은 수행시간을 마이크로세컨드 단위로 나타낸 것이다.

표 2 암호화 알고리즘별 수행시간

	8	16	32	64	128	256	512	1024	2048	4096	8192
DES-HW	33.6	37.1	44.2	54.9	77.8	123.5	214.8	397.4	747.1	1459.9	2888.3
3DES-HW	34.8	37.6	44.7	57.3	79.5	125.5	215.9	400.4	749.2	1463.5	2891.5
DES-SW	45.8	48.0	51.8	58.8	75.6	102.2	161.2	280.1	519.0	991.2	1937.2
3DES-SW	134.3	141.2	149.3	172.0	203.5	284.1	435.8	746.5	1364.7	2597.6	5081.6

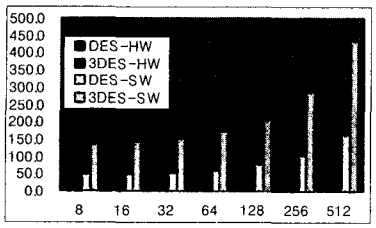


그림 11 암호화 알고리즘별 수행시간(작은 크기의 데이타)

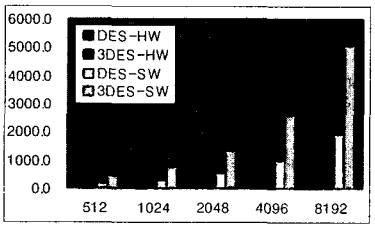


그림 12 암호화 알고리즘별 수행시간(큰 크기의 데이타)

입력 데이타의 크기를 8 Byte부터 8192 Byte까지 변화를 주어가면서 알고리즘별 수행시간을 비교해 보았을 때, 하드웨어적인 기법과 소프트웨어적인 기법의 차이를 발견할 수 있다. 하드웨어적 기법인 DES-HW와 3DES-HW는 암호 알고리즘의 복잡도에 상관없이 수행시간이 거의 비슷하게 나오는 것을 알 수 있으며, 입력 데이타의 크기가 증가함에 따라서 수행시간이 비슷하게 증가하고 있다. 여기서 알 수 있는 사실은 암호화 프로세서에서의 연산시간 차이만이 수행시간의 차이를 가져올 수 있는데, 거의 비슷한 수행시간을 보이고 있으므로, 암호화 프로세서에서 DES와 Triple DES 연산을 수행하는데 걸리는 시간적 차이는 전체 수행시간에 비해서 미미하다는 것을 알 수 있다. 반면에 소프트웨어 기법인 DES-SW와 3DES-SW는 암호 알고리즘의 복잡도에 영향을 받아서 수행 시간이 대략 두 배에서 세배정도 차이가 나고 있다. 즉 소프트웨어적으로 수행한 DES와 Triple DES의 수행시간을 비교해 보면, Triple DES가 DES를 3 번 사용하여 암호화를 처리하는 알고리즘이므로, 수행 시간의 차이는 예상된 결과라고 볼 수 있다. 소프트웨어적으로 수행한 DES-SW와 하드웨어적으로 수행한 DES-HW를 상호 비교한 경우, 128 Byte 보다 큰 입력 데이타에 대해서는 소프트웨어적으로 수행한 DES가 더 짧은 수행시간을 걸렸다. 이것에 대한 이유는 마이크로 벤치마크의 결과에서 알 수 있는 내용이다. 즉 마이크로 벤치마크에서 하드웨어 암호화 기법의 구간별 수행 시간을 살펴보면, 입력 데이타의 크기가 증가할수록 디바이스에 대한 PIO가 차지하는 비중이 점점 커지는 것을 알 수 있다. 하드웨어 DES는 데이타 교환을 위하여 폴링(polling)을 이용한 PIO방식을 사용함으로 메모리 복사 오버헤드가 늘어나고, 반면에 소프트웨어로 구현된 DES는 상대적으로 메모리 복사 오버헤드가 줄어든다. 따라서 하드웨어 DES에 비해서 소프트웨어 DES는 수행시간이 줄어들게 되는 것이다.

다음의 그림 13과 14는 암호화 프로세서의 성능 스펙과 암호화 보드의 성능을 비교분석한 그래프이다. 암호화 프로세서의 성능 스펙은 암호화 칩 자체가 처리할 수 있는 최대 성능을 의미한다.

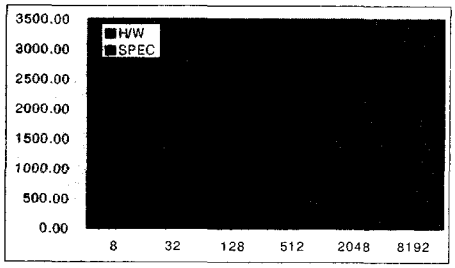


그림 13 DES의 암호화 보드 성능과 암호화 프로세서 성능 스펙 비교

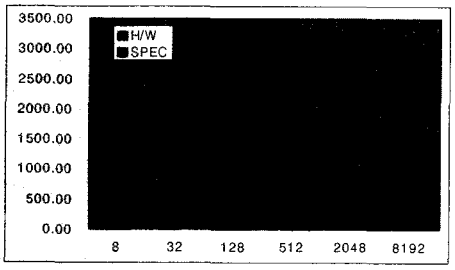


그림 14 Triple DES의 암호화 보드 성능과 암호화 프로세서 성능 스펙

그림 13과 14를 살펴보면 DES의 하드웨어 암호화 성능이 소프트웨어의 암호화 성능에 비해서 좋지 않게 나타나는 이유를 더욱 분명히 알 수 있다. 그림 13에서 알 수 있듯이 순수하게 암호화 프로세서가 처리할 수 있는 성능 스펙에 비해서, 암호화 보드에 암호화 프로세서를 적재하여 암호화 연산을 수행하였을 경우, 수행시간이 크게 차이가 나고 있다. 데이타의 크기에 따라서 성능의 차이가 가변적이지만 대략적으로 약 10배 가까이 수행시간의 차이를 보이고 있다. 즉 암호화 프로세서는 대략 10퍼센트 정도만 사용되고 나머지 90 퍼센트의 시간은

암호화 프로세서까지 데이타를 전달하는데 소요되었음을 알 수 있다. 결과적으로 암호화 프로세서의 성능이 좋아져도 데이타 전달에 소요되는 오버헤드에 의해서 전체 암호화 시스템의 성능은 크게 향상되지 않는 것이다.

5.2.2 부하 상황에서 암호화 성능 비교 분석

그림 15, 16은 데이타의 크기가 128 Byte일 경우에 부하 상황에서의 알고리즘 별 수행시간을 보여주고 있다. X축은 시스템의 부하정도를 표시하고 있는 것으로 단위는 %이다. Y축은 수행시간을 보여주고 있으며 단위는 마이크로세컨드이다. 그림 15는 소프트웨어적인 암호화 수행시간을 보여주고 있는데, 시스템의 부하에 따라서 Triple DES의 경우 DES보다 부하 상황에 영향을 더 받는 것을 확인할 수 있다. 반면 그림 16을 보면, 하드웨어적인 암호화 수행 시간을 보여주고 있는데, DES와 Triple DES가 거의 비슷한 수행시간을 보여주고 있음을 알 수 있다. 즉, 하드웨어적인 암호화 기법의 경우 알고리즘의 계산량에 상관없이 비슷한 수행시간을 보여주고 있으므로 시스템에 부하가 걸린 상황에서 하드웨어적인 암호화 기법이 더 효과적임을 알 수 있다. 이렇게 부하 상황에서 하드웨어적인 암호화 기법이 계산량에 상관없이 거의 같은 수행시간을 보여주는 이유로는 암호화 프로세서 연산에 걸리는 시간보다는 입출력에 걸리는 시간이 전체 수행시간의 대부분을 차지하고 있기 때문이다.

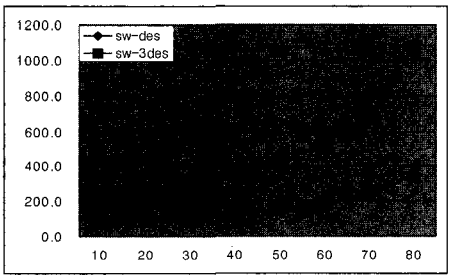


그림 15 부하 상황에서 소프트웨어적인 암호화 수행 시간

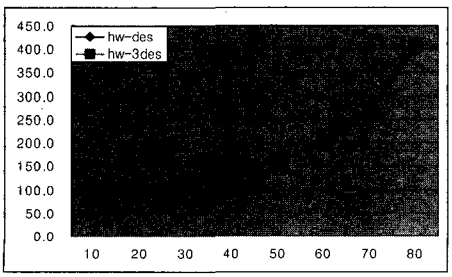


그림 16 부하 상황에서 하드웨어적인 암호화 수행 시간

6. 결 론

암호화 작업의 대부분은 많은 계산량을 가지고 있으며, 이것은 시스템 프로세서 자원을 독점하여 전체 시스템의 처리량을 떨어뜨리는 원인으로 작용한다. 이에 본 논문에서는 암호화 프로세서를 장착한 암호화 보드, 디바이스 드라이버 및 라이브러리를 구현하여 하드웨어적인 방식으로 암호화 작업을 수행하였을 경우의 성능을 소프트웨어적인 방식으로 암호화 작업을 수행하였을 경우와 함께 비교 분석하였다. 실험 결과, 암호화 작업을 수행하는 데 있어서 암호화 보드를 사용하는 하드웨어적인 방식이라고 하더라도 디바이스에 접근하는 메커니즘이 폴링을 통한 PIO인 경우 계속적으로 시스템 프로세서의 자원을 소모하므로 시스템 성능을 향상시키지 못하고 있음을 알 수 있다. 즉, 하드웨어적인 암호화 수행시간의 대부분은 디바이스와 데이타를 주고받기 위해 PIO를 하는데 사용되고 있으며, 이러한 PIO의 오버헤드는 오히려 DES 정도의 계산량을 가진 암호화 알고리즘에 대해서는 호스트 프로세서를 사용하는 것이 더 효율적이라고 할 만큼 그 오버헤드가 크다. 하지만, Triple DES와 같이 계산량이 많은 알고리즘의 경우 하드웨어적인 방식으로 암호화 작업을 수행하는 것이 약 2배에서 4배까지의 성능향상이 있음을 확인할 수 있었다. 이러한 문제점은 하드웨어 암호화 기법이 하드웨어 암호화 프로세서가 제공할 수 있는 성능 스펙의 10퍼센트 정도만을 사용하고 있으며, 90퍼센트는 응용 프로그램과 암호화 데이타를 송수신하는데 사용되고 있다는 성능 스펙에 대한 비교를 통해서 구체적으로 알 수 있다. 또한, 소프트웨어 암호화 기법은 호스트 프로세서에 부하가 증가하면 암호화 성능이 급격히 줄어드는데 비해서, 하드웨어 암호화 기법은 과부하가 발생하더라도 암호화 성능에 거의 영향을 받지 않고 수행되는 것을 보이고 있다. 이러한 이유는 하드웨어 암호화 기법의 경우 대부분의 시간을 암호화 데이타를 입출력하는 부분에서 소요하고 있으므로, 호스트 프로세서에 영향을 거의 받지 않게 되는 것이다. 본 연구의 향후 계획은 앞에서 언급한 바와 같이, 현재 하드웨어 기법에서 문제가 되고 있는 데이타 전송 지연 시간을 줄일 수 있는 방법에 대한 연구이며, 이러한 연구를 통하여 하드웨어 암호화 시스템의 전체 성능을 향상 시킬 수 있으리라 기대한다.

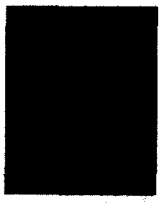
참 고 문 헌

[1] 김 철, 암호학의 이해, 영풍문고, 1996.
[2] H. Feistel, "Cryptography and Computer Privacy,"

Scientific American. Vol 228, 1973.
[3] National Institute of Standards and Technology, "Data Encryption Standard," FIPS Publication 46-1, January 1988.
[4] J. Dyer, R. Perez, S.W. Smith, and M. Lindemann, "Application support architecture for a high-performance, programmable secure coprocessor," 22nd National Information Systems Security Conference, October 1999.
[5] Tygar, J.D. and Yee, B.S., "Secure Coprocessors in Electronic Commerce Applications," Proceedings 1995 USENIX Electronic Commerce Workshop, 1995, New York.
[6] J. D. Tygar and Bennet Yee. "Dyad: A system for using physically secure coprocessors," Technical report, Carnegie Mellon University, May 1991.
[7] P. C. Clark and L. J. Hoffmann. "BITS: A Smartcard Protected Operating System," Communications of the ACM. 37: 66-70. November 1994.
[8] S.W. Smith, S.H. Weingart. "Building a High-Performance, Programmable Secure Coprocessor," Computer Networks (Special Issue on Computer Network Security.) 31: 831-860. April 1999.
[9] S. W. Smith. "Secure Coprocessing Applications and Research Issues," Los Alamos Unclassified Release LAUR-96-2805, Los Alamos National Laboratory. August 1996.
[10] S. W. Smith and V. Austel. "Trusting trusted hardware: Towards a formal model for programmable secure coprocessors," In Proceedings of the Thrid USENIX Workshop on Electronic Commerce, September 1998.
[11] S.W.Smith,E.R.Palmer,S.H.Weingart."Using a High-Performance, Programmable Secure Coprocessor," Proceedings, Second International Conference on Financial Cryptography. Springer-Verlag LNCS, 1998.
[12] C . K. Koc, "RSA Hardware Implementation," TR 801, RSA Laboratories, April 1996.
[13] A. Elbirt, "An FPGA Implementation and Performance Evaluation of the CAST-256 Block Cipher," Technical Report, Cryptography and Information Security Group, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, Worcester, MA, May 1999.
[14] Kaps, J., and Paar, C., "Fast DES Implementation for FPGAs and its Application to a Universal Key-Search Machine," 5th Annual Workshop on Selected Areas in Cryptography (SAC '98), Queen's University, Kingston, Ontario, Canada.
[15] Tom Shanley, Don Anderson, PCI System Architecture, MindShare Inc.
[16] http://www.stitec.com/
[17] 128비트 블록 암호알고리즘(SEED) 개발 및 분석보고서, 정보보호진흥원 연구보고서 1998.12.
[18] http://www.stitec.com/product/scc102l.html
[19] Writing Device Drivers, Sun Microsystems Inc, 1998.

아 재 용
2000년 고려대학교 컴퓨터학과(학사)
2002년 고려대학교 컴퓨터학과(석사)
2002년 ~ 현재 LG 전자 우면 R&D 연구소. 관심분야는 운영체제, 보안 운영체제, 네트워크

고 영 웅
1997년 고려대학교 컴퓨터학과(학사)
1999년 고려대학교 컴퓨터학과(석사)
1999년 ~ 현재 고려대학교 컴퓨터학과 박사과정. 관심분야는 운영체제, 실시간 시스템, 멀티미디어



홍 철 호
2001년 고려대학교 컴퓨터학과(학사)
2001년 ~ 현재 고려대학교 컴퓨터학과 석사과정. 관심분야는 운영체제, 보안 운영체제

유 혁
정보과학회논문지 : 정보통신
제 29 권 제 3 호 참조