情 報 保 護 學 會 誌
第 16 卷 第 3 號, 2006. 6

64비트 마이크로프로세서에 적합한 블록암호 ARIA 구현방안

장 환 석*, 이 호 정*, 구 본 욱*, 송 정 환**

요 약

본 논문에서는 한국 산업규격 KS 표준 블록암호 알고리즘인 ARIA의 핵심논리들을 64비트 프로세서 환경에서 효율적으로 구현하는 방법을 제안하고, 제안된 방법으로 구현된 ARIA를 대표적인 64비트 마이크로프로세서인 Intel Pentium4 x64, Intel Itanium, Intel XEON(EM64T)을 이용하여 그 효율성을 평가하였으며, 32비트 프로세서에 적합하게 구현된 ARIA와 효율성을 비교하고 문제점을 분석하였다.

Ⅰ. 서 론

마이크로프로세서의 발달은 컴퓨팅 환경의 발전과 함께 PC, 워크스테이션, 서버 등의 고성능화를 실현시켰다. 인텔은 1971년에 최초의 4비트 마이크로프로세서인 4044를 개발하였고, 1978년에 최초의 16비트 프로세서인 8086/8088을 발표했다. 8086은 1 메가바이트의 메모리를 처리할 수 있었으며, 가상메모리는 불가능 하였으나, 1982년에 발표한 16비트 프로세서 80286은 가상메모리가 가능하며, 16 메가바이트의 메모리를 처리할 수 있었다.

최초의 32비트 마이크로프로세서는 1985년에 발표된 80386이며, 현재 많이 사용되고 있는 펜티엄 4 시리즈 까지도 32비트 프로세서의 한계를 넘지 못하고 있다.

최근 주요 마이크로프로세서 공급원인 Intel 과 AMD 는 64비트 프로세서인 Itanium2, Pentium4 x64, Athlon64, Turion, Opteron 등을 발매하고 있으며, 그에 발맞추어 Windows server 2003 x64, Windows XP x64 등 64비트 OS와 SQL Server 2005, Visual Studio 2005 등의 Application을 출시하며 64비트 컴퓨팅 시대를 열었다[1].

전자정부 시스템 등 다양한 정보보호 환경을 대비하여 개발된 블록암호 알고리즘 ARIA는 2004년 12월에 한국 산업규격 KS 표준으로 선정되었으며, 안전성과 효율성은 미국, 유럽, 일본의 표준 블록암호 알고리즘과 비교하여 대등한 정도의 수준을 가지고 있다. ARIA는 기본적으로 하드웨어와 8비트 프로세서 환경에 적합한 형태를 가지고 있으며, 32비트 프로세서 환경에서도 효율적으로 구현될 수 있는 방안이 제시되었다[2].

본 논문에서는 ARIA의 핵심논리들을 64비트 프로세서 환경에서 효율적으로 구현하는 방법을 소개하고, 대표적 64비트 프로세서인 Intel Pentium4 3.0 GHz (EM64T), Intel Itanium 1.7 GHz (IA64), Intel Xeon 2.0 GHz (EM64T)을 이용하여 효율성을 평가하였으며, 32비트 프로세서에 적합하게 구현된 ARIA와 효율성을 비교하고 문제점을 분석하였다.

1. 64비트 프로세서[1]

현재 사용되고 있는 64비트 마이크로프로세서는 크게 x64(64비트 extension)[5] 부류와 IPF(Intel Itanium Processor Family)[6] 부류로 나눌 수 있다. x64 부류는 기존 32비트 프로세서 구조를 64비트로 확장한 형태이며 AMD사의 Athlon64, Opteron64 와 Intel사의 EM64T(Extended memory 64 technology)가 적용된 XEON, Pentium4 x64 등이 이 부류에 속한다. 이 부류의 프로세서들은 64비트 프로세서 이면서 16비트, 32비

* 한양대학교 수학과 박사과정(campjang,mathlead,kidkoo@ihanyang.ac.kr)
** 한양대학교 수학과 부교수(camp123@hanyang.ac.kr)

트 응용소프트웨어와 쉽게 호환된다. IPF 부류는 병렬구조에 기반한 프로세서이며, Intel 사의 Itanium이 이에 속한다. 32비트 응용소프트웨어와의 호환을 위해서는 별도의 에뮬레이션이 필요하다.

64비트 프로세서가 32비트 프로세서에 비하여 월등히 향상된 점은 메모리 운용능력이다. 메모리 운용능력의 비교는 다음 표와 같다.

〔표 1〕 메모리 운용능력의 비교

	32비트	64비트
총 가상 주소 공간	4 GB	16 TB
32비트 프로세스를 위한 가상 주소 공간	2 GB	2 GB
64비트 프로세스를 위한 가상 주소 공간	없음	8 TB
Paged pool	470 MB	128 GB
Non-paged pool	256 MB	125 GB
시스템 캐쉬	1 GB	1 TB

x64 프로세서 내부에는 Legacy 모드를 지원하기 위해 32비트 커널이 존재하며, Long 모드는 64비트 모드와 호환모드를 자동으로 스위칭 한다.

x64 프로세서의 구성은 다음과 같다.

〔표 2〕 x64 프로세서의 구성

동작모드		OS	응용 프로그램 재컴파일	초기설정		레지스터 확장	GPR 크기
				주소 크기	데이터 크기		
Long 모드	64bit 모드	새로운 64비트 OS	필요	64 비트	32 비트	가능	64비트
	호환 모드		불필요	32 비트		불가능	32비트
Legacy 모드		Lagacy 32비트 OS					

기존의 32비트 프로세서는 한 번에 처리할 수 있는 변수의 최대 크기가 32비트인데 비해, 64비트 프로세서는 64비트의 변수를 사용할 수 있으며, 포인터의 크기도 64비트를 사용할 수 있기 때문에 메모리 주소의 수가 기존 32비트 보다 43억 배 정도 더 많다. 따라서 기존 32비트 프로세서가 기가바이트 급의 메모리 운용능력을 가진 것에 비해, 64비트 프로세서는 테라바이트 급의 메모리 운용능력을 가지게 된다. 또한, 명령어 처리 집합(Instruction Set Architecture, ISA)과 레지스터(Register)의 크기가 64비트를 지원하기 때문에 역시 이론적으로 2^{32} 배의 속도 향상을 기대할 수 있다.

32비트와 64비트의 호완성은 아직 논란의 여지가 있지만, 32비트 운용프로그램은 64비트 환경에서도 대부분 사용 가능하다. 64비트 프로세서는 기존 32비트 프로세서에서 사용하던 데이터 형태와 논리적 연산을 모두 포함하고 있으며, 64비트 형 데이터 형태를 추가적으로 운용할 수 있기 때문이다.

현재 64비트 프로세서를 사용하는 시스템의 경우 32비트 시스템보다 속도 면에서 크게 효율적이라고는 할 수 없다. 프로세서의 내부 연산은 64비트로 이루어지지만, External address bus나 Data bus 등의 외부적 입/출력부는 32비트로 동작하고 있기 때문이다. 따라서 64비트 운영체제를 사용하고 있는 시스템이라 하더라도 64비트 고유 연산을 사용한 대용량 프로그램이 아니라면, 프로세서의 처리속도가 같은 32비트 시스템과 속도의 효율성 차이는 크지 않다. 결국 64비트 프로세서는 속도의 향상을 위해 개발된 것이 아니라 대용량의 데이터를 처리해야 하는 서버 등에서 활용하기 위해 개발되었기 때문에, 단일 시스템에서의 사용에서는 큰 속도 향상을 기대하기 어렵다.

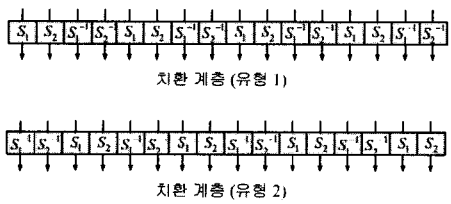
II. 본 론

1. 블록암호 알고리즘 ARIA〔2〕

블록암호 알고리즘 ARIA는 고정된 128비트 입/출력에, 가변길이(128, 192, 256비트) 키에 따른 가변 라운드(12, 14, 16)가 적용된 대합(Involution) SPN 구조의 블록암호이다. 내부 함수는 다음과 같이 세 부분으로 구성되어 있다.

- 라운드 키 삽입 : 128비트 라운드 키를 라운드 입력 128비트와 비트별 XOR 한다.
- 치환 계층 : 두 유형의 치환 계층이 있으며 각각은 2 종의 8비트 입/출력 S-box와 그들의 역변환으로 구성된다.
- 확산 계층 : 간단한 16x16 대합이진행렬을 사용한 바이트 간의 확산 함수로 구성되어 있다.

치환 계층은 8비트 입/출력을 가지는 S-box S_1과 S_2와 그의 역 S_1^{-1}와 S_2^{-1}로 구성되어 있다. 이들 치환 계층은 테이블 참조로 구현하는 것이 일반적이며 최적의 선택이라 할 수 있다. ARIA의 치환 계층은 다음 [그림 1] 과 같이 구성되어 있으며,



[그림 1] ARIA의 S-box 계층

(유형 1)은 홀수라운드에, (유형 2)는 짝수라운드에 사용된다. 이는 전체 구조가 대합 형태를 갖도록 하기 위함이다.

확산 계층은 ARIA를 다른 블록암호 알고리즘과 구별 짓는 주요 부분으로 16x16 대합이진행렬을 사용한다. 확산 함수는 입력 16 바이트에 대하여 바이트 단위의 행렬 곱을 수행한 결과의 16 바이트를 출력으로 한다. 다음은 ARIA에 사용된 확산 계층을 나타낸 것이다.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \end{pmatrix} = \begin{pmatrix} 0&0&0&1&1&0&1&0&1&1&0&0&0&1&1&0 \\ 0&0&1&0&0&1&0&1&1&1&0&0&1&0&0&1 \\ 0&1&0&0&1&0&1&0&0&0&1&1&1&0&0&1 \\ 1&0&0&0&0&1&0&1&0&0&1&1&0&1&1&0 \\ 1&0&1&0&0&1&0&0&1&0&0&1&0&0&1&1 \\ 0&1&0&1&1&0&0&0&0&1&1&0&0&0&1&1 \\ 1&0&1&0&0&0&0&1&0&1&1&0&1&1&0&0 \\ 0&1&0&1&0&0&1&0&1&0&0&1&1&1&0&0 \\ 1&1&0&0&1&0&0&1&0&0&1&0&0&1&0&1 \\ 1&1&0&0&0&1&1&0&0&0&0&1&1&0&1&0 \\ 0&0&1&1&0&1&1&0&1&0&0&0&0&1&0&1 \\ 0&0&1&1&1&0&0&1&0&1&0&0&1&0&1&0 \\ 0&1&1&0&0&0&1&1&0&1&0&1&1&0&0&0 \\ 1&0&0&1&0&0&1&1&1&0&1&0&0&1&0&0 \\ 1&0&0&1&1&1&0&0&0&1&0&1&0&0&1&0 \\ 0&1&1&0&1&1&0&0&1&0&1&0&0&0&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{pmatrix}$$

라운드 키 삽입 부분은 키 스케줄에서 생성된 라운드 키와 라운드 함수 입력간의 XOR연산으로 이루어져 있다. 키 스케줄은 암호화에 사용되는 라운드 함수를 응용한 3 라운드 256비트 Feistel 구조를 이용하여 4개의 128 비트 초기값(W_0, W_1, W_2, W_3)을 생성 한 후, 이것을 이용하여 라운드 키를 생성하는 구조이다. 각 암호화 라운드 키는 다음과 같은 방법으로 생성된다.

$$\begin{aligned} ek_1 &= (W_0) \oplus (W_1^{\gg 19}), & ek_2 &= (W_1) \oplus (W_2^{\gg 19}), \\ ek_3 &= (W_2) \oplus (W_3^{\gg 19}), & ek_4 &= (W_0^{\gg 19}) \oplus (W_3), \\ ek_5 &= (W_0) \oplus (W_1^{\gg 31}), & ek_6 &= (W_1) \oplus (W_2^{\gg 31}), \\ ek_7 &= (W_2) \oplus (W_3^{\gg 31}), & ek_8 &= (W_0^{\gg 31}) \oplus (W_3), \\ ek_9 &= (W_0) \oplus (W_1^{\ll 61}), & ek_{10} &= (W_1) \oplus (W_2^{\ll 61}), \\ ek_{11} &= (W_2) \oplus (W_3^{\ll 61}), & ek_{12} &= (W_0^{\ll 61}) \oplus (W_3), \\ ek_{13} &= (W_0) \oplus (W_1^{\ll 31}), & ek_{14} &= (W_1) \oplus (W_2^{\ll 31}), \\ ek_{15} &= (W_2) \oplus (W_3^{\ll 31}), & ek_{16} &= (W_0^{\ll 31}) \oplus (W_3), \\ ek_{17} &= (W_0) \oplus (W_1^{\ll 19}) \end{aligned}$$

복호화 라운드 키는 암호화 라운드 키와 다르며 암호화 라운드 키로부터 유도된다. 먼저 키의 순서가 바뀌고 처음과 마지막 라운드 키를 제외한 암호화 라운드 키에 대한 확산함수의 결과가 복호화 라운드 키가 된다.

2. 64비트 환경에서의 ARIA 구현 방안

64비트 환경에서 ARIA 구현의 핵심은 확산계층에 대한 구현이다. ARIA의 확산계층 A를

$$A = L \cdot M \cdot L = L \cdot Q \cdot L \cdot P \cdot M$$

이라 할 때 고려해야 할 사항은 다음과 같다.

- $P \cdot M$ 과 S-box 출력들을 결합하여 64비트 워드들의 참조 테이블로 구현
- 행렬 L을 64비트 환경에서 효율적으로 구현
- 치환 행렬 Q를 64비트 환경에서 효율적으로 구현

64비트 환경에서는 하나의 정수형 변수 레지스터가 64비트로 정의되므로, 64비트 워드 단위로 연산을 구현하는 것이 가장 효율적이라 할 수 있다. 우선 $P \cdot M$ 과 S-box의 출력들을 결합하여 64비트 워드들의 테이블로 구현하는 방법에 대해 알아보자. 32비트 환경에서의 구현과 유사한 방법을 이용하되, 입출력을 64비트로 설정한다. 8비트 입출력의 S-box 출력 8개를 연접하여 하나의 64비트 워드를 생성한다. [그림 1]에서 보는 바와 같이 총 2 종류의 유형이 존재하며, 이를 각각의 유형별 128비트 입력에 대해, 다음과 같이 64비트 워드 두 개로 정의한다.

$$\begin{pmatrix} A_{MSB}^{64} = (S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1})^T \\ A_{LSB}^{64} = (S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1})^T \end{pmatrix}$$

$$\begin{pmatrix} B_{MSB}^{64} = (S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2)^T \\ B_{LSB}^{64} = (S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2)^T \end{pmatrix}$$

〔그림 1〕의 치환계층 (유형 1)에 대한

$P \cdot M \cdot \begin{pmatrix} A_{MSB}^{64} \\ A_{LSB}^{64} \end{pmatrix}$에 해당하는 연산은 다음과 같이 표현 된다.

$$\begin{bmatrix} 0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&1&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0 \end{bmatrix} \cdot \begin{bmatrix} A_{MSB}^{64} \\ \\ \\ A_{LSB}^{64} \end{bmatrix}$$

위 식에서 A_{MSB}^{64}가 참여하는 연산만 살펴보면 아래와 같음을 알 수 있다.

$$\begin{bmatrix} 0&1&1&1&0&0&0&0 \\ 1&0&1&1&0&0&0&0 \\ 1&1&0&1&0&0&0&0 \\ 1&1&1&0&0&0&0&0 \\ 0&0&0&0&0&1&1&1 \\ 0&0&0&0&1&0&1&1 \\ 0&0&0&0&1&1&0&1 \\ 0&0&0&0&1&1&1&0 \end{bmatrix} \cdot \begin{bmatrix} A_{MSB}^{64} \\ \\ \\ \\ \\ \\ \\ \\ \end{bmatrix}$$

이 결과를 S-box들의 관점에서 보면 다음과 같으며, 열(Column) 들의 연산으로 볼 수 있다. 이 각각의 열들의 형태로 부터 8개의 테이블을 구성하게 된다.

S-box 계층의 A_{LSB}^{64} 또한 위와 동일하게 연산이 이루어지므로, 동일한 테이블을 사용할 수 있다. 따라서 치환계층 (유형 2)를 고려한다면 64비트 구현을 위해서는 총 16개의 테이블이 필요하다.

다음으로, 행렬 L을 64비트 환경에서 효율적으로 구현하는 방법에 대해 고려해 보자. L을 64비트 환경에서 효율적으로 구현하기 위하여 적당한 형태로의 변환이 필요하다. 이를 위해 치환행렬을 사용하며, 열 치환과 행 치환 둘 중 어느 것으로도 변환 가능하므로, 적절히 이용하도록 한다.

$$\begin{bmatrix} 0\oplus S_2 \oplus S_1^{-1} \oplus S_2^{-1} & \oplus 0 & \oplus 0 & \oplus 0 \\ S_1 \oplus 0 \oplus S_1^{-1} \oplus S_2^{-1} & \oplus 0 & \oplus 0 & \oplus 0 \\ S_1 \oplus S_2 \oplus 0 \oplus S_2^{-1} & \oplus 0 & \oplus 0 & \oplus 0 \\ S_1 \oplus S_2 \oplus S_1^{-1} \oplus 0 & \oplus 0 & \oplus 0 & \oplus 0 \\ 0 \oplus 0 \oplus 0 \oplus 0 & \oplus S_2 \oplus S_1^{-1} & \oplus S_2^{-1} & \\ 0 \oplus 0 \oplus 0 \oplus 0 & \oplus S_1 \oplus 0 & \oplus S_1^{-1} & \oplus S_2^{-1} \\ 0 \oplus 0 \oplus 0 \oplus 0 & \oplus S_1 \oplus S_2 \oplus 0 & & \oplus S_2^{-1} \\ 0 \oplus 0 \oplus 0 \oplus 0 & \oplus S_1 \oplus S_2 \oplus S_1^{-1} & & \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ S_1 \\ S_1 \\ S_1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} S_2 \\ 0 \\ S_2 \\ S_2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} S_1^{-1} \\ S_1^{-1} \\ 0 \\ S_1^{-1} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} S_2^{-1} \\ S_2^{-1} \\ S_2^{-1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ S_1 \\ S_1 \\ S_1 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ S_2 \\ 0 \\ S_2 \\ S_2 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ S_1^{-1} \\ S_1^{-1} \\ 0 \\ S_1^{-1} \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ S_2^{-1} \\ S_2^{-1} \\ S_2^{-1} \\ 0 \end{bmatrix}$$

선형변환 A에서 $L \cdot Q \cdot L$을 대상으로 하며, $L' \cdot Q' \cdot Q \cdot Q'' \cdot L''$의 형태로 변환 한다. 이는 행렬 L에 각각 행 치환과 열 치환을 적용한 것이다. 즉,

$$L = L' \cdot Q' = Q'' \cdot L'' \qquad (1)$$

이다. 행렬 L'을 아래와 같이 설정하고, 또한 $L' = L''$가 되도록 설정한 후 식 (1)을 만족하게 하는 치환행렬 Q'와 Q''을 찾는다.

$$L' = \begin{bmatrix} 1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&1&0&0&0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&1&0&0&0&0&0&0&0&1&0 \\ 0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&0&0&1&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&0 \\ 0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1 \\ 0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1 \end{bmatrix}$$

이때 사용된 치환행렬 Q'와 Q''은 아래와 같다.

$$Q' = Q'' = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix}$$

행렬 L'의 형태를 설정한 방법에 대해 살펴보자. L의 형태는 다음과 같이 4x4 단위행렬들로 표현된다.

$$L = \begin{bmatrix}
I_{4\times4} & I_{4\times4} & I_{4\times4} & O_{4\times4}\\
I_{4\times4} & O_{4\times4} & I_{4\times4} & I_{4\times4}\\
I_{4\times4} & I_{4\times4} & O_{4\times4} & I_{4\times4}\\
O_{4\times4} & I_{4\times4} & I_{4\times4} & I_{4\times4}
\end{bmatrix}$$

하지만 위의 형태는 32비트 워드들에 관한 식으로는 간단히 표현되어 구현이 쉽지만, 64비트 워드를 다루는 데에는 효율적이지 못한 형태이다. 그러므로 64비트 워드를 다루기 위해 치환 행렬을 적용하여 행렬의 구성이 최대한 많은 $I_{8\times8}$을 포함하도록 변형하여야 한다. 다음은 변형된 L'의 형태이다.

$$L' = \begin{bmatrix}
I_{4\times4} & I_{4\times4} & \multicolumn{2}{c}{\multirow{2}{*}{$I_{8\times8}$}}\\
I_{4\times4} & I_{4\times4} & &\\
\multicolumn{2}{c}{\multirow{2}{*}{$I_{8\times8}$}} & I_{4\times4} & I_{4\times4}\\
& & I_{4\times4} & I_{4\times4}
\end{bmatrix}$$

위의 형태도 $I_{4\times4}$ 형태를 포함하고 있으므로, 입력변수(A, B, C, D)를 32 비트 워드로 간주하자. 다음과 같은 결과를 얻을 수 있다.

$$\begin{bmatrix}
I_{4\times4} & I_{4\times4} & \multicolumn{2}{c}{\multirow{2}{*}{$I_{8\times8}$}}\\
I_{4\times4} & I_{4\times4} & &\\
\multicolumn{2}{c}{\multirow{2}{*}{$I_{8\times8}$}} & I_{4\times4} & I_{4\times4}\\
& & I_{4\times4} & I_{4\times4}
\end{bmatrix} \cdot \begin{bmatrix} A\\B\\C\\D \end{bmatrix}$$

$$= \begin{bmatrix} A\oplus B\oplus C\\ A\oplus B\oplus D\\ A\oplus C\oplus D\\ B\oplus C\oplus D \end{bmatrix} = \begin{bmatrix} A\oplus B\oplus C\\ B\oplus A\oplus D\\ A\oplus C\oplus D\\ B\oplus D\oplus C \end{bmatrix}$$

입력된 4개의 32비트 워드를 다음과 같이 연접하여 2개의 64비트 워드로 간주하자.

64비트 워드 : $X=(A\|B)^T$, $Y=(C\|D)^T$ 라 한다면,

$$L' \cdot \begin{pmatrix} X\\ Y \end{pmatrix} = \begin{pmatrix} (A\|B)^T\oplus(B\|A)^T\oplus(C\|D)^T\\ (A\|B)^T\oplus(C\|D)^T\oplus(D\|C)^T \end{pmatrix}$$

$$\Rightarrow \begin{pmatrix} X\oplus((X\gg32)\oplus(X\ll32))\oplus Y\\ X\oplus Y\oplus((Y\gg32)\oplus(Y\ll32)) \end{pmatrix}$$

로 64비트 구현 환경에 대응하여 구현할 수 있다.

마지막으로, $Q'\cdot Q\cdot Q''$에 대한 구현을 살펴보자. $Q'\cdot Q\cdot Q''$에 대한 연산 결과는 다음과 같다.

$$Q'\cdot Q\cdot Q'' = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0
\end{bmatrix}$$

위 결과는 32비트 환경에서 Q를 연산하는 방법과 크게 다르지 않으므로, 효율성에 크게 영향을 끼치지 않을 것으로 생각된다. $Q'\cdot Q\cdot Q''$에 대한 64비트 구현은 다음과 같이 나타낼 수 있다.

$$Q'\cdot Q\cdot Q''\cdot \begin{pmatrix} X\\ Y \end{pmatrix} =$$

$$\left[\begin{array}{l}(X \wedge 0xffffffff00000000) \oplus (((X \gg 16) \wedge \\ 0xffff) \oplus ((X \ll 16) \wedge 0xffff0000)) \\ \\ ((Y \ll 8) \wedge 0xff00ff0000ff0000) \oplus ((Y \gg \\ 8) \wedge 0xff00ff0000ff00) \oplus ((Y \gg 24) \wedge 0xff) \\ \oplus ((Y \ll 24) \wedge 0xff000000)\end{array}\right]$$

3. 64비트 환경에서의 ARIA 구현 결과

블록암호 알고리즘 ARIA는 8비트 입/출력을 가지는 S-box와 16x16 이진행렬을 사용하고 있는데, 이는 8비트 연산에 해당한다. 그러므로 ARIA의 자체 논리로는 64비트에 최적으로 대응하지 못한다. 따라서 본 논문에서는 워드를 64비트로 정의하고 위에서 제안한 방법을 이용하여 ARIA를 64비트 프로세서 환경에 적합하게 구현하였다.

3.1 성능평가 환경 및 컴파일러

본 논문에서는 효율성 분석을 위하여 MS(Microsoft) Windows 2003 Server x64를 사용하였다. Window 시스템의 중 Windows 2003 Server x64가 현재 유일한 64비트 OS이기 때문이다.

3.1.1 성능평가 환경

64비트 응용프로그램을 구동하기 위해서는 64비트 프로세서를 포함한 시스템과 64비트 OS 그리고 64비트를 지원하는 컴파일러가 필요하다. 소스 코드는 ANSI C를 사용하였으며, Endian의 차이가 있기 때문에 UNIX 시스템에서는 동작하지 않는다. 본 논문을 위한 성능평가 환경은 아래 [표 3]과 같다.

[표 3] 성능평가 환경

	O/S	CPU	RAM
P1	Window 2003 server IA64 edition	Intel Itanium 1.7 GHz (IA64)	4GB
P2	Window 2003 server x64 edition	Intel Pentium4 (Prescott) 3.0 GHz (EM64T)	1GB
P3	Window 2003 server x64 edition	Intel Xeon 2.0 GHz (EM64T)	1GB

3.1.2 컴파일러

범용으로 사용할 수 있는 상위레벨 언어인 C의 컴파일러는 다양하게 존재한다. 그러나 64비트 시스템의 과도기에 있는 현재, 64비트를 완벽하게 지원하는 컴파일러는 아래 4종류로 압축할 수 있다.

- MS Visual Studio 2005 Release Candidate Version
- MS Visual Studio .NET 2003
- GNU C Compiler Release 2005.9.28
- Intel C++ Compiler 9.0

위에 소개된 컴파일러 중 GNU C Compiler의 경우 무료로 사용할 수 있는 장점이 있지만, 64비트 연산에서 오류가 발생한다는 지적이 있었으며, MS Visual Studio .NET 2003의 경우 MS Visual Studio 2005에서 해당 기술을 적용했기 때문에 본 논문에서는 MS Visual Studio 2005[7]와 Intel C++ Compiler 9.0[8]을 사용하여 테스트했다.

[표 4] 컴파일러

	컴파일러 종류
C1	Visual Studio 2005 beta2 with PSDK (MS Windows 2003 Platform Software Development Toolkit)
C2	Intel 64 bit C compiler ver 9.0 for 64-bit

MS Visual Studio 2005는 아직 개발단계이기 때문에 GUI(Graphic User Interface) 상에서 64비트 컴파일을 완벽하게 지원하지 못한다. 따라서 Windows 2003 Server PSDK의 설치 후, 플랫폼별 컴파일을 하여 테스트 했다. 컴파일러마다 약간의 차이는 있지만, 기존의 변수 선언 형태는 동일하게 존재한다. 즉, 기존 32비트 이하의 크기를 가지는 데이터 형은 변하지 않고 64비트를 지원하는 데이터 형이 추가된 것이다. 64비트 컴파일러 상의 정수형 데이터 종류는 아래 [표 5]와 같다.

_int32와 _int64는 32비트 Visual Studio에서도 지원하는 데이터 형태이지만 _int64의 경우 소프트웨어적으로 두 개의 정수형 변수를 이어주었기 때문에, 32비트 Visual Studio에서의 속도는 정수형 변수 두 개를 사용하는 것 보다 연산 속도가 느리다.

〔표 5〕 64비트 컴파일러의 정수형 데이터

Data Type	Range(bit)	지원 컴파일러
(unsigned) int	32	*MSVS, Intel
(unsigned) char	8	MSVS, Intel
long	32	MSVS, Intel
long long	64	MSVS, Intel
_int32	32	MSVS
_int64	64	MSVS
DWORD64	64	MSVS, Intel

* MSVS : MS Visual Studio 2005, Intel : Intel C++ Compiler 9.0

그러나 64비트 컴파일러에서는 하나의 정수로 인식하기 때문에 하나의 정수를 처리하는 효율성을 가진다.

3.2 ARIA 64비트 구현 결과

각각의 성능평가 환경 및 컴파일러에 따른 실험 결과는 다음과 같다. ARIA는 그 구조상 암호화와 복호화의 연산 과정이 동일하며, 단지 암호화의 키 스케줄과 복호화의 키 스케줄이 상이한 형태이다.

효율성의 측정은 각 컴파일러의 Retail mode와 Release mode에서 이루어 졌으며, 컴파일 옵션을 선택하지 않은 경우와 옵션에서 속도 최적화(Maximize Speed) 옵션(/O2)을 선택한 경우에 대해 분석했다. 아래 〔표 6〕은 효율성 분석을 위해 사용된 컴파일러의 컴파일 명령을 나타내며, MS Visual Studio 2005의 경우 "bufferoverflowU.lib"라는 라이브러리를 같이 링크해주어야 한다. "bufferoverflowU.lib"는 MS PSDK에 포함된 라이브러리로써, MS Windows 내에서 실행되는 응용프로그램이 버퍼오버런(buffer over run) 오류를 범하지 않도록 방지해주는 라이브러리이다. 버퍼오버런은 MS Windows에 치명적인 오류를 가져오기 때문에, 악성코드 등에서 많이 사용되며, MS Visual Studio에서는 컴파일되는 응용프로그램이 MS Windows에 치명적 영향을 끼치지 않도록 "bufferoverflowU.lib"이 링크되지 않는 경우 "보안성 검정" 오류를 출력하며 컴파일 되지 않도록 설정하였다.

〔표 6〕 컴파일러의 컴파일 명령

컴파일러	옵션	명령줄
MS Visual Studio 2005	최적화 없음	cl.exe aria.c bufferover-flowU.lib
	속도 최적화	cl.exe aria.c bufferover-flowU.lib /O2
Intel C++ Compiler	최적화 없음	icl.exe aria.c
	속도 최적화	icl.exe aria.c /O2

아래 〔표 7〕는 64비트 ARIA의 각 플랫폼 및 컴파일러별 속도를 측정하기 위한 코드이며, 측정된 속도의 단위는 Mbps(Megabyte per second, 단위초당 메가바이트)이다. 속도를 측정하기 위하여 아래 〔표 7〕과 같이 C 내장 함수인 clock() 함수를 사용하여 수행속도를 계산하였다.

〔표 7〕 속도 측정을 위한 Pseudo Code

```
1. 측정시작;
2. FOR I = 1 : 수행회수
3. 측정 위한 함수();
4. END FOR
5. 측정종료;
6. 수행시간 = 측정종료시간 - 측정시작시간;
7. 출력(128×수행회수/수행시간/1024/1024)
```

- 최적화된 컴파일을 하지 않은 경우

〔표 8〕 Intel Itanium 1.7 GHz (IA64) 환경에서의 결과 (단위 : Mbps)

P1	C1	C2
Encryption Key Schedule	61.035	610.352
Decryption Key Schedule	14.031	71.806
Encryption only	93.900	203.451
Encryption with Encryption Key Schedule	36.991	174.386
Decryption with Decryption Key Schedule	14.031	71.806

〔표 9〕 Intel Pentium4 3.0 GHz (EM64T) 환경에서의 결과 (단위 : Mbps)

P2	C1	C2
Encryption Key Schedule	259.724	762.939
Decryption Key Schedule	84.931	156.500
Encryption only	131.258	239.724
Encryption with Encryption Key Schedule	86.575	156.500
Decryption with Decryption Key Schedule	39.125	86.575

〔표 10〕 Intel Xeon 2.0 GHz (EM64T) 환경에서의 결과 (단위 : Mbps)

P3	C1	C2
Encryption Key Schedule	629.302	103.724
Decryption Key Schedule	163.602	45.692
Encryption only	266.145	128.921
Encryption with Encryption Key Schedule	200.406	49.163
Decryption with Decryption Key Schedule	91.381	34.108

• 속도 최적화된 컴파일을 한 경우

〔표 11〕 Intel Itanium 1.7 GHz (IA64) 환경에서의 결과(속도 최적화) (단위 : Mbps)

P1	C1	C2
Encryption Key Schedule	610.352	610.352
Decryption Key Schedule	152.588	71.806
Encryption only	244.141	203.451
Encryption with Encryption Key Schedule	174.386	174.386
Decryption with Decryption Key Schedule	71.806	71.806

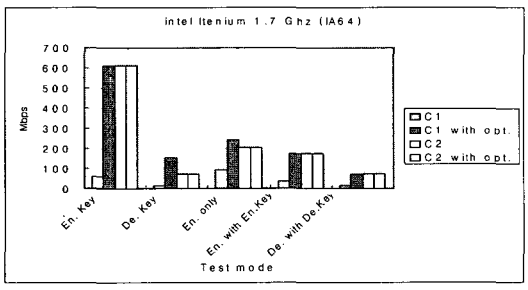
〔표 12〕 Intel Pentium4 3.0 GHz (EM64T) 환경에서의 결과 (단위 : Mbps)

P2	C1	C2
Encryption Key Schedule	762.939	762.959
Decryption Key Schedule	193.762	193.762
Encryption only	259.724	259.724
Encryption with Encryption Key Schedule	196.888	196.888
Decryption with Decryption Key Schedule	87.250	87.193

〔표 13〕 Intel Xeon 2.0 GHz (EM64T) 환경에서의 결과(속도 최적화) (단위 : Mbps)

P3	C1	C2
Encryption Key Schedule	629.302	629.302
Decryption Key Schedule	169.355	155.192
Encryption only	266.145	266.145
Encryption with Encryption Key Schedule	201.408	188.157
Decryption with Decryption Key Schedule	91.382	91.382

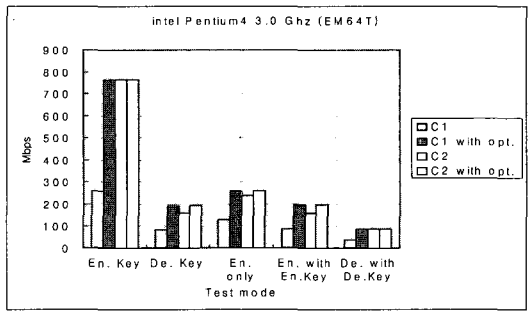
〔표 8〕과 〔표 11〕로부터 Intel Itanium 1.7 GHz (IA64) 환경에서 속도 최적화를 적용한 경우와 그렇지 않은 경우는 다음 그래프와 같은 차이를 보인다.



〔그림 2〕 Intel Itanium 1.7 Ghz 환경에서의 결과 비교

Intel 컴파일러의 경우 속도 최적화 옵션이 별다른 영향을 주지 못하지만, Visual studio에 내장된 컴파일러의 경우 속도가 많이 향상되었음을 알 수 있다.

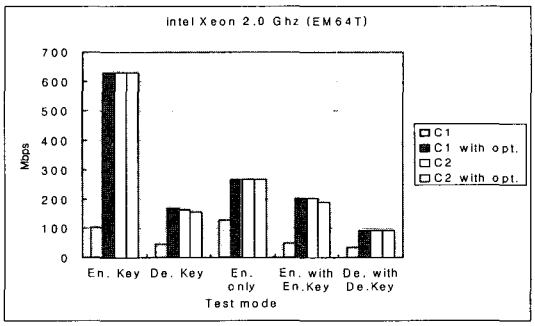
〔표 9〕과 〔표 12〕로부터 Intel Pentium4 3.0 GHz (EM64T) 환경에서 속도 최적화를 적용한 경우와 그렇지 않은 경우는 다음 그래프와 같은 차이를 보인다.



〔그림 3〕 Intel Pentium4 3.0 Ghz(EM64T) 환경에서의 결과 비교

Intel 컴파일러의 경우 속도 최적화 옵션에서 약간의 속도 증가 효과를 얻을 수 있었고, Visual studio에 내장된 컴파일러의 경우 속도가 많이 향상되었음을 알 수 있다.

〔표 10〕과 〔표 13〕으로부터 Intel Xeon 2.0 GHz (EM64T) 환경에서 속도 최적화를 적용한 경우와 그렇지 않은 경우는 다음 그래프와 같다.



〔그림 4〕 Intel Xeon 2.0 Ghz 환경에서의 결과 비교

Intel 컴파일러의 경우 속도 최적화 옵션에서 약간의 속도 증가 효과를 얻을 수 있었고, Visual studio에 내장된 컴파일러의 경우 속도가 많이 향상되었음을 알 수 있다.

3.3 ARIA 64비트 구현에 따른 연산 수 분석

본 절에서는 32비트 변수와 64비트 변수를 사용한 경우, 구현 논리 및 연산의 수에 관해 살펴본다. 32비트 구현은 기본 연산이 모두 32비트 변수로 이루어지며, 64비트 구현은 64비트 변수로 이루어진다고 가정한다.

3.3.1 키 삽입 단계

32비트 구현은 4개의 32비트 변수의 ⊕로 구성된다.
64비트 구현은 2개의 64비트 변수의 ⊕로 구성된다.

3.3.2 S-box 계층

32, 64비트 환경 모두 기본적인 논리는 8비트 입출력의 S-box 연산을 기본으로 하기 때문에, 각 입력을 8비트 단위로 변환해야 하며, 출력은 확산계층의 일부 논리와 결합하여 각 8비트 입력에 해당하는 32비트 혹은 64비트 출력들의 ⊕로 구성된다. 예를 들어 ARIA의 구현은 다음과 같다.

- 32비트 구현의 경우

확산계층의 일부와 결합하여 다음과 같은 연산이 이루어지므로, 32비트 원소들로 된 4개의 테이블을 생성하여 32 비트 출력에 대응할 수 있다.

$$\begin{bmatrix} 0&1&1&1\\ 1&0&1&1\\ 1&1&0&1\\ 1&1&1&0 \end{bmatrix} \cdot \begin{bmatrix} S_1\\ S_2\\ S_1^{-1}\\ S_2^{-1} \end{bmatrix} = \begin{bmatrix} 0\\ S_1\\ S_1\\ S_1 \end{bmatrix} \oplus \begin{bmatrix} S_2\\ 0\\ S_2\\ S_2 \end{bmatrix} \oplus \begin{bmatrix} S_1^{-1}\\ S_1^{-1}\\ 0\\ S_1^{-1} \end{bmatrix} \oplus \begin{bmatrix} S_2^{-1}\\ S_2^{-1}\\ S_2^{-1}\\ 0 \end{bmatrix}$$

이 때, 연산의 수는 32비트 입력을 S-box의 입력으로 변환하는 과정에서 각 32비트 당 3회의 쉬프트 연산이 필요하며, 총 12회의 쉬프트 연산이 사용된다. 또한 32비트 출력을 내기 위해 위의 논리로 테이블을 생성하므로, 각 32비트 당 3회의 ⊕가 필요하며, 총 12회의 ⊕가 사용된다.

- 64비트 구현의 경우

확산계층의 일부와 결합하여 다음과 같은 연산이 이루어지므로, 64비트 원소들로 된 8개의 테이블을 생성하여 64 비트 출력에 대응할 수 있다.

$$\begin{bmatrix} 0&1&1&1&0&0&0&0\\ 1&0&1&1&0&0&0&0\\ 1&1&0&1&0&0&0&0\\ 1&1&1&0&0&0&0&0\\ 0&0&0&0&0&1&1&1\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&1&1&0&1\\ 0&0&0&0&1&1&1&0 \end{bmatrix} \cdot \begin{bmatrix} S_1\\ S_2\\ S_1^{-1}\\ S_2^{-1}\\ S_1\\ S_2\\ S_1^{-1}\\ S_2^{-1} \end{bmatrix}$$

$$= \begin{bmatrix} 0\\ S_1\\ S_1\\ S_1\\ 0\\ 0\\ 0\\ 0 \end{bmatrix} \oplus \begin{bmatrix} S_2\\ 0\\ S_2\\ S_2\\ 0\\ 0\\ 0\\ 0 \end{bmatrix} \oplus \begin{bmatrix} S_1^{-1}\\ S_1^{-1}\\ 0\\ S_1^{-1}\\ 0\\ 0\\ 0\\ 0 \end{bmatrix} \oplus \begin{bmatrix} S_2^{-1}\\ S_2^{-1}\\ S_2^{-1}\\ 0\\ 0\\ 0\\ 0\\ 0 \end{bmatrix}$$

$$\oplus \begin{bmatrix} 0\\ 0\\ 0\\ 0\\ 0\\ S_1\\ S_1\\ S_1 \end{bmatrix} \oplus \begin{bmatrix} 0\\ 0\\ 0\\ 0\\ S_2\\ 0\\ S_2\\ S_2 \end{bmatrix} \oplus \begin{bmatrix} 0\\ 0\\ 0\\ 0\\ S_1^{-1}\\ S_1^{-1}\\ 0\\ S_1^{-1} \end{bmatrix} \oplus \begin{bmatrix} 0\\ 0\\ 0\\ 0\\ S_2^{-1}\\ S_2^{-1}\\ S_2^{-1}\\ 0 \end{bmatrix}$$

이 때, 연산의 수는 64비트 입력을 S-box의 입력으로 변환하는 과정에서 각 64비트 당 7회의 쉬프트 연산이 필요하며, 총 14회의 쉬프트 연산이 사용된다. 또한 64비트 출력을 내기 위해 위의 논리로 테이블을 생성하므로, 각 64 비트 당 7회의 ⊕가 필요하며, 총 14회의 ⊕가 사용된다.

3.3.3 확산계층

S-box 계층의 출력 $S(x)$를 입력으로 하는 확산계층의 연산은 다음과 같이 이루어지며,

$$A(x) = L \cdot Q \cdot L \cdot P \cdot M \cdot S(x)$$

이때, S-box 계층의 구현에서 이미 사용된 부분인 $P \cdot M$을 제외한 나머지 부분의 연산은 다음과 같다. 여기서 L, Q는 다음과 같다.

$$L = \begin{bmatrix} 1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0\\ 1&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0\\ 0&1&0&0&0&0&0&0&0&1&0&0&0&1&0&0\\ 0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&0\\ 0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1\\ 1&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0\\ 0&1&0&0&0&1&0&0&0&0&0&0&0&1&0&0\\ 0&0&1&0&0&0&1&0&0&0&0&0&0&0&1&0\\ 0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&1\\ 0&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0&0&1&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0&0&0&1&0&0&0&1&0\\ 0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&1 \end{bmatrix}$$

$$Q = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \end{bmatrix}$$

• 32비트 구현의 경우

위의 16x16 이진행렬 L은 다음과 같이 표현된다.

$$\begin{bmatrix} I_{4\times4} & I_{4\times4} & I_{4\times4} & 0_{4\times4} \\ I_{4\times4} & 0_{4\times4} & I_{4\times4} & I_{4\times4} \\ I_{4\times4} & I_{4\times4} & 0_{4\times4} & I_{4\times4} \\ 0_{4\times4} & I_{4\times4} & I_{4\times4} & I_{4\times4} \end{bmatrix}$$

이는 32비트 연산에 쉽게 대응되며, 구현은 8회의 ⊕로 이루어진다. Q의 경우, 첫 번째 32비트는 치환 연산이 일어나지 않으며, 나머지의 경우는 5회의 ⊕와 8회의 쉬프트, 4회의 AND 연산으로 구현된다.

• 64비트 구현의 경우

16x16 이진행렬 L은 64비트 변수에 대응하여 다음과 같이 표현된다.

$$\begin{bmatrix} I_{4\times4} & I_{4\times4} & \multicolumn{2}{c}{\multirow{2}{*}{$I_{8\times8}$}} \\ I_{4\times4} & I_{4\times4} & & \\ \multicolumn{2}{c}{\multirow{2}{*}{$I_{8\times8}$}} & I_{4\times4} & I_{4\times4} \\ & & I_{4\times4} & I_{4\times4} \end{bmatrix}$$

이는 64비트 연산에 완벽하게 대응하지 못하며, 32비트 연산을 포함하고 있는 형태이다. 구현은 6회의 ⊕와 2회의 순환회전으로 이루어진다. Q의 경우, 5회의 ⊕와 6회의 쉬프트, 7회의 AND 연산으로 구현된다.

위의 내용을 정리하여, 실제로 구현된 32비트 환경에서 구현된 코드와 64비트 환경에서 구현된 코드와의 연산 개수를 〔표 14〕에 정리하였다. 이는 64비트 변수를 사용하여 ARIA를 구현하였을 경우 연산의 수가 32비트 변수를 이용한 ARIA보다 많음을 보여준다.(128비트 키를 사용한 경우)

〔표 14〕 ARIA의 연산 수 비교

연산	32비트 구현			64비트 구현		
	암/복호화	암호화 키스케줄	복호화 키스케줄	암/복호화	암호화 키스케줄	복호화 키스케줄
⊕	427	181	544	381	131	472
테이블 참조	192	48	400	192	48	400
쉬프트	244	60	280	338	84	392
순환회전	-	13*	13*	-	13**	13**
AND	44	12	56	77	21	98

* : 128비트 워드를 32비트 워드 4개로 나눴을 때, 순환회전 횟수를 의미

** : 128비트 워드를 64비트 워드 2개로 나눴을 때, 순환회전 횟수를 의미

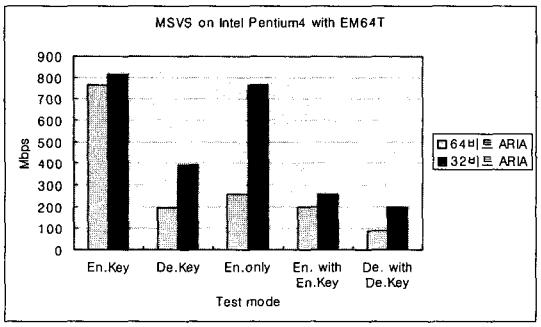
3.4 ARIA 64비트 구현 결과 분석

결과 분석을 위하여 64비트 ARIA와 같은 환경에서 32비트 ARIA를 구현하여 속도 비교를 하였다. 단 Intel Xeon 2.0Ghz는 제외하였으며, Intel c++ compiler와 Visual Studio 간의 컴파일 옵션의 차이 때문에 속도 최적화(Maximize Speed) 옵션을 선택한 경우에 대해서만 비교하였다. 또한 컴파일러의 동일성을 제공하기 위해 32비트 ARIA도 64비트 OS에서 컴파일 하였다. 아래 〔표 15〕, 〔표 16〕은 각 플랫폼에서 32비트 ARIA의 속도를 나타내고 있으며, 〔그림 5〕, 〔그림 6〕, 〔그림 7〕와 〔그림 8〕은 각각 Intel Pentium4 with EM64T와 Intel Itanium CPU에서 32비트 및 64비트 ARIA의 속도를 컴파일러 별로 비교하였다.

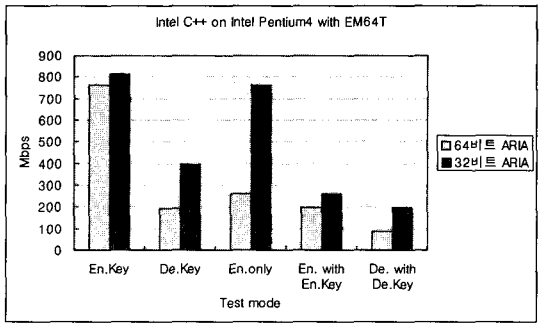
• 최적화된 컴파일을 하지 않은 경우

〔표 15〕 32비트 ARIA의 효율성 분석 결과

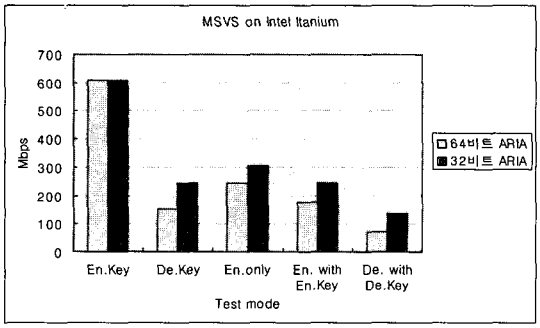
		Enc.Key	Dec.Key	Crypt. Only	Crypt. Enc	Crypt. Dec
P1	C1	29.773	14.707	55.487	19.376	11.626
	C2	610.352	244.141	305.176	203.451	135.634
P2	C1	156.500	77.752	131.258	70.971	46.064
	C2	813.802	393.775	762.939	259.724	196.888



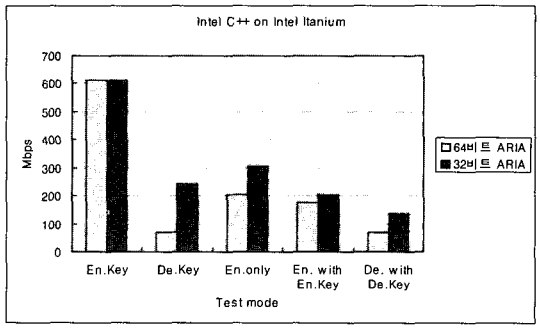
〔그림 5〕 컴파일러별 ARIA 32비트/64비트 속도 비교



〔그림 6〕 컴파일러별 ARIA 32비트/64비트 속도 비교



〔그림 7〕 컴파일러별 ARIA 32비트/64비트 속도 비교



〔그림 8〕 컴파일러별 ARIA 32비트/64비트 속도 비교

- 속도 최적화된 컴파일을 한 경우

〔표 16〕 32비트 ARIA의 효율성 분석 결과

		Enc.Key	Dec.Key	Crypt. Only	Crypt. Enc	Crypt. Dec
P1	C1	610.352	244.141	305.176	244.141	135.634
	C2	610.352	244.141	305.176	203.451	135.634
P2	C1	813.802	393.775	762.939	259.724	196.888
	C2	813.802	393.775	762.939	259.724	196.888

V. 결 론

위의 결과에 의하면 본 논문에서 사용된 플랫폼에서 64비트 ARIA는 32비트 ARIA보다 속도 면에서 효율성의 증가를 기대하기가 어렵다. 이와 같은 결과가 도출된 이유로는 다음 두 가지를 생각할 수 있다.

- 64비트 ARIA의 연산수가 32비트 ARIA의 연산수 보다 많다.
- 현재 사용되는 64비트 프로세서는 시스템의 각 장치와의 호환성을 위해 외부 데이터 버스 등이 32비트로 설계되어 있다. 따라서 64비트 입력을 받더라도 32비트로 나누어 데이터를 받아들이기 때문에, 데이터 병목현상에 의해 속도의 효율성이 떨어진다. 또한 64비트 프로세서의 개발 목적이 속도의 향상에 있는 것이 아니라, 대용량의 메모리 관리를 통한 대용량 데이터의 처리이기 때문에 단순 속도 향상 효과는 크지 않다.

위의 두 가지 이유에 의해 64비트 프로세서에 적합하게 구현된 ARIA는 32비트 프로세서에 적합하게 구현된 ARIA에 대하여 속도 효율성의 증가효과를 기대하기 어려운 것으로 판단된다.

결국 IA64나 x64 with EM64T간의 기본 연산에서의 차이는 없으며, 기본적인 연산은 프로세서의 클럭 속도에 의존하고 있는 것으로 판단된다.

또한 64비트 ARIA에서 사용되는 연산이 평문 혹은 암호문의 입출력을 제외하면 32비트 워드 혹은 바이트단위의 연산이기 때문에, 64비트만의 고유 연산을 사용하고 있지 않다. 따라서 32비트와 64비트간의 데이터 변화과정과 32비트 이하의 연산과정 등이 속도 저하에 영향을 주고 있다.

본 논문에서 확인한 바와 같이 ANSI C 언어의 논리로 만들어진 32비트 ARIA와 64비트 ARIA의 경우 현재의 64비트 프로세서 환경에서는 32비트에 적합하게 구현된 ARIA가 더 효율적으로 동작하는 경우가 발생한다. 따라서 64비트 프로세서의 구조에 좀 더 최적화된 암호알고리즘 설계와 구현방안의 연구가 필요하다.

참 고 문 헌

[1] Microsoft, "Microsoft x64 Conference Proceeding", *정보보호학회논문지*, 24(6), June 1997.

[2] 국가보안기술연구소, "ARIA 알고리즘 명세서 (ARIA v. 1.0)", 2004

[3] A. John, R. Peter, "Electric Communication Development", *Communications of the ACM*, 40, pp. 71-79, May 1997.

[4] 이청년, 이청녀, *암호의 발전에 관한 연구*, 한국출판사, pp.11-21, 1999

[5] Intel, "Intel Extended Memory 64 Technology" *http://www.intel.com/technology/64bitextensions/index.htm.*

[6] Intel, "Intel Itatium2 Processor" *http://www.intel.com/products/processor/itanium2/index.htm.*

[7] Microsoft, "Microsoft Visual Studio 2005", *http://msdn.microsoft.com/vstudio.*

[8] Intel, "Intel C++ Compiler 9.0", *http://www.intel.com.*

[9] Microsoft, "Microsoft Windows Server 2003 SP1 PSDK", *http://www.microsoft.com.*

〈著 者 紹 介〉

장 환 석 (Hwan Seok Jang)
학생회원

2002년 2월 : 한양대학교 수학과 이학사
2004년 2월 : 한양대학교 대학원 수학과 이학석사
2004년 3월~현재 : 한양대학교 대학원 수학과 박사과정
2004년 11월~2월 : 한국정보보호진흥원 위촉연구원
관심분야 : 암호학, 정보보호

이 호 정 (Lee, Ho Jung)
학생회원

2002년 2월 : 한양대학교 수학과 이학사
2004년 2월 : 한양대학교 대학원 수학과 이학석사
2004년 3월~현재 : 한양대학교 대학원 수학과 박사과정
관심분야 : 암호학, 컴퓨터포렌식, 정보보호

구 본 욱 (Bon Wook Koo)
학생회원

2001년 2월 : 한양대학교 수학과 이학사
2003년 2월 : 한양대학교 대학원 수학과 이학석사
2004년 3월~현재 : 한양대학교 대학원 수학과 박사과정
관심분야 : 암호학, 정보보호

송 정 환 (Jung Hwan Song)
정회원

1984년 2월 : 한양대학교 수학과 이학사
1989년 5월 : Syracuse University 수학과 이학석사
1993년 5월 : Rensselaer Polytechnic Institute 수학과 이학박사
1999년 3월~현재 : 한양대학교 수학과 부교수
관심분야 : 암호학, 수리계획법, 최적론