情報保護學會論文誌
第18卷 第5號, 2008. 10

안전한 센서 네트워크를 위한 스트림 암호의 성능 비교 분석*

윤 민[1†], 나 형 준[2], 이 문 규[1‡], 박 근 수[3]
[1]인하대학교 컴퓨터정보공학부, [2]티맥스 소프트, [3]서울대학교 컴퓨터공학부

Performance Analysis and Comparison of Stream Ciphers for Secure Sensor Networks*

Min Yun[1†], Hyoung Jun Na[2], Mun-Kyu Lee[1‡], Kunsoo Park[3]
[1]School of Computer and Information Engineering, Inha University,
[2]TmaxSoft, [3]School of Computer Science and Engineering, Seoul National University

요 약

무선 센서 네트워크는 센서 노드 또는 모트(mote)라 불리는 소형 장치들로 이루어진 무선 네트워크이다. 최근 센서 네트워크에 대한 연구가 활발한 가운데 센서 네트워크에서의 보안에 관한 연구 또한 활발히 진행되고 있다. 센서 노드 및 센서 네트워크 상의 정보를 안전하게 저장, 전송하기 위해서는 암호 알고리즘의 구현이 필요하며, 이 암호 알고리즘들은 센서 노드의 한정된 자원을 효과적으로 활용할 수 있도록 효율적인 구현이 필수적이다. 센서 노드 상에서 이용될 수 있는 암호로는 TinyECC 등의 공개키 암호와 AES와 같은 표준 블록 암호가 있으나, 스트림 암호는 최근에서야 eSTREAM 프로젝트에서 표준화가 완료되어 아직 센서 노드상에서 사용 가능성이 명확하지 않은 실정이다.

이에 본 논문에서는 eSTREAM의 2단계와 3단계에 채택되었던 10개 소프트웨어 기반 암호들 중 9개의 암호들을 MicaZ 모트 상에 구현하여 성능을 비교하고, 특히 최종적으로 eSTREAM에 채택된 SOSEMANUK, Salsa20, Rabbit을 포함한 6개 암호에 대해서는 MicaZ에 적합하도록 최적화하였다. 또한 참조 구현으로써 하드웨어용 스트림 암호 및 AES-CFB에 대한 실험 결과도 제시한다. 본 논문의 실험에 따르면, 대부분의 스트림 암호가 약 31Kbps - 406Kbps의 암호화 성능을 보임으로써 센서 노드에서 사용하기에 큰 무리가 없음을 확인할 수 있었다. 특히 최종적으로 채택된 SOSEMANUK, Salsa20, Rabbit의 경우 센서 노드에 적합한 128바이트 크기의 작은 패킷의 암호화에서 각각 406Kbps, 176Kbps, 121Kbps의 속도를 보여주고, 70KB, 14KB, 22KB의 ROM 및 2811B, 799B, 755B의 RAM을 사용함으로써, 106Kbps의 속도를 보여준 소프트웨어 기반 AES에 비해 우수한 성능을 보임을 알 수 있었다.

ABSTRACT

A Wireless Sensor Network (WSN for short) is a wireless network consisting of distributed small devices which are called sensor nodes or motes. Recently, there has been an extensive research on WSN and also on its security. For secure storage and secure transmission of the sensed information, sensor nodes should be equipped with cryptographic algorithms. Moreover, these algorithms should be efficiently implemented since sensor nodes are highly resource-constrained devices. There are already some existing algorithms applicable to sensor nodes, including public key ciphers such as TinyECC and standard block ciphers such as AES. Stream ciphers, however, are still to be analyzed, since they were only recently standardized in the eSTREAM project.

In this paper, we implement over the MicaZ platform nine software-based stream ciphers out of the ten in the second and final phases of the eSTREAM project, and we evaluate their performance. Especially, we apply several optimization techniques to six ciphers including SOSEMANUK, Salsa20 and Rabbit, which have survived after the final phase of the eSTREAM project. We also present the implementation results of hardware-oriented stream ciphers and AES-CFB for reference. According to our experiment, the encryption speeds of these software-based stream ciphers are in the range of 31-406Kbps, thus most of these ciphers are fairly acceptable for sensor nodes. In particular, the survivors, SOSEMANUK, Salsa20 and Rabbit, show the throughputs of 406Kbps, 176Kbps and 121Kbps using 70KB, 14KB and 22KB of ROM and 2811B, 799B and 755B of RAM, respectively. From the viewpoint of encryption speed, the performances of these ciphers are much better than that of the software-based AES, which shows the speed of 106Kbps.

Keywords : wireless sensor network, cryptography, stream cipher, MicaZ, eSTREAM

Ⅰ. 서 론

무선 센서 네트워크는 다수의 센서 노드들로 이루어진 네트워크이다. 이 센서 노드들은 물리적으로 작고, 서로 간에 무선으로 통신을 하며, 네트워크 토폴로지의 사전 정보 없이 배치되는 특성이 있다. 센서 노드들은 물리적인 크기에 대한 제한 때문에 저장 공간과 계산 능력, 에너지 공급, 통신 대역폭에 제한을 받는다. 이러한 센서 네트워크는 센서 노드를 통해 다양한 정보를 감지하고 감지된 정보를 처리하여 우리의 삶을 자동화시키고 편리함을 제공할 수 있을 것으로 기대되나, 일상생활에 의존도가 높아질수록 이로 인한 위험성 또한 높아지기 때문에 센서 네트워크를 통해 제공되는 정보들을 신뢰하고 동시에 개인의 프라이버시를 보장할 수 있는 방안이 필요하다. 즉 현실적이고 안전한 센서 네트워크를 구현하기 위해서는 감지된 정보를 안전하게 처리하고 관리할 수 있는 암호 알고리즘이 필요하다.

암호 알고리즘은 크게 공개키 암호와 대칭키 암호로 나눌 수 있다. 센서 네트워크에서 사용할 수 있는 공개키 암호로 TinyOS[1] 위에서 작동되는 타원곡선 암호인 TinyECC[2]가 이미 개발되어 사용되고 있다. 대칭키 암호는 다시 블록 암호와 스트림 암호로 나눌 수 있고 이 중 블록 암호로는 ZigBee[3]의 표준으로 AES (Advanced Encryption Standard)가 사용되고 있다. 그러나 스트림 암호의 경우에는 표준화가 최근에서야 종료됨으로써, 센서노드 상의 성능 분석이 필요한 실정이다.

이미 관련연구로서 ATmega128L에서 스트림 암호들의 속도와 ROM 및 RAM 메모리공간 사용을 분석한 결과가 있으나[4], 이는 실제 센서 노드 장치가 아닌 AVR Studio 4에서 시뮬레이션을 한 결과였고, 아직까지 실제로 센서 노드상에서 구현하여 성능을 분석한 사례는 없었다. 본 논문에서는 eSTREAM[5]의 2단계와 3단계에서 'focus cipher'로 선택되었던 10개의 소프트웨어 암호들 중 HC-128을 제외한 9개 암호들을 센서 노드상에서 구현하였고, 이중 최종적으로 채택된 SOSEMANUK, Salsa20, Rabbit[1]을 포함한 6개 암호에 대해서는 센서 노드에 적합하도록 추가적인 최적화를 수행하여 성능을 비교하였다.

본 논문의 실험에 따르면 대부분의 스트림 암호가 약 31Kbps - 406Kbps의 암호화 성능을 보임으로써 센서 노드에서 사용하기 적합함을 확인할 수 있었다. 특히 최종적으로 채택된 SOSEMANUK, Salsa20, Rabbit의 경우 적절한 최적화 과정을 거친 후에는 센서 노드에 적합한 128바이트 크기의 작은 패킷의 암호화에서 각각 406Kbps, 176Kbps, 121Kbps의 속도를 보여주고 70KB, 14KB, 22KB의 ROM 및 2811B, 799B, 755B의 RAM 메모리 공간을 사용함으로써, 유사한 정도의 최적화를 적용할 때 106Kbps의 속도를 보여준 소프트웨어 기반 AES에 비해 우수한 성능을 보임을 알 수 있었다.

접수일 : 2008년 1월 4일; 수정일 : 2008년 6월 21일;
채택일 : 2008년 8월 12일
* 본 연구는 지식경제부 및 정보통신연구진흥원의 IT핵심기술개발사업의 일환으로 수행하였음. [2008-F-045-01, 장애인 및 고령자를 위한 Digital Guardian 기술개발]
† 주저자, chiuyun@inhaian.ac.kr
‡ 교신저자, mklee@inha.ac.kr

1) HC-128 역시 eSTREAM에 최종적으로 채택되었으나, 내부 상태 공간의 크기 문제로 실험 대상에서 제외하였다.

Ⅱ. eSTREAM 프로젝트

eSTREAM[5]은 EU ECRYPT network에서 조직한 스트림 암호 선정 프로젝트로서, NESSIE 프로젝트[6]에 제출된 6개 스트림 암호의 실패로 인하여 시작되었다. 2004년 11월 처음 후보를 받았고, 2008월 4월에 종료되었다. 이 프로젝트는 3단계로 나누어져 있고 소프트웨어용과 하드웨어용에 따라 알맞은 알고리즘을 찾는 것이 목표였다. 공식적으로 2006년 3월 27일에 1단계, 2007년 3월에 2단계가 끝났고 2007년 4월부터 3단계가 시작하여 각 용도에 따라 여러 개의 알고리즘들이 선택되었다.

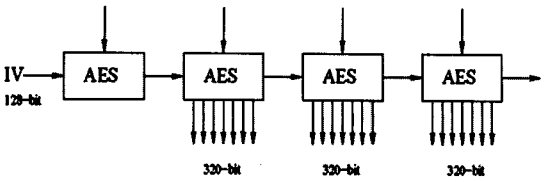
eSTREAM의 2단계와 3단계에서 'focus cipher'로 선택되었던 소프트웨어 스트림 암호에는 HC-128[7], LEX[8], Phelix[9], Py[10], Salsa20[11], SOSEMANUK[12], CryptMT[13], NLS[14], Rabbit[15], Dragon[16]이 있는데, 이 스트림 암호들은 다양한 크기의 키와 초기 벡터(Initial Vector : IV)를 사용하며 적당한 크기의 내부 상태 공간(Internal State)을 유지한다. 암호화나 복호화 과정은 보통 키스트림을 추출하여 평문과 키스트림을 XOR하여 암호문을, 암호문과 키스트림을 XOR하여 평문을 만들어 낸다. 키스트림을 생성하는 과정은 대개 초기화, 내부 상태 업데이트, 키스트림 추출의 3가지 단계로 구성되며, 내부 상태 업데이트와 키스트림 추출의 반복 과정을 통해 필요한 키스트림을 얻어낸다. 초기화는 키 설정(Key setup)과 IV 설정(IV setup)을 통하여 내부 상태 공간을 채우는 과정이다. 내부 상태 업데이트 과정은 비선형 함수나 기타 알고리즘을 이용하여 내부 상태 공간을 업데이트 하는 과정이다. 키스트림 추출 과정은 내부 상태를 이용하여 비선형 필터나 기타 알고리즘을 이용하여 키스트림을 추출하는 과정이다.

이 절에서는 위에서 나열된 10개의 소프트웨어 기반 스트림 암호들 중 본 논문에서 최적화 대상으로 삼았던 6개의 암호에 대하여 간단히 설명한다. 나머지 4개 암호들의 자세한 설명은 [5]에서 확인할 수 있다.

2.1 LEX[8]

LEX의 디자인은 블록암호인 AES에 기초하고 있다. LEX는 128비트의 키와 128비트의 IV를 가지며, 128비트 AES의 CFB(Ciphertext FeedBack)모드를 이용한다.

초기화 과정에서는 128비트의 키에 대해 AES 키 스케줄링을 수행하고 128비트의 IV에 대해서 AES로 한 번 암호화를 하여 이 때 나온 128비트의 암호문을 내부 상태 공간으로 사용한다.



〔그림 1〕 LEX의 초기화와 키스트림 생성

키스트림은 [그림 1]에서 볼 수 있듯이 내부 상태 공간을 초기화하고 AES를 이용하여 추출한다. 128비트 AES의 경우 10라운드로 구성이 되어있는데 LEX에서는 AES의 한 라운드를 내부 상태 업데이트 과정으로 이용한다.

키스트림 추출 과정은 다음과 같이 이루어진다. 내부 상태 공간의 크기가 128비트, 즉 16바이트이고 이를 b0, b1, …, b14, b15의 16개의 바이트로 보고, AES의 홀수 라운드 수행 후에는 내부 상태 공간에서 b0, b8, b2, b10로 32비트를 추출하고 짝수 라운드 수행 후에는 내부 상태 공간에서 b1, b9, b3, b11로 32비트 키스트림을 추출한다. 따라서 AES를 한번 수행하는데 320비트의 키스트림을 추출한다.

2.2 Py[10]

Py는 Rolling Array를 이용한 스트림 암호이다. 최대 256비트의 키와 최대 128비트의 IV를 이용한다.

Rolling Array는 rotate 연산이 반복적으로 적용되는 벡터 형태의 자료구조로 볼 수 있다. 이 외에도 두 가지 기초적인 연산을 추가적으로 사용하는데 하나는 swap 연산이고 다른 하나는 덧셈 연산이다.

이러한 Rolling Array는 연산을 할 때마다 다음과 같이 rotate를 수행 한다.

tmp=S[0], S[0] = S[1], S[1] = S[2], …, S[N-1]=tmp (S[0, …, N-1]은 배열)

구체적으로, swap 연산은 주어진 k에 대해서 swap(S[0], S[k]); rotate(S) 수행을 하고, 덧셈 연산은 주어진 v에 대해서 S[0] += v; rotate(S) 수행을 한다.

이러한 Rolling Array는 회전 연산에 대한 비용이 매우 크기 때문에 실제의 Array보다 큰 공간을 할당하고

```
/* swap and rotate P */
swap(P[0],P[Y[185]& 0xFF]);
rotate(P);

/* Update s */
s+ = Y[P[72]]- Y[P[239]];
s = ROTL32(s,((P[116]+18)& 31));

/* Output 8 bytes (least significant byte first) */
output((ROT32(s,25)⊕ Y[256])+ Y[P[26]]);
output((        s    ⊕ Y[-1])+ Y[P[208]]);

/* Update and rotate Y */
Y[-3] = (ROTL32(s,14)⊕ Y[-3])+ Y[P[153]];
rotate(Y);
```

〔그림 2〕 Py의 키스트림 생성

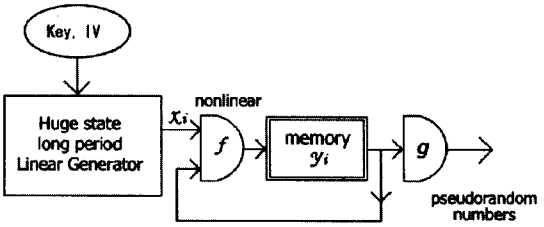
Array의 초기 주소를 증가하는 방식으로 효과적인 연산을 수행할 수 있다. 예를 들어 S[0, ···, N-1]의 배열에 회전 연산을 수행하면 S[N]에 S[0]을 할당하고 S 배열을 S[1, ···, N] 으로 간주하게 되면 실제로 회전하는 연산과 같게 된다. 그러나 실제 배열을 할당할 수 있는 공간은 한정되어있기 때문에 실제 배열이 한계를 넘어서게 되면 다시 시작점으로 되돌리는 연산이 필요로 한다. 따라서 Py의 최적화는 시작점으로 되돌리는 연산을 최대한 줄이는 것이 중요하다.

Py에서는 2개의 Rolling Array를 사용하는데 이들은 서로 간에 영향을 준다. 하나는 P로 256바이트의 값을 순열로 갖고 매 단계마다 swap 연산을 수행하여 업데이트 연산을 한다. 다른 하나는 Y로 32비트씩 260개의 배열 값을 갖고 -3, ···, 256의 범위를 사용한다. Y의 업데이트 연산은 Y를 직접적인 방법과 P를 통한 간접적인 방법으로 접근하여 새로운 값을 업데이트하고 회전하는 것이다.

Py는 이렇게 2개의 Rolling Array P와 Y, 그리고 32비트의 s를 내부 상태 공간으로 유지하고 이를 업데이트하며 키스트림을 추출한다. 각 단계에서 P와 Y는 회전하면서 총 8바이트의 키스트림을 추출한다. s는 P로부터 Y를 간접적으로 접근하여 얻은 2개의 워드 값을 혼합하여 업데이트된다. [그림 2]는 이러한 내부상태 업데이트 및 키스트림 추출 과정을 보여준다.

2.3 CryptMT[13]

CryptMT는 128비트에서 최대 2048비트까지의 128비트의 배수로 이루어진 키와 IV를 이용한다. CryptMT



〔그림 3〕 CryptMT의 키스트림 생성

는 Simple Fast Mersenne Twister (SFMT) 기반의 의사난수발생기와 한 개의 워드 크기의 메모리를 갖는 필터로 구성되어있다. SFMT는 128 * 156비트의 큰 내부 상태 공간을 갖고 매 라운드마다 128비트의 의사난수를 생성한다. SFMT에서 생성된 128비트의 정수는 128비트의 메모리를 갖는 필터를 거쳐서 64비트의 정수를 키스트림으로 이용하게 되는데 실제로는 2라운드마다 128비트 마스크를 이용하여 128비트의 키스트림을 생성한다. [그림 3]은 이러한 과정을 설명하는데, SFMT에서 생성된 128비트의 의사난수 x_i는 128비트의 메모리 y_i를 갖는 필터 f를 거치고 128비트 마스크 g를 거쳐서 키스트림이 된다.

2.4 Salsa20[11]

Salsa20은 256비트 혹은 128비트의 키와 64비트의 IV를 이용한다. Salsa20은 해쉬 함수, 확장 함수, 암호화 함수로 구성이 된다. 내부상태공간은 32비트 워드로 이루어진 4x4 테이블로 총 64바이트의 크기를 갖는다. 내부상태공간의 업데이트는 64바이트의 입력과 64바이트의 출력을 갖는 해쉬 함수를 통해 이루어진다. 해쉬 함수는 Salsa20의 핵심인데, 실제로 Salsa20은 해쉬 함수의 counter 모드를 스트림 암호로 이용한 것으로, 64바이트의 평문 블록에 키와 IV, 블록 넘버를 해쉬한 결과를 XOR 하여 암호화를 한다. 이러한 해쉬 함수는 여러 개의 quarterround로 구성되어 있는데, 각 quarterround는 [그림 4]에서처럼 XOR, 덧셈, rotate 연산으로 구성

if $y = (y_0, y_1, y_2, y_3)$ then
quarterrou nd(y) $= (z_0, z_1, z_2, z_3)$, where

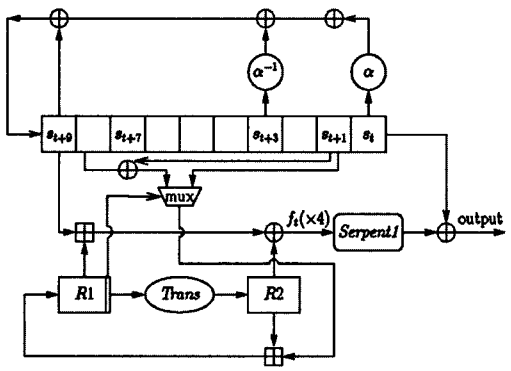
$z_1 = y_1 \oplus ((y_0 + y_3) <<< 7)$.
$z_2 = y_2 \oplus ((z_1 + y_0) <<< 9)$.
$z_3 = y_3 \oplus ((z_2 + z_1) <<< 13)$.
$z_0 = y_0 \oplus ((z_3 + z_2) <<< 18)$.

〔그림 4〕 Salsa20의 quarterround

되어 있다.

2.5 SOSEMANUK[12]

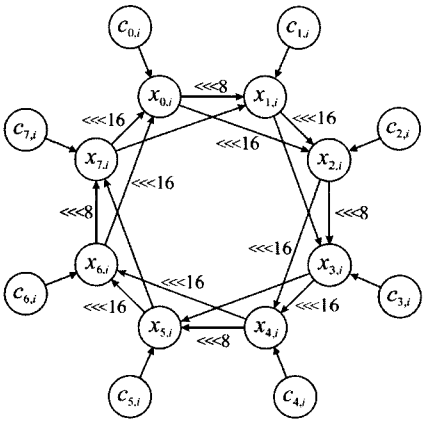
SOSEMANUK은 128비트 또는 256비트의 키와 128비트의 IV를 이용한다. SOSEMANUK은 SNOW 2.0[17] 스트림 암호에 이용된 원리와 SERPENT[18] 블록 암호의 변환들로 구성이 된다. 전체 32라운드로 구성이 되어 있는 SERPENT의 키 추가와 선형 변환을 제외한 단일 라운드 Serpent1을 키스트림 생성에 이용하고 24라운드만을 이용한 Serpent24를 암호화 과정에서 초기화에만 이용한다. SOSEMANUK의 내부상태공간은 Finite State Machine(FSM)과 Linear Feedback Shift Register(LFSR)으로 구성이 되어있다. FSM과 LFSR이 4번 업데이트 될 때마다 FSM의 Serpent1을 통과한 128비트의 출력과 LFSR의 128비트의 출력을 이용하여 128비트의 키스트림을 생성한다. [그림 5]는 이 과정을 보여주는데, FSM은 R1과 R2의 32비트 메모리로 구성되고 32비트의 f_t의 출력을 생성한다. LFSR의 재귀 규칙은 특정 상태 원소들에 유한체 $GF(2^{32})$의 원소 α 또는 α-1를 곱하고 XOR하는 방식으로 구성되어 있다.



〔그림 5〕 SOSEMANUK의 키스트림 생성

2.6 Rabbit[15]

Rabbit은 128비트의 키와 64비트의 IV를 이용한다. Rabbit은 내부 상태 공간으로 8개의 32비트 상태 변수 $x_{0,i}$, ⋯,$x_{7,i}$, 8개의 32비트 counter $c_{0,i}$, ⋯,$c_{7,i}$, 1개의 1비트 counter carry를 이용한다. 매 라운드 마다 내부상태공간을 Next-state Function을 통하여 업데이트를 하고



〔그림 6〕 Rabbit의 Next-state Function

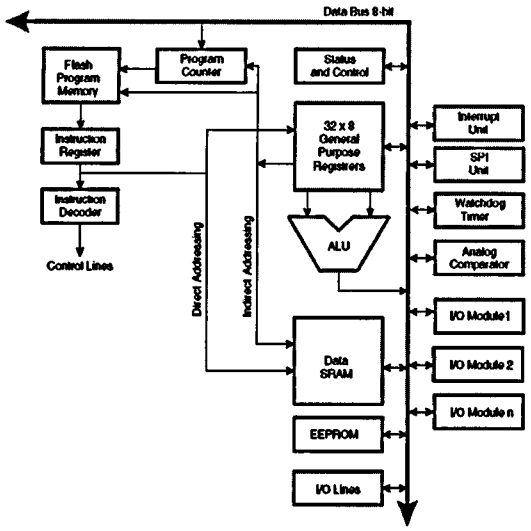
상태 변수들을 정해진 방식에 의해서 128비트의 키스트림을 생성한다.

Ⅲ. eSTREAM 암호의 구현

이 절에서는 위에서 설명한 스트림 암호들을 센서 노드에 구현하는 방법에 대해 설명한다. 구현을 위한 센서 노드로는 8비트 프로세서인 ATmega128L[19]을 장착한 MicaZ 센서 노드를 이용하였다. ATmega128L은 128KB의 In-System Programmable Flash Memory인 프로그램 메모리, 4KB의 EEPROM, 4KB의 SRAM 데이터 메모리로 구성이 되어 있다.

본 논문에서 구현하고자 하는 스트림 암호들은 32비트 일반 PC에 최적화된 암호들이기 때문에 센서 노드에서 구현하기에는 프로그램과 데이터를 저장할 수 있는 공간의 제약이 있다. 그러므로 이 센서 노드에서 프로그램 메모리와 데이터 메모리를 어느 정도 보장할 수 있는지가 중요하다. PC에서 주로 많이 쓰이고 있는 Von Neumann 아키텍쳐 기반의 프로세서에서는 디스크에서 프로그램을 읽어서 메모리에 읽어 오고 그 다음 메모리에서 프로세서로 명령어들이 이동하여 decode되고 실행이 된다. 따라서 동일 메모리를 명령어들과 데이터가 같이 사용하기 때문에 데이터 메모리로 사용할 수 있는 크기는 전체 메모리 용량보다 작아지게 된다.

그러나 ATmega128L에서는 [그림 7]과 같이 명령어들이 SRAM을 거치지 않고 바로 프로그램 메모리에서 Instruction Register를 거쳐 Instruction Decoder로 가기 때문에 4KB SRAM 전체를 데이터를 위한 공간으로 활용할 수 있다.



〔그림 7〕 ATmega128L의 구조

본 논문의 실험 환경은 ATLMega128L을 사용하는 MicaZ 계열의 센서 노드 위에서 널리 사용되는 운영체제인 TinyOS이다. TinyOS는 nesC[20]로 구현되어 있고 모듈 방식으로 구성되어있기 때문에 스트림 암호를 모듈로 구현하여 TinyOS 기본 모듈과 같이 빌드하여 실행하도록 하였다. MicaZ 상의 프로그래밍을 위해서 nesC로 작성된 프로그램은 avr-gcc(version -0.0tinyos-1w)를 이용하여 컴파일 하였다.

기본적으로는 각 암호들마다 eSTREAM 사이트에 제출되어 있는 소스 코드를 포팅하여 구현하고자 하였고, 메모리 사용 등의 문제로 직접적인 포팅이 불가능한 암호들에 대해서는 소스를 MicaZ에 적합하도록 적절히 변형하였다.

우선 각 eSTREAM 사이트에 제출되어있는 스트림 암호의 코드가 사용하고 있는 내부상태공간의 크기를 측정하였고 결과는 [표 1]과 같다.

Dragon, Phelix, Salsa20, SOSEMANUK, NLS, Rabbit의 경우 내부상태공간의 크기가 SRAM의 크기인 4KB보다 훨씬 적은 양의 메모리를 사용하기 때문에 별도의 변형 없이 간단한 포팅 작업만으로 실험이 가능하였다. 한편 HC-128의 경우 내부상태공간의 크기가 최소 4KB이기 때문에 구현이 불가능하였다. LEX의 경우 내부상태공간의 크기는 SRAM의 크기보다 작지만 AES 암호화 과정에서 이용하는 S박스 등 별도의 공간이 필요하기 때문에 소스코드의 변형이 필요하였고, Py와 CryptMT의 경우 4KB이상의 메모리를 요구하기 때

〔표 1〕 내부상태공간의 크기

스트림 암호	내부 상태 공간의 크기
Dragon	288B
HC-128	4300B
LEX	232B
Phelix	132B
Py	4196B
Salsa20	64B
SOSEMANUK	452B
CryptMT	4400B
NLS	228B
Rabbit	136B

문에 이 역시 적절한 변형 및 최적화가 필요하였다. 아래에서는 이러한 변형들에 대해 설명한다.

3.1 LEX

LEX는 내부 상태 공간으로 232B만을 사용하지만 LEX에서는 AES 암호화 루틴을 필요로 하기 때문에 AES에서 사용하는 S박스들의 크기를 추가적으로 고려하여야 한다. eSTREAM 사이트에 제출된 LEX 프로그램에서는 Barreto의 AES code[21]를 기반으로 하고 있는데, 이 버전에서는 encryption table이 개당 1KB씩 5개가 필요하다. 즉 총 테이블의 크기가 5KB이기 때문에 센서 노드 상에서 구현이 불가능하였다. 따라서 본 논문에서는 S-Box의 크기를 조정하였다. 구체적으로 S-Box의 입력의 크기를 Barreto의 구현방식에 따른 32비트가 아닌 AES 기본 명세에 따른 8비트로 하고, S-Box와 함께 $GF(2^8)$상에서의 2배, 3배 된 결과를 미리 저장하는 xTime 테이블을 구성하여, 메모리의 사용을 줄임과 동시에 적절한 성능을 보장할 수 있었다.

3.2 Py

Py의 경우도 원래의 구현에서 상당부분 변형이 필요하였다. 일단 내부 상태 공간으로 4196B가 필요하기 때문에 최적화를 통해 메모리 사용량을 줄이는 것이 필요하였다. Py는 Rolling Array를 이용하는데 이 때 이용하는 회전 연산은 속도 면에서 비효율적이기 때문에 최적화하여 사용하고 있다. 즉, 내부 상태 공간에 3개의 Rolling Array를 유지하는데 실제보다 큰 공간을 할당

해 사용하도록 하여 내부 상태 공간이 4KB보다 커지게 되었다. 추가적으로 암호화 과정에서 내부 상태 공간 크기의 수만 배 되는 공간을 더 할당하여 배열의 시작점을 실제 배열의 처음 부분으로 되돌리는 연산을 줄이도록 하였다. 그러나 그만큼 메모리를 더 사용하게 되어 실제로 사용되는 데이터 메모리는 매우 크기 때문에 속도상의 최적화를 최대한 지양하면서 메모리 사용량을 4KB 안에 맞추는 것이 필요했다.

먼저 내부 상태 공간의 크기가 4KB를 초과하기 때문에 불필요한 공간과 남는 공간에 대한 최적화가 필요하였다. 이 과정을 설명하기 위해서는 Py의 구조를 좀 더 자세히 분석해보는 일이 필요하다. Py의 초기화 과정 중에 IV 설정에서는 S 박스를 이용하여 P 순열을 만들고 s와 EIV Rolling Array를 업데이트 한 후, 3개의 Rolling Array에 Y배열의 크기만큼의 연산을 적용하여 업데이트를 하게 된다. (여기서 EIV Rolling Array는 초기화 과정에서만 임시로 사용하는 배열이다) 이러한 과정의 최적화를 위해 내부 상태 공간은 Rolling Array 3개에 대해서 Y배열만큼 더 할당하여 사용하고 있다. 그러나 EIV배열은 고정된 위치의 값만 업데이트하고 사용하기 때문에 간단한 연산으로 회전 없이 같은 효과를 낼 수 있어 EIV의 크기를 원래 크기로 줄임으로 낭비되는 공간을 제거하여 내부 상태 공간의 크기를 4KB 이내로 줄일 수 있었다. 이 외에도 속도 최적화를 위해 별도로 임시 변수들을 할당하여 이용했던 부분을 최소화함으로써 메모리 요구량을 감소시킬 수 있었다.

3.3 CryptMT

CryptMT는 eSTREAM에 제출된 소스 코드에서는 4400B의 내부상태공간을 사용하기 때문에 메모리 사용량을 줄이는 것이 필요하였다. CryptMT의 경우 키와 IV의 크기가 다양하여 최대 2048비트를 사용하게 된다. 다른 스트림 암호가 사용하는 키와 IV의 크기는 128비트 혹은 256비트이기 때문에 CryptMT의 키와 IV 크기도 각각 256비트, 128비트로 제한하였고 그 결과 내부 상태공간을 2760B만을 이용하여 구현이 가능하였다.

또한 CryptMT v3에서는 더욱 빠른 암호화 속도를 위해 128비트 단위의 변수를 이용한 SIMD 명령어를 이용하도록 하였지만 ATmega128L 프로세서에서는 이 명령어를 지원하지 않기 때문에 이러한 최적화 부분을 제거하였다.

Ⅳ. eSTREAM 암호의 성능 비교

이 절에서는 위의 스트림 암호들을 구현하여 MicaZ 센서 노드상에서 성능을 측정한 결과를 제시하며, 성능 비교의 척도로 사용할 수 있도록 블록 암호인 AES 및 하드웨어 기반 스트림 암호들에 대한 성능 측정 결과도 제시한다. 또한, 역시 성능 비교에 활용할 수 있도록 Pentium에서의 실험 결과도 제시한다.

4.1 성능 측정

센서 노드 간 통신을 할 때에는 적은 량의 데이터를 주로 전송하게 된다. 특히 MicaZ 센서 노드 플랫폼에서 사용하는 통신 칩 CC2420에서는 통신 큐 크기가 128바이트로 제한되어 있고 센서 간의 무선 통신의 표준인 ZigBee에서도 최대 패킷 크기가 128바이트로 제한되어 있다. 따라서 본 논문에서는 128바이트의 평문을 암호화하여 전송하는 상황을 가정하고 있다. 그리고 ATmega128L 기반 MicaZ 대상 이 외에, 비교를 목적으로 PentiumD에서도 이러한 시간을 측정하였으며, 정밀도를 높이기 위해 각각 1000번씩 수행하고 그것을 다시 5번 반복하여 평균값으로 시간을 측정하였다. 또한, 비교 목적을 위해서 하드웨어용 스트림 암호들도 레퍼런스 소스 코드를 이용해서 모두 소프트웨어 구현을 하여 시간을 측정하였다. 또한 AES의 CFB모드의 암호화 시간도 측정을 하였는데 센서 노드에 구현한 알고리즘은 LEX에서 사용한 방법을 이용하였고 PC에 구현한 알고리즘은 eSTREAM 벤치마크 프로그램 안에 들어있는 레퍼런스 소스 코드를 그대로 이용하였다.

아래 두 표는 위의 시간 측정 결과이다. 표에서 시간 단위는 μs이고 RAM과 ROM의 크기 단위는 바이트이다. 표의 두 번째 및 세 번째 열은 각각 키와 IV의 비트 수를 나타내며, 네 번째 및 다섯 번째 열은 키와 IV의 설정 시간이다. 여섯 번째 열은 128바이트 크기의 데이터를 암호화하는 시간이며, 일곱 번째 열은 128바이트 패킷에 대한 바이트 당 암호화 시간, 즉 (키 설정 시간 + IV 설정 시간 + 128바이트 암호화 시간) / 128 이다. [표 2]에서 여덟 번째와 아홉 번째 열은 해당 암호가 사용하는 RAM과 ROM의 크기이다. 표의 행들은 1 바이트 당 암호화 시간을 기준으로 정렬하여 표시하였다.

〔표 2〕 ATmega128L에서 eSTREAM 암호의 구현
(회색 부분은 AES 및 하드웨어 기반 스트림 암호들에 대한 성능 측정 결과임)

Primitive	Key	IV	Key setup	IV setup	128바이트 데이터 암호화	1바이트 당 암호화 시간	RAM SIZE	ROM SIZE
LEX	128	128	317.43	1103.24	5125.37	51.14	2775	17598
CryptMT	128	128	55.28	1731.17	5346.95	55.73	3453	27198
CryptMT	256	128	75.63	2270.29	5346.19	60.09	3469	27226
NLS	128	64	2511.45	2107.45	3844.50	66.12	1929	25134
NLS	128	128	2511.33	2417.93	3844.08	68.54	1937	25128
AES-CFB	128	128	315.65	0	9357.21	75.56	2776	15854
AES-CFB	256	128	315.65	0	9357.22	75.56	2792	15870
Dragon	128	128	288.81	3353.63	6574.02	79.82	3027	20996
Dragon	256	256	291.25	3375.48	6564.83	79.93	3059	21028
Rabbit	128	64	2900.74	2736.29	5778.59	89.18	823	23656
SOSEMANUK	128	64	4403.20	4607.55	3685.29	99.19	3187	53676
SOSEMANUK	256	128	4416.30	4625.68	3678.77	99.38	3211	53700
Salsa20	128	64	35.03	13.03	24678.79	193.18	783	15890
Salsa20	256	64	35.44	13.04	24712.88	193.45	799	15906
Py	128	64	2961.50	28800.56	2583.40	268.32	3571	16448
Py	256	128	3159.30	30274.14	2586.86	281.41	3595	16472
F-FCSR-H	128	128	35.35	20655.51	20560.31	322.27	767	13930
Phelix	128	128	6161.10	6064.03	32701.15	350.99	851	21924
Phelix	256	128	6176.30	6087.64	32698.24	351.27	867	21948
TRIVIUM	80	80	93.93	36718.09	34467.63	556.87	747	37056
TRIVIUM	80	64	94.11	36702.80	34508.19	557.07	745	37062
Pomaranch	128	64	22.74	38433.04	183423.26	1733.43	1650	13866
Pomaranch	128	112	22.82	38431.74	183425.10	1733.43	1656	13872
MICKEY-128	128	64	28.53	127808.20	370523.30	3893.40	845	15184
MICKEY-128	128	128	28.30	151460.20	371772.70	4088.00	853	15184
Edon80	80	64	7686.55	38028.51	624959.98	5239.65	2198	14896
Grain-128	128	96	1.46	141403.20	559563.80	5476.30	957	12836
DECIM	80	64	22.55	308033.46	1542908.32	14460.66	961	14694

4.2 성능 비교

키 설정 및 IV 설정시간을 제외하고 암호화 시간만으로 MicaZ상의 수행시간과 PentiumD상의 수행시간을 비교하였을 때 실제 클럭 차이인 300배(8Mhz : 2400Mhz)보다 적게는 8배, 많게는 110배 정도 암호화하는 시간이 추가적으로 증가 하였다. 예를 들어, CryptMT(128비트 키)의 경우 MicaZ에서의 128바이트 암호화 시간은 5346.95μs인데 반해 PentiumD에서는 1.5638μs로 3400배인데, 이것은 클럭 차이인 300배보다 11배 큰 것이다. 이러한 현상의 주된 원인은 우선 이 암호들이 32비트 컴퓨터에 맞게 최적화된 암호들이기 때문에 32비트 데이터를 처리할 경우 8비트 아키텍처인 ATmega128L에서는 4배의 클럭이 소요되는 것을 들 수 있다. 그리고 암호마다 8배에서 110배까지 차이가 다른 이유는 연산의 종류에 따른 차이라고 보여진다. 즉, 이들 암호에서 사용하는 연산에는 +, XOR 등과 같은 단일 연산이 있고 rotate나 swap, 그리고 워드에서 바이트, 바이트에서 워드로 변환하는 복합 연산이 있는데, 이들 연산을 사용하는 방식에 따라 시간의 증가에

〔표 3〕 PentiumD 3.4Ghz에서 eSTREAM 암호의 구현
(회색 부분은 AES 및 하드웨어 기반 스트림 암호들에 대한 성능 측정 결과임)

Primitive	Key	IV	Key setup	IV setup	128바이트 데이터 암호화	1바이트 당 암호화 시간
TRIVIUM	80	64	0.0559	0.4564	0.4119	0.0072
TRIVIUM	80	80	0.0559	0.4601	0.4119	0.0072
LEX	128	128	0.0990	0.1781	0.7383	0.0079
Salsa20	128	64	0.0186	0.0098	1.0422	0.0083
Salsa20	256	64	0.0186	0.0097	1.0401	0.0084
AES-CTR	**128**	**128**	**0.1899**	**0.0185**	**1.0478**	**0.0098**
Rabbit	**128**	**64**	**0.2673**	**0.3231**	**0.8968**	**0.0116**
SOSEMANUK	256	128	0.4503	0.2956	0.9401	0.0132
NLS	128	64	0.4132	0.2999	1.0900	0.0141
SOSEMANUK	128	64	0.4716	0.3801	0.9794	0.0143
AES-CTR	256	128	0.3170	0.0188	1.5126	0.0144
NLS	128	128	0.4132	0.3640	1.1549	0.0151
Phelix	128	128	0.2762	0.7630	0.9940	0.0159
Phelix	256	128	0.2527	0.7983	1.0212	0.0162
Dragon	128	128	0.0569	0.7252	1.3636	0.0167
Dragon	256	256	0.0572	0.7287	1.3571	0.0168
CryptMT	**128**	**128**	**0.0567**	**0.6919**	**1.5638**	**0.0181**
CryptMT	256	128	0.0655	0.7807	1.8043	0.0207
Py	128	64	1.4892	2.9141	1.1329	0.0433
Py	**256**	**128**	**1.5892**	**2.9674**	**1.1307**	**0.0444**
F-FCSR-H	**128**	**128**	**0.0252**	**3.6932**	**7.8545**	**0.0904**
MICKEY-128	**128**	**64**	**0.0209**	**22.7852**	**63.3085**	**0.6728**
MICKEY-128	**128**	**128**	**0.0209**	**27.0952**	**63.2235**	**0.7058**
Pomaranch	**128**	**112**	**0.0168**	**29.0457**	**235.4854**	**2.0668**
Pomaranch	**128**	**64**	**0.0168**	**29.0801**	**236.9588**	**2.0786**
Grain-128	**128**	**96**	**0.0214**	**79.2436**	**316.5400**	**3.0922**
Edon80	**80**	**64**	**3.8028**	**46.4228**	**482.6229**	**4.1629**
DECIM	**80**	**64**	**0.0242**	**163.7383**	**911.0583**	**8.3970**

큰 영향을 미치는 것으로 보인다.

예를 들어 CryptMT의 경우에는 복합 연산을 사용하지 않아 상대적으로 시간 증가가 적었던 반면에 Salsa20의 경우 quarterround 연산이 여러 개의 +, XOR, rotate의 연산으로 이루어졌기 때문에 데이터 암호화에서 큰 시간 증가를 보였다.

V. 최적화

위의 성능 비교 결과에 따르면 128바이트의 데이터를 암호화 할 때를 기준으로 LEX, CryptMT, NLS등은 AES보다 더 빠른 바이트당 암호화 시간을 나타냈으나, Rabbit, SOSEMANUK, Salsa20, Py, Phelix등은 AES에 비해 암호화에 소요되는 시간이 길었으며, Dragon은 비슷한 성능을 보여주었다. 이는 스트림 암호의 특성상 키 및 IV의 설정에 상대적으로 많은 시간을 소모하기 때문인데, 이 점을 고려하더라도 Salsa20과 Phelix는 키 및 IV 설정 시간을 제외한 데이터 암호화 시간에서도 AES보다 느린 것을 확인할 수 있다.

한편, IV 설정에서 AES 및 LEX는 앞서 설명한

〔표 4〕 SOSEMANUK, Salsa20, Rabbit의 상대적 수행시간 증가
(실제 클럭 차이인 300배를 제외하고 센서 노드에서 추가로 증가된 시간)

Primitive	Key	IV	Key setup	IV setup	128바이트 데이터 암호화
SOSEMANUK	128	64	**31**	**40**	12
SOSEMANUK	256	128	**32**	52	13
Salsa20	128	64	6	4.5	**79**
Salsa20	256	64	6	4.5	**79**
Rabbit	128	64	36	**28**	**21.5**

S-Box의 조정 등 각종 최적화가 적용된 code이나, 나머지 암호들은 추가적인 최적화의 여지가 남아 있다고 할 수 있다. 따라서 이 절에서는 최종적으로 eSTREAM에서 선정된 Salsa20, SOSEMANUK, Rabbit에 대해 별도의 추가적인 최적화를 통해 성능 향상을 시키고자 한다. 본 논문에서 분석한 바에 따르면, 센서 노드상에서 스트림 암호들의 성능이 저하된 이유는 32비트에 최적화되어 설계되어 있는 각 알고리즘들이 8비트의 프로세서에 구현되었기 때문이다. 따라서 이러한 알고리즘을 8비트 연산에 맞게 적절히 변형하면 성능 향상을 기대할 수 있다.

먼저, 우선적으로 개선할 대상이 어느 부분인지를 알아보기 위해 SOSEMANUK, Salsa20, Rabbit의 상대적인 수행시간 증가를 살펴보면 [표 4]와 같다. 두 번째와 세 번째 열은 키와 IV의 비트 수이고, 네 번째, 다섯 번째 그리고 여섯 번째 열은 Pentium과 ATmega128L의 실제 클럭 차이인 300배보다 더 증가된 시간의 배율이다.

표에서 굵게 표시한 부분은 특히 주목할 만한 대상을 의미하는데, 예를 들어 Salsa20의 데이터 암호화 시간은 거의 80배에 가까운 증가를 보여주고 있고, SOSEMANUK의 IV설정, 키 설정과 Rabbit의 키 설정, IV 설정 역시 상대적으로 큰 시간의 증가를 보여주고 있다. 아래에서는 이러한 문제점을 해결하기 위해 주요 연산을 개선하는 방법과 메모리의 사용 패턴을 개선하는 방법을 제안한다.

5.1 연산의 최적화

각 암호들은 32비트의 +, XOR, rotate들이 연산의 주를 이루고 있다. 이 연산들을 avr-gcc가 변환시킨 8비트 마이크로프로세서용 바이너리 코드에 대해 역어셈블(disassemble) 한 후 어셈블리코드 상에서 살펴보면, +의 경우 4개의 레지스터 중 최하위 레지스터는 add연산을 하고 상위 3개의 레지스터에 대해서는 adc (add with carry) 연산을 하여 하위 레지스터에서 올라온 캐리를 처리해 줌으로써 32비트 덧셈을 효율적으로 처리해 주었고, XOR의 경우 4개의 레지스터 단위로 XOR를 해주고 있어서 개선의 여지가 거의 없었다. 그러나 rotate의 경우에는 rotate 하는 비트수가 레지스터의 크기인 8비트의 배수가 아닌 경우 왼쪽으로 rotate할만큼의 비트 수를 시프트 해주고 오른쪽으로 32와 rotate해 줄 비트 수의 차를 시프트 해준 후에 XOR를 해주는 어셈블리코드가 생성되었다. 더욱이, ATmega128L 프로세서의 명령어 집합(instruction set)에는 1비트 시프트 및 4비트 시프트 이외의 시프트 명령이 없기 때문에, 최악의 경우 하나의 레지스터에 대해 32번의 시프트 연산이 수행되어 네 개의 레지스터에 대해 총 128번의 시프트 연산이 수행되는 것을 확인할 수 있었다. 최적화 대상인 세 암호들에서 rotate연산이 빈번하게 발생하고 있으므로, 이 연산을 8비트의 프로세서에 적합하게 변형하면 키 설정, IV 설정 그리고 암호화 시간이 대폭 단축될 것으로 기대할 수 있다.

이를 위해 본 논문에서는 우선 rotate연산을 8비트에 맞게 최적화 하기 위해, 상황에 따라 32비트 워드를 32비트 단위, 또는 8비트 단위로 4회 접근 가능한 32비트 크기의 유니온 자료구조를 [그림 8]과 같이 정의하였다.

다음에는, 먼저 레지스터 크기인 8비트의 배수인 8, 16, 24비트의 시프트에 대해서는 8비트 요소들끼리 적절히 위치를 바꾸어 레지스터 단위의 직접적인 변환을 구현함으로써 시간을 단축시키고, 추가적으로 필요한 시프트 또는 rotate에 대해서는 4개의 8비트 변수들에 대해 직접적으로 시프트 연산을 구현하였다. 예를 들어, 각 암호들의 rotate연산을 살펴보면 Salsa20의 경우 7비트, 9비트, 13비트, 18비트, SOSEMANUK의 경우 3비트, 5비트, 7비트, 11비트, 13비트, 22비트로 rotate하는 비트수가 정해져 있어서 각각의 경우에 대해 개별적으

```
typedef union
{
 uint32_t ori;
 uint8_t div[4];
}forRot;
```

〔그림 8〕 효율적인 rotate 연산을 위한 union 자료구조의 정의

로 함수를 만들어줌으로써 속도를 대폭 향상할 수 있었다. 이러한 최적화로 인해 Salsa20에 대해서는 ROM의 사용량에 있어서도 향상이 있었는데, 이것은 eSTREAM 사이트에 제출되어 있는 코드 상에서는 rotate 연산이 매크로로 구현되어 있던 반면에 본 논문에서는 이를 함수로 구현함으로써 코드 사용량이 감소하기 때문이다.

Rabbit의 경우에도 8비트, 16비트의 rotate연산이 있지만, 이 경우에는 8의 배수의 rotate이기 때문에 원래의 avr-gcc 자체가 레지스터끼리의 교체로 어셈블리코드를 생성해 주므로 추가적인 시간 단축은 기대하기 어렵다.

5.2 메모리(RAM) 사용 패턴의 조정

앞의 [표 1]을 보면 SOSEMANUK, Salsa20, Rabbit의 내부상태가 차지하는 공간은 각각 452, 64, 136B이다. 그런데 각 암호들을 MicaZ에 초기 구현한 결과에서는 각각 차지하는 RAM 메모리의 크기가 3211, 783, 823B로 상대적으로 큰 값을 보여주고 있다. MicaZ의 RAM 메모리의 크기는 4KB로 극히 제한되어 있기 때문에, 각 암호들은 RAM 사용패턴에 변화가 필요하다.

SOSEMANUK의 경우 키 설정에서 24라운드를 수행하며 생성된 3200비트의 임시 키를 저장하기 위하여 400B의 추가적인 공간을 필요로 하는데, 키 설정과 IV 설정을 동시에 수행해주면 ROM의 사용량이 증가하지만 ROM에 비해 상대적으로 훨씬 더 희소한 자원이라 할 수 있는 RAM의 요구량이 400B만큼 감소하게 된다.

한편, Rabbit의 경우에는 SOSEMANUK과 Salsa20과는 다르게 내부상태를 초기화 하는데 IV를 꼭 필요로 하지 않고 있다. 그렇기 때문에 eSTREAM 사이트에 제출된 소스코드에는 키 설정만 적용시킨 내부상태와 IV 설정까지 적용시킨 내부상태, 두 가지상태를 유지시키기 위하여 2개의 내부상태를 유지하고 있다. 그렇지만 MicaZ의 RAM 메모리의 크기 제한 때문에 두 가지 내부상태를 유지하는 것은 부담으로 작용한다. 따라서 내부상태를 하나만 유지하도록 수정하였다. 이렇게 할 경우 키 설정만 이루어진 내부 상태를 사용 할 수 없게 되지만 키와 IV를 설정 하는 일은 스트림 키를 생성하기 전에 결정되므로 IV 설정 함수 호출의 유무는 스트림 키 생성 전에 결정 할 수 있기 때문에, 키 설정만 수행한 내부상태를 가지고 스트림 키를 생성할지, 아니면 IV 설정까지 수행한 내부 상태를 가지고 스트림 키를 생성할지 선택 할 수 있다.

Salsa20의 경우에는 내부상태는 32비트 변수 16개만으로 이루어져있어서 RAM 크기에 대한 성능향상을 이끌어내기는 어려웠다.

5.3 최적화 결과

SOSEMANUK, Rabbit, Salsa20에 대해 연산을 최적화하고 메모리사용 패턴을 변화시킴으로써 [표 5]와

〔표 5〕 최적화된 SOSEMANUK, Salsa20, Rabbit의 성능

Primitive	Key	IV	Key setup	IV setup	128바이트 데이터 암호화	1바이트 당 암호화 시간	RAM SIZE	ROM SIZE
SOSEMANUK	128	64	1758.05	1963.52	2639.67	49.69	3187	55454
SOSEMANUK	256	128	1760.90	1962.45	2639.10	49.70	3211	55478
Rabbit	128	64	2848.20	2683.12	5681.17	89.18	755	22440
Salsa20	128	64	35.02	13.03	8239.14	64.74	783	14438
Salsa20	256	64	35.02	13.03	8239.27	64.74	799	14454
SOSEMANUK	128	64	1758.05	1879.46	2594.14	48.68	3187	64910
SOSEMANUK	256	128	1760.73	1878.37	2593.47	48.69	3211	64934
Salsa20	128	64	35.02	13.03	7786.98	61.21	783	16738
Salsa20	256	64	35.02	13.03	7787.45	61.21	799	16742

같은 결과를 얻을 수 있었다.

[표 5]에 의하면 Salsa20의 암호화 시간이 24712.88 μs 에서 8239.15μs로 줄어서 약 3배 정도의 속도 향상을 보였고, SOSEMANUK의 키 설정과 IV 설정, 그리고 암호화 시간 역시 각각 2.5배, 2.3배, 1.3배의 향상을 보였다.

[표 5]의 회색 부분은 좀 더 빠른 속도 향상을 위해 각 rotate들에 대해 함수가 아닌 매크로를 사용한 경우들로써, 이러한 경우 ROM 크기가 증가하는 대신 약간의 추가적인 속도 향상이 이루어졌다.

마지막 두 줄의 SOSEMANUK의 경우 키 설정과 IV 설정을 한번에 수행하여 키 설정 과정에서 낭비되는 RAM 메모리공간을 없앤 결과인데, ROM 사용이 다소 증가하지만, 4KB로 극히 제한적인 RAM 사용량을 400B 절약할 수 있다.

Ⅵ. 결 론

본 논문에서는 센서 노드에서 사용할 수 있는 스트림 암호를 선택하기 위해서 최근 주목을 받았던 eSTREAM 프로젝트에 제출 및 채택된 암호들을 구현, 비교하였다. Dragon, SOSEMANUK, Salsa20, Phelix, NLS, Rabbit의 경우에는 eSTREAM 사이트에 제출된 참조 코드만으로 초기 구현이 가능했고 LEX의 경우는 S-Box 크기 조정, Py의 경우에는 속도 증대를 위해 이용되었던 임시메모리 최소화, CryptMT의 경우 키와 IV 크기를 제한하고 확장 명령어를 이용하는 부분을 제거하여 구현이 가능했다. 그러나 HC-128은 내부상태 자체가 지나치게 많은 RAM을 사용하기 때문에 센서 노드에서 구현이 불가능하였다.

본 논문에서는 또한 eSTREAM 프로젝트에 최종 채택된 SOSEMANUK, Salsa20, Rabbit에 대해서는 초기 구현 이외에 속도향상을 위해 주요 연산에 대한 추가적인 최적화를 수행하고 메모리 사용패턴을 변형하여 RAM 사용량을 감소시켰다. 본 논문의 실험 및 분석에 따르면, eSTREAM 프로젝트에 최종적으로 통과한 세 가지 암호들은 순수 암호화 시간만을 놓고 비교할 경우 각각 406Kbps, 176Kbps, 121Kbps로써 비슷한 정도의 자원을 사용하도록 설정한 소프트웨어 기반의 AES보다 모두 빠른 성능을 보여주고 있다. 본 논문에서 구현 및 최적화된 스트림 암호 소프트웨어들은 각 암호들의 내부 상태에 비해 상대적으로 많은 RAM 공간을 요구하고 있는데, 이는 속도를 향상시키기 위해 할당한 사전 테이블 이외에도 센서 노드라는 특성상 순수 암호뿐 아니라 타이머나 LED같은 기타 자원을 위한 메모리 사용이 원인이라 분석된다.

한편, 단순히 기밀성(confidentiality)을 위한 암호화 및 복호화 이외에도 인증(authentication) 및 무결성(integrity) 등 각종 암호학적 요구사항에 대한 센서 네트워크의 충족 가능성을 확인하는 일이 필요하므로, 블록 및 스트림 암호 이외에도 해쉬 함수(hash function)나 메시지 인증 코드(message authentication code : MAC), 의사난수함수(pseudorandom function : PRF) 등 관련 프리미티브(primitive)에 대해서도 센서 노드 상에서 성능을 비교하고 분석해 보는 작업이 중요한 향후 연구가 될 수 있을 것이다.

참고문헌

[1] TinyOS, http://tinyos.net.

[2] A. Liu, P. Ning, "TinyECC : Elliptic Curve Cryptography for Sensor Networks (Version 1.0)," http://discovery.csc.ncsu.edu/softare/TinyECC/, November 2007.

[3] ZigBee, Wireless Control That simply Works, http://www.Zigbee.org.

[4] G. Meiser, T. Eisenbarth, K. Lemke-Rust, C. Paar, "Software Implementation of eSTREAM Profile I Ciphers on Embedded 8-bit AVR Microcontrollers," http://www.ecrypt.eu.org/stream/sw.html

[5] eSTREAM, the ECRYPT Stream Cipher Project, http://www.ecrypt.eu.org/stream/.

[6] NESSIE, https://www.cosic.east.kul- euven.be/nessie/.

[7] H. Wu, "Stream Cipher HC-128," http://www.ecrypt.eu.org/stream/hcp3.html/

[8] A. Biryukov, "A new 128 bit key stream cipher : LEX," http://www. ecrypt.eu.org/stream/lexp3.html

[9] D. Whiting, B. Schneier, S. Lucks, F. Muller, "Phelix - Fast Encryption and Authentication in a Single Cryptograp hic Primitive," http://www.ecrypt.eu. org/stream/phelixp2.html

[10] E. Biham, J. Seberry, "Py (Roo) : A Fast and

Secure Stream Cipher Using Rolling Arrays," http://www.ecrypt .eu.org/stream/pyp2.html

[11] D. Bernstein, "Salsa20," http://www. ecrypt. eu.org/stream/salsa20p3.html

[12] C. Berbain, O. Billet, A. Canteaut, N. Courtois, H. Gilbert, L. Goubin, A. Gouget, L. Granboulan, C. Lauradoux, M. Minier, T. Pornin, H. Sibert, "Sosemanuk, a fast software-oriented stream cipher," http://www.ecrypt.eu.org/stream/sosemanukp3.html

[13] M. Matsumoto, M. Saito, T. Nishimura, M. Hagita, "CryptMT Strea m Cipher Version 3," http://www.ecrypt.eu.org/stream/cryptmtp3.html

[14] P. Hawkes, M. Paddon, G. Rose, M. W. de Vries, "Primitive Specification for NLSv2," http://www.ecrypt.eu.org/ stream/nlsp3.html

[15] M. Boesgaard, M. Vesterager, T. Christensen, E. Zenner, "The Stream Cipher Rabbit," http://www.ecrypt.eu. org/stream/rabbitp3.html

[16] K. Chen, M. Henricksen, W. Millan, J. Fuller, L. Simpson, E. Dawson, H. Lee, S. Moon, "Dragon : A Fast Word Based Stream Cipher," http://www.ecrypt.eu.org/stream/dragonp3.html

[17] P. Ekdahl, T. Johansson. "A new version of the stream cipher SNOW," SAC 2002, LNCS 2295, pp. 47-61, 2002.

[18] E. Biham, R. Anderson, L. Knudsen. "SERPENT : A new block cipher proposal," FSE'98, LNCS 1372, pp. 222-238, 1998.

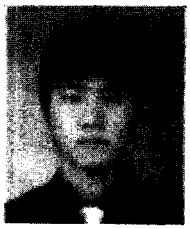
[19] Atmel, ATmega128(L) Datasheet, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf.

[20] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler. "The nesC language : A Holistic Approach to Networked Embedded Systems," PLDI 2003, June 2003.

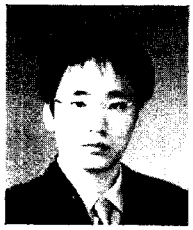
[21] Paulo Barreto's public domain C implementation of AES, http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndael-fst-3.0.zip.

[22] Y. W. Law, J. Doumen, P. Hartel "Survey and Benchmark of Block Ciphers for Wireless Sensor Networks," *ACM Transactions on Sensor Networks*, Vol. 2, No. 1, pp. 65-93, February 2006.

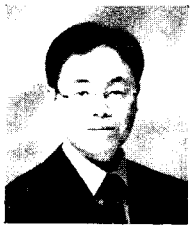
〈著者紹介〉



윤 민 (Min Yun) 학생회원
2008년 2월 : 인하대학교 정보공학계열 졸업
2008년 3월~현재 : 인하대학교 컴퓨터정보공학 석사과정
<관심분야> 정보보호, 유비쿼터스 보안 등



나 형 준 (Hyoung Jun Na) 학생회원
2006년 2월 : 한국과학기술원 전산학과 졸업
2008년 2월 : 서울대학교 컴퓨터공학부 석사
2008년 3월~현재 : 티맥스 소프트
<관심분야> 컴퓨터이론, 정보보호 등



이 문 규 (Mun-Kyu Lee) 정회원
1996년 2월 : 서울대학교 컴퓨터공학과 졸업
2003년 8월 : 서울대학교 전기컴퓨터공학부 박사
2003년 8월~2005년 2월 : 한국전자통신연구원 선임연구원
2005년 3월~현재 : 인하대학교 컴퓨터정보공학부 조교수
<관심분야> 암호알고리즘, 부채널분석 등

박 근 수 (Kunsoo Park) 종신회원
1983년 : 서울대학교 컴퓨터공학과 졸업
1992년 : Columbia University 전산학 박사
1991년 11월~1993년 8월 : King's College, University of London, Lecturer
1993년 8월~현재 : 서울대학교 컴퓨터공학부 교수
<관심분야> 컴퓨터이론, 암호학, 바이오인포매틱스 등