안전한 센서 네트워크를 위한 스트림 암호의 비교와 구현

나형준[O,1] 이문규[2] 박근수[1]
[1]서울대학교 컴퓨터공학부
[2]인하대학교 컴퓨터공학부
[1]{hjna[O], kpark}@theory.snu.ac.kr
[2]{mklee}@inha.ac.kr

Implementation and Performance Evaluation of Stream Ciphers for Secure Sensor Network

Hyoung Jun Na[O,1], Mun-Kyu Lee[2], Kunsoo Park[1]
[1]School of Computer Science and Engineering, Seoul National University
[2]School of Computer Science and Engineering, Inha University

요 약

최근 센서 네트워크에 대한 연구가 활발한 가운데 센서 네트워크에서의 보안에 관한 중요성 또한 대두되고 있어, 센서 노드 및 센서 네트워크 상의 정보를 안전하게 관리하기 위한 암호 알고리즘의 구현이 필수적이다. 센서 노드 상에서 이용될 수 있는 암호로는 TinyECC 등의 공개키 암호와 AES와 같은 표준 블록 암호가 있으나, 속도 면에서 좀더 효율적일 것으로 기대되는 스트림 암호는 아직 표준화된 바가 없으며, 현재 eSTREAM 프로젝트에서 표준화가 진행 중에 있다. 이에 본 논문에서는 센서 노드에 가장 적합한 스트림 암호를 찾기 위해서 eSTREAM의 2단계에 제출 되어있는 스트림 암호들 중 소프트웨어용 암호 7개를 구현하고 성능을 비교한다. 또한 참조 구현으로서 하드웨어용 스트림 암호 및 AES-CTR에 대한 실험 결과도 제시한다. 본 논문의 실험 결과에 따르면 위 스트림 암호 중 Dragon이 속도 측면에서 가장 효율적인 것으로 나타났으며, 초당 약 12.5KB의 암호화 성능을 보여 센서 노드에서 사용하기에 적합한 것으로 판단된다.

1. 서 론

무선 센서 네트워크는 많은 수의 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 제한 때문에 저장공간과 에너지 공급, 통신 대역폭에 제한을 받는다. 이러한 센서 네트워크는 센서 노드를 통한 정보를 감지하고 감지된 정보를 처리하여 우리의 삶을 자동화시키고 편리함을 제공한다. 하지만 일상 생활에 의존도가 높아질수록 이로 인한 위험성 또한 높아지기 때문에 센서 네트워크를 통해 제공되는 정보들을 신뢰하고 동시에 개인의 프라이버시를 보장할 수 있어야 한다. 즉 현실적이고 안전한 센서 네트워크를 구현하기 위해서 감지된 정보를 안전하게 처리하고 관리할 수 있는 암호 알고리즘이 필요하다.

크게 암호 알고리즘은 공개키 암호와 대칭키 암호로 나눌 수 있다. 센서 네트워크에서 사용할 수 있는 공개키 암호는 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 암호는 Zigbee[3]의 표준으로 AES가 사용되고 있지만 아직 스트림 암호는 표준이 정해져 있지 않은 상태이다.

이 논문에서는 센서 노드에서 가장 적합한 스트림 암호를 찾기 위해서 eSTREAM[4]의 2단계에 제안되어 있는 스트림 암호들 중 'focus cipher'로 선택된 7개의 소프트웨어 암호들을 구현하여 비교한다.

2. eSTREAM 프로젝트

eSTREAM[4]은 EU ECRYPT network에서 조직한 널리 보급되어 적용되기에 알맞은 새로운 스트림 암호를 선정하는 프로젝트이다. eSTREAM은 NESSIE 프로젝트[5]에 제출된 6개 스트림 암호의 실패로 인하여 시작되었다. 2004년 11월 처음 후보를 받았고 2008월 1월에 종료될 예정이다. 이 프로젝트는 각각의 단계로 나누어져 있고 소프트웨어용과 하드웨어용에 따라 알맞은 알고리즘을 찾는 것이 목표이다.

공식적으로 2006년 3월 27일에 1단계가 끝났고 2006년 8월 1일부터 2단계가 시작하여 각 용도에 따라 여러 개의 알고리즘들이 선택되었다. 본 논문에서는 센서에서 사용할 수 있는 암호로 2단계에 선택된 소프트웨어 스트림 암호를 비교 구현하고자 한다. eSTREAM은 여러 암호 알고리즘에 대해 많은 암호해독과 성능 측정을 통하여 6개월마다 재분류하고 있기 때문에 안정성과 성능을 보장할 수 있다.

eSTREAM의 2단계에서 'focus cipher'로 선택 된 소프트웨어 스트림 암호에는 Dragon[6], HC-256[7], LEX[8], Phelix[9], Py[10], Salsa20[11], SOSEMANUK[12]이 있다. 이 들 스트림 암호들은 128비트 혹은 256비트의 키를 가지고 있고 적당한 크기의 내부 상태 공간(Internal State)이 있으며 키와 초기 벡터(IV)를 이용하여 내부 상태 공간을 채우고 비선형 함수나 기타 알고리즘을 이용하여 내부 상태 공간을 업데이트 하면서 키스트림을 추출하는 형태를 갖고 있다.

다음에서는 이들 7개의 스트림 암호에 대해 설명할 것이다.

2.1 LEX

LEX의 디자인은 블록암호인 AES에 기초하고 있다. LEX는 128비트의 키와 128비트의 초기 벡터, 128비트의 AES를 이용한다.

처음 초기화는 128비트의 키에 대해여 AES 키 스케줄링을 수행하고 128비트의 초기 벡터에 대해서 AES로 한번 암호화를 하여 이 때 나온 128비트의 암호문을 내부 상태 공간으로

사용한다.

키스트림은 위와 같이 내부 상태 공간을 초기화 하고 AES를 이용하여 추출한다. 128비트 AES의 경우 10라운드로 구성이 되어있는데 LEX에서는 각 라운드마다의 내부 상태 공간을 이용하여 키스트림을 추출한다. 내부 상태 공간은 크기가 128비트 즉 16바이트이고 b0, b1, …, b14, b15의 16개의 바이트로 이루어진다. AES의 홀수 라운드 수행 후에는 내부 상태 공간에서 b0, b8, b2, b10의 순서로 32비트를 추출하고 짝수 라운드 수행 후에는 내부 상태 공간에서 b1, b9, b3, b11의 순서로 32비트 키스트림을 추출한다. 총 AES 한번 수행하는데 320비트의 키스트림을 추출한다.

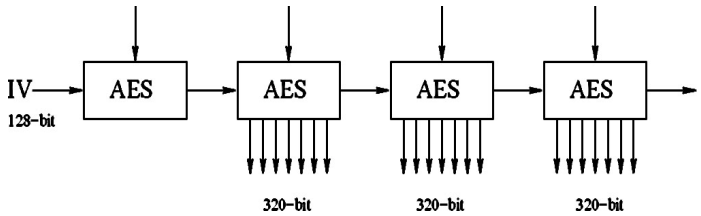

그림 1 LEX의 초기화와 키스트림 생성

2.2 Py

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

Rolling Array는 단위들이 주기적으로 한 단위씩 회전하는 벡터이다. Rolling Array는 회전하면서 수행하는 기초적인 연산으로 구성된다. 이러한 연산의 예로 2가지를 사용하는데 하나는 스왑 연산이고 다른 하나는 덧셈 연산이다.

회전은 S[0, …, N-1]을 배열이라고 보면 다음과 같이 수행을 한다.

tmp=S[0], S[0] = S[1], S[1] = S[2], …, S[N-1]=tmp

스왑 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 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. Py의 최적화는 시작점으로 되돌리는 연산을 최대한 줄이는 것이 중요하다.

Py에서는 2개의 Rolling Array를 사용하는데 서로 간에 영향을 준다. 하나는 P로 256바이트의 값을 순열로 갖고 매 단계마다 스왑 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 회전한다.

Py는 이렇게 2개의 Rolling Array P와 Y 그리고 32비트의 s를 내부 상태 공간으로 유지하면서 키스트림을 추출한다. 각 단계에서 P와 Y는 회전하면서 총 8바이트의 키스트림을 추출한다. s는 P로부터 Y를 간접적으로 접근하여 얻은 2개의 워드 값을 혼합하여 업데이트된다.

키초기화(Key Setup)는 키를 이용하여 Y를 초기화 한다. 빠르고 비선형적인 혼합을 위해서 8x8비트의 S 박스를 사용한다. 키초기화는 키의 크기, 초기 벡터의 크기, 키의 마지막 몇 바이트 값, S 박스인 내부 순열에 의존하여 s를 초기화하면서 시작하고 키를 두 번 s에 혼합한다. 그리고 s는 Y를 초기화하는데 이용된다.

초기벡터 초기화(IV Setup)는 2부분으로 나누어진다. 처음 부분에서는 S 박스를 이용하여 P 순열을 만들고 s와 EIV rolling array를 업데이트 한다. 그 다음 부분에서는 3개의 rolling array들을 업데이트하고 s를 업데이트 한다.

```
/* 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

2.3 Dragon

Dragon은 128비트 키와 초기 벡터 혹은 256비트 키와 초기 벡터를 이용한다. Dragon은 non-linear feedback shift register(NLFSR)을 기초로 하는 스트림 암호이다. Dragon은 1024비트의 NLFSR과 F라는 업데이트 함수, M이라는 64비트 메모리를 갖고 있다. F 함수는 한 라운드에 한번씩 호출이 되고 내부 상태 공간을 이용하여 64비트의 키스트림을 생성한다. 키초기화와 초기 백터 초기화에서는 키와 초기 벡터를 이용하여 Dragon의 내부 상태 공간을 채운다.

$$\text{Input} = \{B_0 \| ... \| B_{32}, M\}$$

1. $(M_L \| M_R) = M$
2. $a = B_0, b = B_9, c = B_{16}, d = B_{19}, e = B_{30} \oplus M_L, f = B_{31} \oplus M_R$
3. $(a', b', c', d', e', f') = F(a, b, c, d, e, f)$
4. $B_0 = b', B_1 = c'$
5. $B_i = B_{i-2}, 2 \le i \le 31$
6. $M = M + 1$
7. $k = a' \| e'$

$$\text{Output} = \{k, B_0 \| ... \| B_{31}, M\}$$

그림 3 Dragon의 키스트림 생성

2.4 HC-256

HC-256은 256비트의 키와 256비트의 초기 벡터를 이용한다. HC-256은 2개의 비밀 테이블로 구성 되어 있다. 각 테이

블은 32비트의 엘리먼트 1024개로 이루어져 있다. 매 2048 단계를 거쳐서 2개의 테이블의 모든 엘리먼트를 업데이트 한다. 각 단계에서는 32bit-to-32bit 맵핑을 이용하여 32비트 출력을 생성하고 선형 마스킹을 적용하여 의 키스트림을 생성한다.

```
i = 0;
repeat until enough keystream bits are generated.
{
    j = i mod 1024;
    if( i mod 2048 < 1024)
    {
        P[j] = P[j] + P[j - 10] + g1(P[j - 3], P[j - 1023]);
        si = h1(P[j - 12]) ⊕ P[j];
    }
    else
    {
        Q[j] = Q[j] + Q[j - 10] + g2(Q[j - 3], Q[j - 1023]);
        si = h2(Q[j - 12]) ⊕ Q[j];
    }
    end - if
    i = i + 1;
}
end - repeat
```

그림 4 HC-256의 키스트림 생성

2.5 Salsa20

Salsa20은 256비트 혹은 128비트의 키와 64비트의 초기 벡터를 이용한다. Salsa20은 해쉬 함수, 확장 함수, 암호화 함수로 구성이 된다. 64바이트의 입력과 64바이트의 출력을 갖는 해쉬 함수는 Salsa20의 핵심이다. 해쉬 함수의 counter 모드를 스트림 암호로 이용한 것이다. Salsa20은 64바이트의 평문 블록에 키와 초기 벡터, 블록 넘버를 해쉬한 결과를 xor 하여 암호화를 한다. 해쉬 함수는 여러 개의 quarterround로 구성 되어 있는데 quarterround를 행에 따라 적용하는 rowround, 열에 따라 적용하는 columnround, 이를 합한 doubleround로 구성이 되어 있다.

$$\text{if } y=(y_0,y_1,y_2,y_3) \text{ then quarterround}(y)=(z_0,z_1,z_2,z_3), \text{ where}$$
$$z_1 = y_1 \oplus ((y_0 + y_3) <<< 7).$$
$$z_2 = y_2 \oplus ((z_1 + y_0) <<< 9).$$
$$z_3 = y_3 \oplus ((z_2 + z_1) <<< 13).$$
$$z_0 = y_0 \oplus ((z_3 + z_2) <<< 18).$$

그림 5 Salsa20의 quarterround

$$\text{Salsa20}_k(v,0), \text{Salsa20}_k(v,1), \text{Salsa20}_k(v,2), ..., \text{Salsa20}_k(v,2^{64}-1).$$

그림 6 Salsa20의 키스트림 생성

2.6 Phelix

Phelix는 256비트 키와 128비트의 초기 벡터를 이용한다. Phelix의 내부 상태 공간은 각 32비트 씩 9개의 워드들로 구성되어 있다. 내부 상태 공간은 5개의 "active" 상태와 4개의 "old" 상태로 나누어지는데 "old" 상태는 키스트림 출력 함수에서만 이용이 된다. Phelix의 한 라운드는 하나의 "active" 상태를 다른 상태에 더하거나 xor 하는 부분과 그 "active" 상태를 회전하는 부분으로 구성 되어있다. 한 개의 32비트 워드 블록은 20개의 라운드로 구성이 된다.

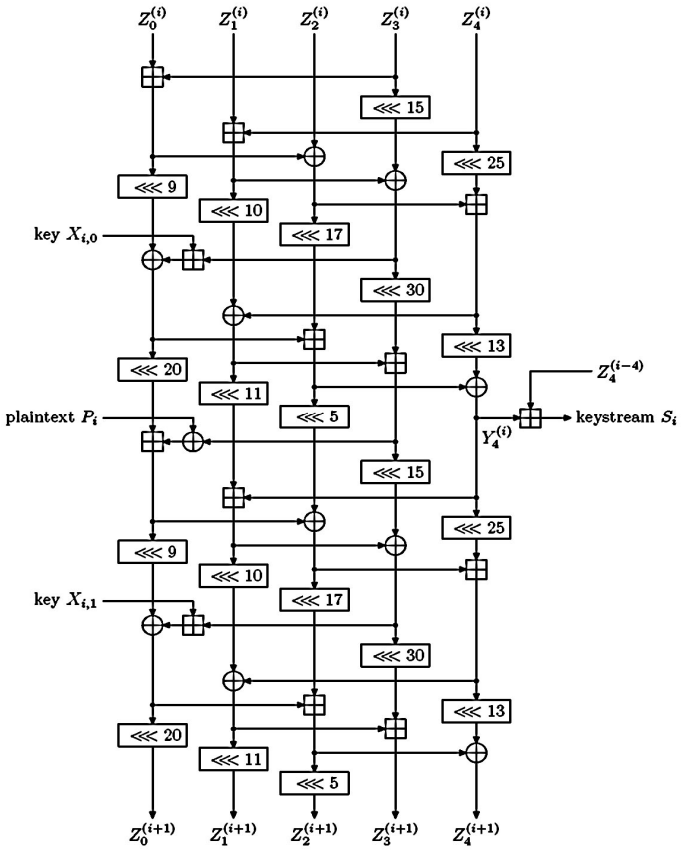

그림 7 Phelix의 키스트림 생성

2.7 SOSEMANUK

SOSEMANUK은 128비트와 256비트의 키와 128비트의 초기 벡터를 이용한다. SOSEMANUK은 SNOW 2.0[13]인 스트림 암호의 principle들과 SERPENT[14]인 블록 암호의 transformation들로 구성이 된다. 전체 32라운드로 구성이 되어 있는 SERPENT의 키 추가와 선형 transformation을 제외한 한 라운드 Serpent1을 키스트림 생성하는데 이용하고 24라운드만을 이용한 Serpent24를 암호화 과정에서 초기화에만 이용한다. SOSEMANUK은 Finte State Machine(FSM)과 Linear Feedback Shift Register(LFSR)를 내부 상태 공간으로 하여 각 FSM과 LFSR을 업데이트 하면서 출력을 생성하고 매 4개의 출력이 생성이 되면 4개의 출력 값에 Serpent1를 적용하여 키스트림을 생성한다.

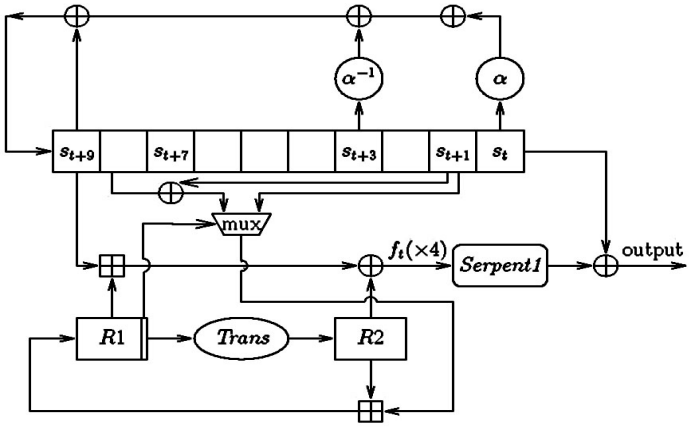

그림 8 SOSEMANUK의 키스트림 생성

3. eSTREAM 암호의 구현

위 스트림 암호들의 구현은 8비트 프로세서인 ATMega128L [15]을 이용하는 MicaZ Mote에서 하였다. ATMega128L은 128KB의 In-System Programmable Flash Memory인 프로그램 메모리, 4KB의 EEPROM, 4KB의 SRAM인 데이터 메모리로 구성이 되어 있다.

본문에서 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. 일반 PC용 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 9와 같이 명령어들을 SRAM을 거치지 않고 바로 프로그램 메모리에서 Instruction Register를 거쳐 Instruction Decoder로 가기 때문에 4KB SRAM 전체를 데이터를 위한 공간으로 활용할 수 있다.

센서 노드에서는 4KB의 SRAM을 가지고 있으나 소프트웨어 스트림 암호를 구현하기에 데이터 메모리가 부족할 가능성이 있다. 이러한 부족한 데이터 메모리를 보충하기 위해서 사용할 수 있는 것이 바로 EEPROM이다. EEPROM은 일반 PC의 하드디스크와 같이 데이터의 장기 보존을 위해서 이용하고 있는데 이 또한 크기가 4KB이기 때문에 부족한 데이터 메모리를 어느 정도는 보충할 수 있다. 그러나 EEPROM을 읽고 쓰는데 필요한 시간과 전력이 일반 SRAM보다 비용이 크기 때문에 EEPROM은 최대한 적게 사용하는 것이 중요하다.

구현 환경은 ATLMega128L을 사용하는 MicaZ 방식의 Mote 위에서 널리 사용되는 운영체제인 TinyOS이다. TinyOS는 nesC[16]로 구현되어 있고 모듈 방식으로 되어있기 때문에 스트림 암호를 모듈로 구현하여 TinyOS 기본 모듈과 같이 빌드하여 실행하도록 하였다.

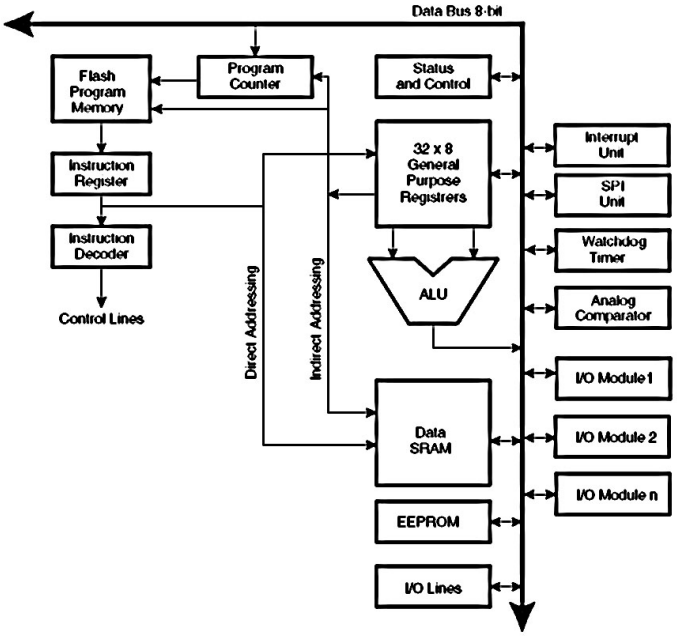

그림 9 ATMega128L의 구조[15]

아래에서는 각 스트림 암호를 어떠한 방식으로 구현하였는지 설명할 것이다. 기본적으로는 각 암호들마다 eSTREAM 사이트에 제출되어 있는 소스 코드를 포팅하여 구현하고자 하였으나, 일부 암호(특히 LEX 및 Py)는 메모리 사용 등의 문제로 직접적인 포팅이 적용되지 않았다. 이러한 경우에는 소스를 적절히 변형하여 MicaZ에 적합하도록 최적화하였다.

3.1 Dragon

Dragon은 내부 상태 공간으로 288바이트가 필요하기 때문에 평문과 암호문, 키, 초기 벡터, 그리고 기타 필요한 임시 변수를 저장할 공간이 4KB안에 충분히 가능하여 별도의 변형 없이 포팅이 가능하였다.

3.2 HC-256

HC-256은 내부 상태 공간이 최소 8KB가 필요하기 때문에 SRAM과 EEPROM 모두를 사용한다고 해도 MicaZ에서는 구현이 불가능 하였다.

3.3 LEX

기본적으로는 eSTREAM에 제출된 소스 코드를 포팅하였으나 상당부분 변형이 필요하였다. 내부 상태 공간은 232바이트만을 사용하지만 LEX에서는 AES의 암호화하는 부분을 필요로 하기 때문에 AES에서 사용하는 S박스들의 크기가 중요하였다. LEX에서 사용하는 AES는 Paulo Barreto's public domain C implementation of AES의 rijndael-alg-fst.c v3.0 implementation[17]을 기반으로 하였기 때문에 encryption table이 개당 1KB씩 5개가 필요하게 된다. 총 테이블의 크기가 5KB이기 때문에 SRAM만을 이용해서는 불가능하였다. 그래서 encryption table 중 4개의 테이블을 미리 EEPROM에 올리도록 하여 SRAM의 사용량을 줄이도록 하였다. EEPROM의 읽기 기능만을 사용하도록 하였기 때문에 큰 속도 저하 없이 구현이 가능하였다.

3.4 Phelix

eSTREAM에 제출한 소스 코드에는 참조 코드와 프로세서에 따른 어셈블리 최적화 된 소스코드가 있었다. 모든 구현을 하나의 형식으로 통일하기 위해서 참조 코드를 이용하여 구현하였다. 내부 상태 공간은 132바이트만을 사용하기 때문에 별다른 최적화 없이 구현이 가능하였다.

3.5 Py

LEX에서와 마찬가지로 원래의 소스코드에서 상당부분 변형이 필요하였다. 일단 내부 상태 공간으로 4196KB가 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 할당해 사용하도록 하여 내부 상태 공간이 4KB보다 커지게 되었다. 추가적으로 암호화 과정에서 내부 상태 공간 크기 수만 배 되는 공간을 더 할당하여 배열의 시작점을 실제 배열의 처음 부분으로 되돌리는 연산을 줄이도록 하였다. 그만큼 메모리를 더 사용하기 때문에 실제로 사용되는 데이터 메모리는 매우 크다. 이러한 이유로 최적화를 최대한 줄이면서 메모리 사용량을 4KB 안에 맞추는 것이 필요했다.

우선은 내부 상태 공간의 크기부터가 4KB가 넘어가기 때문에 불필요한 공간과 남는 공간에 대한 최적화가 필요하였다. 내부 상태 공간은 초기 벡터 초기화 과정에서 최적화를 위해서 rolling array 3개에 대해서 Y배열만큼 더 할당하여 사용하고 있다. 그러나 EIV배열은 고정된 위치의 값만 업데이트하고 사용하기 때문엔 간단한 연산으로 회전 없이 같은 효과를 낼

수 있다. 그래서 EIV의 크기를 원래 크기대로 하고 낭비되는 공간을 제거하도록 하였다.

그 다음으로 암호화 과정에서 최적화를 위한 공간을 제거하도록 하였다. 이 공간은 실제로 사용될 때 내부 상태 공간을 복사하여 rolling array 연산을 수행한 후 다시 내부 상태 공간에 복원시키는 방식으로 하기 때문에 작은 패킷을 암호화 하는 부분에서 비효율적이었다. 그리고 초기 벡터 초기화의 최적화 과정에서 사용했던 내부 상태 공간의 Y배열 크기 정도의 공간이 있었기 때문에 위의 공간을 제거하여도 Y배열 크기 정도의 최적화가 가능 하였다. 결국 어느 정도의 최적화를 보장하면서도 4KB SRAM을 이용해서 구현 가능 하였고 작은 패킷에 대해서도 비효율적이지 않도록 하는 것이 가능하였다.

3.6 Salsa20

eSTREAM에 제출한 소스 코드 중 참조 코드를 포팅하였다. 내부 공간은 64바이트만을 사용하기 때문에 별다른 최적화 없이 쉽게 포팅이 가능 하였다.

3.7 SOSEMANUK

SOSEMANUK의 내부 공간은 452바이트만을 사용하기 때문에 역시 별다른 최적화 없이 포팅이 가능하였다.

4. 실험 결과

실험은 ATMega128L 프로세서와 PentiumD 3.4Ghz 프로세서에 대해서 실험을 하였다. 각각의 실제 클럭은 8Mhz, 2400Mhz 이다.

4.1 시간 측정

PentiumD 프로세서에서는 기존 eSTREAM에서 제공하는 벤치마크 프로그램을 수정하여 128바이트 평문을 암호화 하는 시간, 키초기화 시간, 초기벡터 초기화 시간을 측정하였다.

ATMega128L 프로세서에서도 위와 같이 128바이트의 평문을 암호화 하는 시간, 키초기화 시간, 초기벡터 초기화 시간을 측정하였다. 정밀도를 높이기 위해 각각 1000번씩 수행하고 그것을 다시 5번 반복하여 평균값으로 시간을 측정하였다.

그리고 각각에 대해서 128바이트 패킷을 암호화 하는 시간으로 키초기화 시간과 초기 벡터 초기화 시간, 128바이트 암호화 시간을 합한 것을 추가하였다. 비교 목적을 위해서 하드웨어용 스트림 암호들도 레퍼런스 소스 코드를 이용하여 모두 소프트웨어로 구현하고 시간을 측정하였다. 또한 AES의 CTR 모드의 암호화 시간도 측정을 하였는데 센서 노드에 구현한 알고리즘은 LEX에서 사용한 방법을 그대로 이용하였고 PC에 구현한 알고리즘은 eSTREAM에 벤치마크 프로그램 안에 들어있는 레퍼런스 소스 코드를 그대로 이용하였다.

표1과 표2는 위의 시간 측정 결과이다. 표에서 모든 단위는 μs이고, 전체시간은 $\mu s/byte$ 이다.

4.2 시간 분석

전체적으로 PentiumD와 비교하였을 때 실제 클럭 차이인 300배 보다 최소 6배 정도 최대 260배 정도 암호화 하는 시간이 증가 하였다. 이 암호들은 32비트 컴퓨터에 맞게 최적화된 암호들이고 암호에 따라 32비트 데이터나 64비트 데이터를 처리하는 경우가 발생하여 이 경우 더 많은 연산을 사용하기 때문에 8비트 프로세서인 ATMega128L에서는 클럭 차이보다 더 많은 차이가 나는 것은 올바른 결과라고 생각된다. 그리고 암호마다 6배에서 260배까지 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 큰 영향을 미치는 것으로 보인다.

Primitive	Key	IV	Key setup	IV setup	128 bytes	전체시간
Dragon	128	128	288.81	3353.63	6574.02	79.82
Dragon	256	256	291.25	3375.48	6564.83	79.93
SOSEMANUK	128	64	4403.2	4607.55	3685.29	99.19
SOSEMANUK	256	128	4416.3	4625.68	3678.77	99.38
Salsa20	128	64	35.03	13.03	24678.79	193.18
Salsa20	256	64	35.44	13.04	24712.88	193.45
LEX	128	128	326.33	4667.44	19803.63	193.73
Py	128	64	2961.5	28800.56	2583.4	268.32
Py	256	128	3159.3	30274.14	2586.86	281.41
AES-CTR	128	128	326.38	9.64	37734.01	297.42
Phelix	128	128	6161.1	6064.03	32701.15	350.99
Phelix	256	128	6176.3	6087.64	32698.24	351.27
AES-CTR	256	128	417.24	9.64	53749.15	423.25
TRIVIUM	80	80	93.93	36718.09	34467.63	556.87
TRIVIUM	80	64	94.11	36702.8	34508.19	557.07
MICKEY-128	128	64	28.53	127808.2	370523.3	3893.4
MICKEY-128	128	128	28.3	151460.2	371772.7	4088
Grain-128	128	96	1.46	141403.2	559563.8	5476.3

표 1 ATMega128L에서 eSTREAM 암호의 구현

Primitive	Key	IV	Key setup	IV setup	128 bytes	전체시간
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	0.099	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
SOSEMANUK	256	128	0.4503	0.2956	0.9401	0.0132
SOSEMANUK	128	64	0.4716	0.3801	0.9794	0.0143
AES-CTR	256	128	0.317	0.0188	1.5126	0.0144
Phelix	128	128	0.2762	0.763	0.994	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
Py	128	64	1.4892	2.9141	1.1329	0.0433
Py	256	128	1.5892	2.9674	1.1307	0.0444
MICKEY-128	128	64	0.0209	22.7852	63.3085	0.6728
MICKEY-128	128	128	0.0209	27.0952	63.2235	0.7058
Grain-128	128	96	0.0214	79.2436	316.54	3.0922

표 2 PentiumD 3.4Ghz에서 eSTREAM 암호의 구현

4.2.1 Dragon

MicaZ에서는 PentiumD에 비해 4800배 정도의 시간 증가가 있었다. ROTATE 연산이 없어 실제 클럭 차이보다 큰 시간이 증가가 없었기 때문에 센서 노드에서 제일 빠른 속도를 보여주었다.

4.2.2 SOSEMANUK

7200배 정도의 시간 증가가 있었다. ROTATE 연산이 없어 실제 클럭 차이보다 큰 시간이 증가가 없었기 때문에 센서 노드에서 두 번째 순위를 차지하였다.

4.2.3 Salsa20

23000배 정도의 시간 증가가 있었다. ROTATE 연산을 사용하기 때문에 보다 큰 증가를 보였지만 키초기화 시간과 초기벡터 초기화 시간이 매우 작았고 시간 증가도 LEX보다 적었기 때문에 센서 노드에서 세 번째 순위를 차지하였다.

4.2.4 LEX
25000배 정도의 시간 증가가 있었다. ROTATE나 SWAP 같은 복합 연산이 없었지만 앞서 설명한 대로 SRAM의 부족으로 EEPROM을 사용하기 때문에 EEPROM을 읽는데 있어 시간이 크게 증가하여 네 번째 순위를 차지하였다.

4.2.5 Py
6200배 정도의 시간 증가가 있었다. PentiumD에서 작은 패킷 암호화에 좋지 않은 속도를 보였지만 키스트림을 추출할 때 불필요한 공간 복사를 제거 하였고 시간 증가가 적었기 때문에 다섯 번째 순위를 차지하였다.

4.2.6 Phelix
22000배 정도의 시간 증가가 있었다. ROTATE 연산을 사용하기 때문에 큰 증가를 보였고 PentiumD에서도 좋은 속도를 보여주지 않았기 때문에 여섯 번째 순위를 차지하였다.

5. 결 론

최근 센서 네트워크에 대한 관심이 고조되는 가운데 센서 네트워크의 보안에 관한 연구도 많이 진행되고 있다. 센서 노드에서 사용할 수 있는 암호 알고리즘으로 이미 공개키 암호와 블록 암호들이 제안된 바 있지만 이들은 속도 면에서 효율적이지 못하다. 따라서 센서 노드를 통해서 감지하는 정보의 양이 커지게 되면 이를 암호화 하는데 있어 상대적으로 속도가 빠른 스트림 암호의 사용이 불가피 해질 것으로 보인다.

본 논문에서는 센서 노드에서 사용할 수 있는 스트림 암호를 선택하기 위해서 최근 관심을 받고 있는 eSTREAM 프로젝트에 제출된 암호 후보들을 살펴보았다. Dragon, SOSEMANUK, Salsa20, Phelix의 경우에는 별다른 변형 없이 참조 코드의 포팅이 가능했고 LEX의 경우는 AES의 최적화, Py의 경우에는 최적화를 줄이는 방향으로 구현이 가능했으며 HC-256은 많은 메모리를 사용하기 때문에 구현이 불가능하였다.

실험 결과 128바이트 크기의 데이터를 암호화 하는데 Phelix를 제외하고 모두 AES보다 나은 성능을 보여주었으며 Dragon의 경우에는 AES보다 3~4배 빠른 속도를 보여주어 센서 노드에서 가장 빠른 스트림 암호인 것으로 분석되었다.

결론적으로 센서 노드에서 기타 다른 하드웨어 암호 모듈을 장착하지 않고 순수히 소프트웨어적으로 빠른 암호화를 하기 위해서는 스트림 암호를 사용하는 것이 효율적이며 속도 측면에서 비교했을 때 이러한 스트림 암호 중에서는 Dragon을 선택하는 것이 합리적이라고 판단된다.

참고 문헌

[1] TinyOS, http://tinyos.net.
[2] An Liu, Panos Kampanakis, Peng Ning, "TinyECC: Elliptic Curve Cryptography for Sensor Networks (Version 0.3)", http://discovery.csc.ncsu.edu/software/TinyECC/, February 2007.
[3] ZigBee, Wireless Control That Simply Works, http://www.Zigbee.org.
[4] eSTREAM, the ECRYPT Stream Cipher Project, http://www.ecrypt.eu.org/stream/.
[5] NESSIE
New European Schemes for Signatures, Integrity, and Encryption, https://www.cosic.esat.kuleuven.be/nessie/.
[6] Kevin Chen, Matt Henricksen, William Millan, Joanne Fuller, Leonie Simpson, Ed Dawson, Hoonjae Lee and Sangjae Moon, "Dragon: A Fast Word Based Stream Cipher", http://www.ecrypt.eu.org/stream/ciphers/.
[7] Hongjun Wu, "Stream Cipher HC-256", http://www.ecrypt.eu.org/stream/ciphers/.
[8] Alex Biryukov, "A new 128 bit key stream cipher: LEX", http://www.ecrypt.eu.org/stream/ciphers/.
[9] Doug Whiting, Bruce Schneier, Stefan Lucks, and Frederic Muller, "Phelix - Fast Encryption and Authentication in a Single Cryptographic Primitive", http://www.ecrypt.eu.org/stream/ciphers/.
[10] Eli Biham and Jennifer Seberry, "Py (Roo) : A Fast and Secure Stream Cipher Using Rolling Arrays", http://www.ecrypt.eu.org/stream/ciphers/.
[11] Daniel Bernstein, "Salsa20", http://www.ecrypt.eu.org/stream/ciphers/.
[12] Come Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, C. Lauradoux, Marine Minier, Thomas Pornin and H. Sibert, "Sosemanuk, a fast software-oriented stream cipher", http://www.ecrypt.eu.org/stream/ciphers/.
[13] E. Biham, R. Anderson, and L. Knudsen. SERPENT: A new block cipher proposal. In
Fast Software Encryption - FSE'98, volume 1372 of Lecture Notes in Computer Science,
pages 222-238. Springer-Verlag, 1998.
[14] P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In Selected
Areas in Cryptography - SAC 2002, volume 2295 of Lecture Notes in Computer Science,
pages 47-61. Springer-Verlag, 2002.
[15] Atmel, ATmega128(L) Datasheet, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf.
[16] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. In Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.
[17] Paulo Barreto's public domain C implementation of AES, http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndael-fst-3.0.zip.
[18] YEE WEI LAW, JEROEN DOUMEN, and PIETER HARTEL, "Survey and Benchmark of Block Ciphers for Wireless Sensor Networks", ACM Transactions on Sensor Networks, Vol. 2, No. 1, Pages 65-93, February 2006.