이진 조작을 통한 정적 스택 보호 시 발생하는 명령어 밀림현상 방지 기법

(Instruction-corruption-less Binary Modification Mechanism for Static Stack Protections)

이 영 림 † 김 영 필 †
(Young-rim Lee) (Young-pil Kim)

유 혁 ††
(Hyuck Yoo)

요 약 현재 많은 센서 운영체제에서는 메모리 제약 때문에 스레드 스택을 공유한다. 하지만 대부분의 대상 플랫폼에서는 MMU가 없어서 하드웨어적으로 스택 보호가 이루어지기 어렵다. 이러한 문제를 해결하기 위해 바이너리 코드에 스택 보호 기능을 가진 래퍼 함수를 추가 하고 바이너리 코드 안에 존재하는 스택 연산 명령어들을 스택 보호 기능을 가진 래퍼 함수호출로 바꾸어준다. 이때 스택 영역에 접근하는 명령어들과 스택 관리 모듈로의 분기 명령어간의 명령어 길이 차이에 의한 명령어 밀림현상이 발생한다. 이러한 문제를 해결하기 위해 본 논문에서는 밀림현상을 발생시키지 않고 임의의 명령어를 추가된 임의의 모듈을 호출하는 알고리즘을 제안하였다. 이 알고리즘은 제한된 도달 범위를 가지는 분기명령어를 반복적으로 사용하여 명령어 밀림현상 없이 추가된 임의의 모듈에 도달하게 한다. 본 논문에서 제안한 알고리즘은 센서 노드의 소프트웨어 보안 패치와 소프트웨어적 유지 보수를 용이하게 할 것이다.

키워드 : 센서 운영체제, 임베디드 시스템, 스택 보호

Abstract Many sensor operating systems have memory limitation constraint; therefore, stack memory areas of threads resides in a single memory space. Because most target platforms do not have hardware MMU (Memory Management Unit), it is difficult to protect each stack area. The method to solve this problem is to exchange original stack handling instructions in binary code for wrapper routines to protect stack area. In this exchanging phase, instruction corruption problem occurs due to difference of each instruction length between stack handling instructions and branch instructions. In this paper, we propose the algorithm to call a target routine without instruction corruption problem. This algorithm can reach a target routine by repeating branch instructions to have a short range. Our solution makes it easy to apply security patch and maintain upgrade of software of sensor node.

Key words : Sensor Operating system, Embedded System, Stack Protection

1. 서 론

센서환경에서 하드웨어를 제어하고 그 위에서 동작하는 응용프로그램의 관리를 위해서 운영체제의 존재는 필수적이다. 현재 개발되어 있는 센서 노드를 위한 운영체제는 프로그램 방식에 따라 이벤트 드리븐과 스레드 기반 방식으로 분류 할 수 있다. 스레드 기반방식이 동시성을 제공해 줄 수 있는 장점 때문에 각광 받고 있다[1].

멀티 스레드 시스템을 지원하는 센서 운영체제에서는 다중 스레드들이 동작하는 스택 영역의 보호와 동적으로 메모리 블록을 할당 받거나 반납하여 메모리 공간 사용의 효율을 높이는 메모리 관리가 필요하다.

하지만 현재 많은 멀티 스레드 센서 운영체제에서는 이를 지원하지 못하고 있다. MANTIS[2], Nano Qplus [3]와 같은 멀티 스레드 센서 운영체제는 메모리 제약 때문에 스레드간의 스택을 공유하는데 대부분 대상 플랫폼에 하드웨어 MMU가 없어서 스택 보호가 이루어지기 어렵다. 이로 인해 센서 노드에서 동작하고 있는 프로그램들이 오동작하거나 시스템이 다운되는 문제가 발생한다.

위와 같은 문제는 스택 보호 기능의 추가로 해결 할 수 있다. 스택 보호 기능을 추가하는 정적인 방법에는 소스 코드를 수정하는 방법과 바이너리 이미지를 수정하는 방법이다.

소스 코드를 수정하는 방법은 원본 소스에 추가적으로 스택 관리 모듈을 작성하고, 스택에 접근 할 때는 반

· 이 논문은 2007 한국컴퓨터종합학술대회에서 '바이너리 조작을 통한 정적 스택 보호 시 발생하는 명령어 밀림현상 방지 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 고려대학교 컴퓨터학과
yrlee@os.korea.ac.kr
ypkim@os.korea.ac.kr
†† 종신회원 : 고려대학교 컴퓨터학과 교수
hxy@korea.ac.kr

논문접수 : 2007년 9월 27일
심사완료 : 2007년 12월 28일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제14권 제1호(2008.2)

드시 스택 관리 모듈을 호출하도록 원본 소스를 수정하는 것이다. 이 방법은 소스 코드만을 수정하기 때문에 간편하다. 하지만 원본 소스 코드를 수정하기 위해서는 전반적인 소스 코드의 이해와 소스 코드의 재 컴파일이 필요하다. 이런 단점들은 작업의 복잡도, 비용, 시간을 증가시킨다.

두 번째 방법은 바이너리 이미지에서 스택 영역에 접근하는 모든 명령어를 스택 보호가 가능한 스택 관리 모듈을 호출하는 명령어로 덮어쓰는 방법이다. 이 방법은 첫 번째 방법에서 발생한 단점을 가지지 않는다. 또한 커널 개발의 유연성을 줄 수 있다. 하지만 이 방법은 스택 영역에 접근하는 명령어들과 스택 관리 모듈로 분기하는 명령어들 간의 명령어 길이 차이에 의한 명령어 밀림현상이 발생한다. 본 논문에서는 밀림현상을 발생시키지 않고 원하는 모듈을 호출하는 알고리즘을 설계하고, 실제 스레드기반 운영체제에서 스택 보호를 위해 제안된 Post-Compile Processing Tool[4]에 적용한다. 본 논문의 구성은 다음과 같다. 2장에서는 밀림 현상 방지를 위한 알고리즘을 설명하고, Post-Compile Processing Tool에 이 기법을 적용하는 시나리오와 실험및 성능 평가를 다룬다. 3장에서는 관련연구를 다루고, 마지막 4장에서는 결론과 함께 향후 연구의 방향을 제시한다.

2. 명령어 밀림현상 방지 알고리즘

2.1 명령어 밀림현상 원인

명령어 밀림현상이란 명령어들로 이루어진 바이너리에 새로운 명령어를 삽입할 때 이후의 명령어들의 메모리상의 위치가 밀리는 현상을 말한다. 이런 밀림현상에 의해 밀리는 명령어 중에 무조건 분기 명령어(unconditional branch), 조건 분기 명령어(conditional branch)의 목표 주소 값에 해당하는 명령어가 있다면, 이 명령어의 오퍼랜드인 목표주소 값이 원래 정해진 곳이 아니라 밀림 현상에 의해서 엉뚱한 곳의 값을 가지고 있는 것으로 되는 문제가 발생한다(그림 1). 이를 해결하기 위해서는 명령어 삽입이 아닌 덮어쓰기(overwriting)가 필요하다. 그러나 덮어 쓰고자 하는 명령어가 더 길면 이후의 명령어가 손상(corruption)된다. 대부분의 대상 플랫폼은 가변 길이 명령어 집합을 가지므로 이 문제는

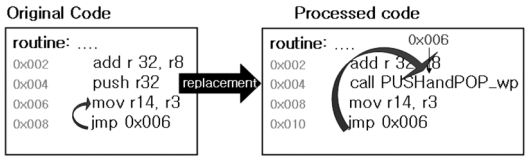


그림 1 명령어 밀림현상

피할 수 없다. 예를 들어 Tiny OS[5], MANTIS, Nano Qplus의 대상 플랫폼인 ATmega 128은 기본적으로 RISC구조로 2 바이트(byte) 명령어 길이를 가진다[6]. 그러나 예외적으로 분기 명령어는 4바이트 명령어 길이를 가진다.

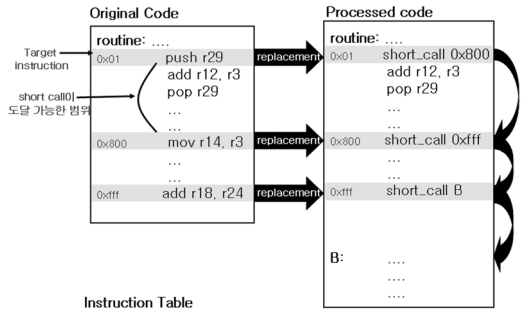
본 논문에서는 덮어쓰기로 밀림 현상을 해결하고 이로 인한 손상을 막을 수 있는 알고리즘을 제안한다.

2.2 short call을 사용한 명령어 밀림현상 방지 알고리즘

본 논문에서 제안하는 접근 방식은 여러 번의 short call을 사용하여 추가된 임의의 모듈에 도달한다. 이 접근 방식의 구조는 그림 2와 같다. short call(보조 분기 명령어)의 위치는 최초 목표 명령어에서 도달 가능한 가장 먼 명령어의 위치이며, 이는 중간의 징검다리의 역할을 수행한다. 즉, 추가된 임의의 모듈에 도달할 때까지 반복적으로 short call로 바꾸어 간다. 바꾸기 전의 명령어들(보조 Victim 명령어)은 추가된 임의의 모듈에서 관리되고 실행된다. short call이 작은 명령어 길이를 가지고 있기 때문에 short call 명령어로 덮어쓰기를 한다면 덮어 쓰여진 명령어를 제외한 다른 명령어에 영향을 주지 않고, 바이너리 수정하는 것이 가능하다. 구체적인 알고리즘을 설명하기 위해서는 몇 가지 용어가 정의되어야한다.

- 목표 명령어 : 바이너리에 원래 있던 명령어. 추가된 임의 모듈을 호출하는 명령어로 덮어써지는 명령어
- Primary 분기 명령어 : 목표 명령어를 추가된 임의 모듈로 분기하기 위해 덮어쓰는 명령어. short call 명령어
- 보조 Victim 명령어 : 바이너리에 원래 있던 명령어로써 징검다리역할을 하는 short call명령어에 의해 덮어써지는 명령어
- 보조 분기 명령어 : 징검다리 역할을 하는 short call 명령어. 보조 Victim 명령어를 덮어쓰는 명령어

Short call을 이용해서 전체 도달 범위를 가지는 jump와 같이 역할을 수행하기 위해서는 목표 명령어를 Primary 분기 명령어(short call 명령어)로 바꾸고 도달 범위 안에 가장 멀리에 있는 보조 Victim 명령어를 보조 분기 명령어로 바꾼다. 추가된 임의의 모듈에 도달할 때까지 반복적으로 보조 Victim 명령어를 보조 분기 명령어로 바꾸어 간다. 이때 빈번한 Short call의 사용을 줄이기 위해 Short call의 도달 범위 안에 4byte 명령어가 있다면 이것을 보조 Victim 명령어로 삼고, 보조 분기 명령어인 Short call 대신에 call 명령어를 사용하여 추가된 임의의 모듈을 호출한다. 빈번한 short call의 사용은 실행 시간 증가와 메모리 낭비를 유발할 수 있으나 위와 같이 처리 하면 short call의 사용을 최소화 할



Instruction Table

Offset	Assistant Victim	Instruction	Operand	
0x01	0	push	r29	
0x800	1	mov	r14	r3
0xfff	1	add	r18	r24

그림 2 Short_call을 이용한 메커니즘

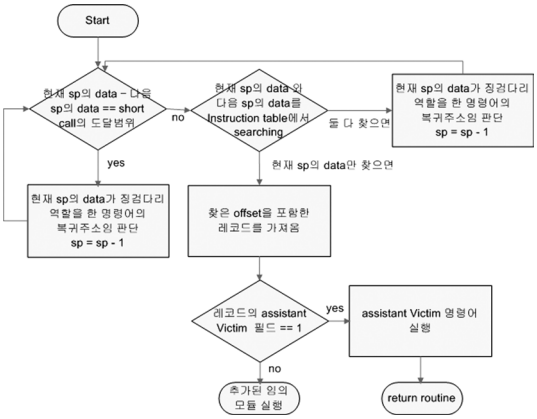


그림 3 스택 기반 시발점 구분 메커니즘

수 있다. 실제로 short call의 호출은 많은 경우 2번을 넘지 않았으므로 그 영향력은 미비하다. 보조 Victim 명령어는 그림 2와 같이 INSTRUCTION 테이블에 유지되고 추가된 임의의 모듈에서 관리된다.

보조 Victim 명령어를 원래 실행 흐름(flow)과 같게 실행하기 위해서는 보조 Victim 명령어에서 시작이 되어 추가된 임의의 모듈로 왔는지 아니면 Primary 분기 명령어(원래 추가된 모듈을 호출하는 명령어)에서 시작되어 추가된 임의의 모듈로 왔는지 구별이 필요하다. 즉 추가된 임의의 모듈로 오게 한 시발점이 되는 명령어를 찾아야 한다.

이를 위해 본 논문에서는 스택을 사용한다. Short call을 사용하기 때문에 복귀 주소(return address)는 스택에 있다. 이 복귀주소들을 보고 시발점이 된 명령어를 구별한다. 구별하기 위한 메커니즘은 그림 3과 같다.

이 메커니즘은 추가된 모듈의 제일 앞부분에서 동작해야 한다. 이 메커니즘을 통해 원래 추가된 임의 모듈을 호출하기 위한 명령어에서 시작되었는지, 중간에 징검다리 역할을 하는 명령어에서 시작되었는지를 구분한다. 구분의 결과, Primary 분기 명령어에서 시작되었다면, 모듈을 수행하고, 보조 Victim명령어에서 시작되었다면, 원래 명령어를 실행하고 최초의 실행 흐름으로 돌아간다.

2.3 실제 시나리오에 적용

본 논문에서 제안하는 기법은 본래 하드웨어 MMU가 없는 플랫폼환경에서 동작하는 멀티 스레드 기반 운영체제에서 스택을 보호하기 위해 제안된 Post-Compile Process Tool의 설계에서 발생한 밀림 현상을 해결하기 위해 고안된 기법이다.

현재 많은 센서 운영체제에서는 메모리 제약 때문에 스레드 스택을 공유한다. 하지만 대부분의 대상 플랫폼에서는 MMU가 없어서 하드웨어적으로 스택 보호가 이루어지기 어렵다. 이러한 문제를 해결하기 위해 Post-Compile Processing Tool의 구조와 세부 메커니즘이 제안되었다(그림 4). Post-Compile Processing Tool은 운영체제 바이너리 코드 안에 존재하는 스택 연산 명령어들을 스택 보호 기능을 가진 래퍼 함수호출로 바꾸어 준다. 이 래퍼 함수는 스택의 오버플로우/언더플로우를 관리해 주고 오리지널 코드에 있던 명령어를 실행한 후 원래 실행 흐름으로 돌아가게 한다.

본 논문에서 적용하는 시나리오는 한국전자 통신 연구원에서 개발된 Nano Qplus와 하드웨어 플랫폼으로 옥타 Nano-24를 대상으로 Post-Compile Process Tool을 사용하는 경우를 들었다.

툴을 적용했을 때, 분기 명령어 길이 차이에 의한 밀림현상이 발생한다. 대상 플랫폼의 명령어들은 기본적으로 2바이트로 정렬되어 있다. 하지만 여기서는 예외적으로 call 명령어의 크기는 4바이트이다. 따라서 코드 생성기(Code Generator)에서 push 명령어를 call 명령어로 바꾸고 2바이트 크기의 push/pop명령어 자리에 push/pop 명령어 대신 call 명령을 쓰게 되면, 그 뒤 모든 명령어의 주소를 뒤로 밀어야 하고, 메모리 오프셋(offset)의 재조정을 해야 한다.

본 논문에서 제안하는 기법을 적용하면 위와 같은 문

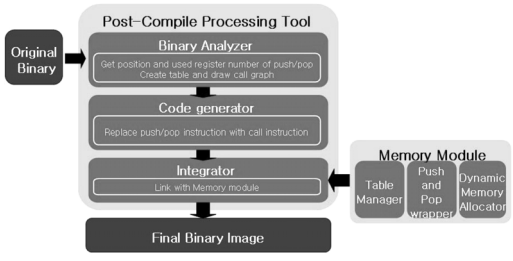
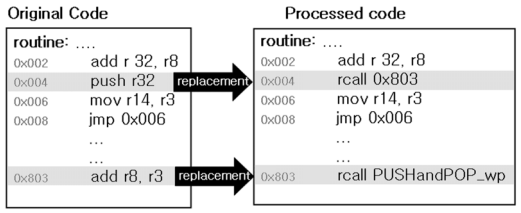


그림 4 Post Compile Processing Tool의 구조



Instruction Table

Offset	Assistant Victim	Instruction	Operand	
0x004	0	push	r32	
0x803	1	add	r8	r3

```
PUSHandPOP_wp:
  # while ()
  {
    while(현재 sp의 data – 다음 sp의 data != 2k)
        sp = sp -1
    if searching 현재 sp의 data & 다음 sp의 data in Instruction table
        sp = sp -1
    else {
        Instruction table에서 찾은 레코드를 가져옴
        break
    }
  }
  #if 찾은 레코드의 Assistant Victim 필드 == 1
  {
      보조 Victim명령어 실행
      return routine
  }
  #if 찾은 레코드의 instruction 필드 == push
  {
      if stack overflow occur
          call memory module
      else {
          push 명령어 실행
          return routine
      }
  }
  else{
          if stack underflow occur
              alarm stackunderflow
          else {
              pop 명령어 실행
              return rountine
          }
  }
```

그림 5 명령어 밀림 현상 방지 기법을 적용하여 개선된 Post Compile Processing Tool의 메커니즘

제를 해결할 수 있다. 대상 플랫폼인 Nano-24의 processor는 8-bit AVR 명령어 집합(instruction set)을 사용한다[6]. 여기서 short call 명령어는 rcall이고, rcall 명령어가 도달할 수 있는 오프셋은 2k의 범위(-2k <= offset <= 2k)이다. 툴의 코드 생성기는 push명령어를 rcall로 바꿔 주고, rcall의 오프셋이 가장 멀리 도달 가능한 명령어를 징검다리 역할을 하는 rcall 명령어로 바꾸어 준다. 그리고 보조 Victim 명령어들을 INSTRUCTION 테이블에 추가한다. PUSHandPOP_wp는 기존의 PUSHandPOP_wp 함수에 밀림방지 기법인 그림 3의 메커니즘을 적용한 함수이다(그림 5).

2.4 실험

이 장에서는 명령어 밀림현상 방지 알고리즘의 오버헤드를 분석하였다. 이를 위해 총 2개의 실험을 진행하였다. 실험 1은 function A를 호출하는 원본 바이너리 이미지를 소스 코드의 수정 없이 function B를 호출하는 바이너리 이미지로 바꾸어 주는 실험이다. 원본 프로그램은 function A를 호출하여 실행한다. Post-Compile Processing Tool은 원본 바이너리 이미지에 있는 function A를 호출하는 4byte call 명령어를 래퍼를 호출하는 명령어로 바꿔준다. 그 후 Post-Compile Processing Tool은 Instruction table에는 function A를 호출하는 명령어 대신 function B를 호출하는 명령어를 넣어 준다.

표 1 명령어 밀림현상 방지 기법의 오버헤드

	Post-Compile Processing Tool 적용 전		Post-Compile Processing Tool 적용 후			
	실험 1	실험 2	실험 1		실험 2	
전체 source 코드 라인 수	33	37	33		37	
수정된 source 코드 라인 수	N/A	N/A	없음		없음	
바이너리 이미지 크기	4810 B	4830 B	5666 B	+17.8%	5698 B	+18%
수행 시간	10msec 400usec	31msec 200usec	10msec 401usec		31msec 938usec	

실험 2는 기본적으로 실험 1과 같지만 4byte 명령어가 아닌 2byte 명령어가 목표 명령어이다. Post-Compile Processing Tool은 원본 바이너리 이미지에 있는 function A를 호출하는 2byte rcall 명령어를 래퍼를 호출하는 명령어로 바꿔준다. 그 후 Post-Compile Processing Tool은 Instruction table에는 function A를 호출하는 명령어 대신 function B를 호출하는 명령어를 넣어 준다.

실험 1과 2를 진행하여 명령어 밀림현상 방지 알고리즘의 오버헤드를 측정한 결과는 표 1과 같다.

2.4.1 메모리와 실행 시간 오버헤드

실험 1과 실험 2의 바이너리 이미지의 크기는 명령어 밀림현상 방지 기법을 적용한 Post Compile Processing Tool이 적용된 후에 18%정도 증가하였는데 이는 래퍼 루틴과 Instruction 테이블이 추가되었기 때문이다. 실험 1과 실험 2의 바이너리 이미지의 크기 증가량이 차이가 있는데 이는 명령어 밀림 현상 방지 기법을 적용하는 명령어의 수가 증가하면 Instruction 테이블의 크기가 증가하는 것은 피할 수 없기 때문이다.

표 1의 실행 시간을 살펴보면, Post Compile Processing Tool의 적용 전후의 실행 시간들을 살펴보면 차이가 있음 볼 수 있다. 이는 래퍼 루틴의 실행과 명령어 밀림현상 방지 기법의 적용에 따르는 실행 시간 오버헤

드이다. 하지만 실험 1과 2의 실행 시간 오버헤드는 1ms 미만으로 미미하다.

3. 관련 연구

본 연구는 MMU없는 플랫폼에서 MMU를 지원하는 연구와 동적 코드 업데이트 연구와 관련이 있다.

하드웨어 MMU가 없는 플랫폼에서 MMU를 지원하는 방법에는 여러 가지 접근 방법이 있다.

첫 번째, 하드웨어 MMU를 가진 시스템을 모방하는 접근 방식이 있다. 이 접근 방식은 응용프로그램에는 수정을 가하지 않지만. 모든 메모리 접근 시 가상 주소를 물리 주소로 바꿔주는 함수를 호출해야 하는 단점이 있다. Softvm[7]은 이 접근 방식을 사용했다.

두 번째, [8]에서는 컴파일러를 이용하여 가상 메모리를 관리한다. vm_assembler를 이용하여 어셈블리 코드에 존재하는 메모리와 관련된 명령어를 가상 주소 변환 함수(가상 주소를 물리 주소로 바꿔주는 함수)를 호출하는 명령어로 바꿔준다.

동적 코드 업데이트 연구에서는 임의의 명령어를 업데이트된 코드로 분기하는 명령어로 교체하고, 업데이트된 코드를 실행 후 원래 실행 흐름으로 돌아오는 여러 가지 방법이 제안 되어 왔다.

첫 번째, RISC기반의 플랫폼을 기반으로 한 [9]에서는 스프링보드를 제안했다. [9]에서는 레지스터 기반의 간접 분기 명령어(register based indirect branch)를 스프링보드에 넣어 새롭게 추가한 임의의 코드로 분기한다. 하지만 [9]에서는 위치적인 제약(스프링보드가 분기 명령어 으로 도달 가능한 범위 안에 있어야 하는 제약사항)과 공간적인 제약(스프링보드를 넣을 만한 홀)이 있어야 한다.

두 번째, CISC기반의 인텔 x86기반 수행된 GILK[10]에서는 코드 삽입(Code Splicing)과 반복 분기(Local Bouncing)를 제안했다. GILK[10] 역시 커널 코드에서 패치 코드로 분기하기 위해 분기 명령어를 사용한다. 본 연구는 두 번째 접근 방식과 유사하나 적용 플랫폼 면에서 차이가 있다.

4. 결 론

본 논문에서는 밀림현상을 발생시키지 않고 여러 번의 short call을 사용하여 추가된 임의의 모듈을 호출하는 알고리즘을 제안하였다. 실제 스레드 기반 운영체제에 서 스택 보호를 위해 제안된 Post-Compile Processing Tool설계에 이 기법을 적용 및 실험 하였다.

본 논문에서 제안한 알고리즘에 의해 수정된 바이너리는 재 컴파일 없이 바로 센서 노드에 올라갈 수 있는 장점을 가진다. 이는 개발의 유연성을 살릴 수 있고, 정적인 조작만을 가하므로 실행시간 오버 헤드가 적다. 또한 임베디드 환경과 같은 잦은 빌드를 요하는 구조에 적합하다. 본 논문에서 제안한 알고리즘을 사용하면 센서 노드의 소프트웨어 보안 패치와 소프트웨어적 유지보수가 용이할 것이다.

참 고 문 헌

[1] Rob von Behren, Jeremy Condit and Eric Brewer, "Why Event Are A Bad Idea," the 9th Workshop on Hot Topics in Operating System, Lihue, Hawaii, USA, May 18-21, 2003.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, R. Han, MANTIS: system support for multimodAl NeTworks of in-situ sensors, Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, September 19-19, 2003, San Diego, CA, USA

[3] Seungmin Park, Jin Won Kim, Kwangyong Lee, Kee-Young Shin, Daeyoung Kim, "Embedded Sensor Networked Operating System," Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '06) pp. 117-124.

[4] 이영림, 김영필, 유혁, "MMU가 없는 Thread기반 운영체제에서 스택 보호를 위한 메모리 관리 기법", 한국정보과학회 추계학술대회, 2006년 10월.

[5] LEVIS, P., MADDEN, S., GAY, D., POLASTRE, J., SZEWCZYK, R., WOO, A., BREWER, E., AND CULLER, D. The emergence of networking abstractions and techniques in tinyos. In Proceedings of the First Symposium on Networked Systems Design and Implementation (2004), USENIX Association, pp. 1-14.

[6] atmega128 datasheet, available : http://www.atmel.com/

[7] Bruce Jacob, "Uniprocessor Virtual Memory without TLBs," IEEE Transaction on Computers, Vol.50, No.5, May 2001.

[8] Siddharth Choudhuri, "Software Virtual Memory Management for MMU-less Embedded Systems," Technical Report CECS-05-16, 2005.

[9] Kishan A, Lam M, "Dynamic Kernel Modification and extensibility," Technical Report of SUIF Group, Department of Computer Science, Stanford University Stanford CA, 2002.

[10] David J. Pearce, Paul H.J. Kelly, Tony Field and Uli Harder, "GILK: A dynamic instrumentation tool for the Linux Kernel," In Proceedings of the 12th International Conference on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation(TOOLS '02), page 220-226, April 2002.