韓國 컴퓨터情報學會 論文誌
第10卷 第3號, 2005. 7.

2005-10-3-1-8

원격 실시간 제어 시스템을 위한 정적 프로그램 분석에 의한 보안 기법

임 성 수*, Kihwal Lee **

Ensuring Securityllable Real-Time Systems by Static Program Analysis

Sung-Soo Lim*, Kihwal Lee **

요 약

본 논문에서는 원격으로 제어 가능한 실시간 시스템에서 발생 가능한 악의적인 코드 삽입에 의한 보안 공격을 방지할 수 있는 방법을 제안한다. 제안하는 방법은 원격으로 제어 코드를 전달하여 시스템을 업그레이드하는 경우 전달된 제어 코드의 안전성을 실행 전에 분석하는 방법으로서 정적 프로그램 분석 방법을 이용한다. 제안하는 분석 방법을 구현한 도구를 컴파일러 내에 삽입하여 실제 원격으로 소프트웨어 업그레이드가 가능한 실시간 시스템 환경에 적용하여 효용성을 검증하였다.

Abstract

This paper proposes a method to ensure security attacks caused by insertion of malicious codes in a real-time control system that can be accessed through networks. The proposed technique is for dynamically upgradable real-time software through networks and based on a static program analysis technique to detect the malicious uses of memory access statements. Validation results are shown using a remotely upgradable real-time control system equipped with a modified compiler where the proposed security technique is applied.

▸ Keyword : 실시간 시스템, 정적 프로그램 분석, 시스템 보안
Real-time system, static program analysis, system security

• 제1저자 : 임성수
• 접수일 : 2005.05.14, 심사완료일 : 2005.07.11
* 국민대학교 자연과학대학 컴퓨터학부 전임강사,
** Department of Computer Science, University of Illinois at Urbana-Champaign, PhD student

Ⅰ. 서 론

실시간 임베디드 시스템의 응용 분야가 다양해지고 통신 기술이 발전함에 따라 망에 연결되어 원격지에 있는 시스템을 제어하는 형태의 실시간 시스템의 비중이 높아지고 있다. 이러한 망에 연결된 실시간 제어 시스템은 인터넷 등의 망을 통해 다양한 장소에서 원격으로 접속하여 제어하는 기능을 제공하기 때문에 기존 데스크탑 및 서버 시스템에서 발생하는 보안 문제를 야기한다. 따라서, 망에 연결된 실시간 제어 시스템에서 발생 가능한 외부로부터의 보안상의 공격을 파악하고 이를 방지할 수 있는 방법이 필요하다. 실시간 제어 시스템은 기존 데스크탑이나 서버 시스템과 비교하여 시스템의 자원 및 응용 소프트웨어의 형태에 있어서 제한적인 환경에서 운영되기 때문에 기존 데스크탑이나 서버 시스템과 다른 방법으로 보안을 보장할 수 있다.

망에 연결된 일반적인 시스템의 보안 문제는 시스템 무결성 보안(Integrity), 서비스 유지 보안(Service availability), 그리고 인증 보안(Confidentiality) 등의 세가지 문제로 분류한다. 시스템 무결성 보안이란 시스템의 운영과 관련한 부분이 악의적으로 수정되거나, 시스템 내에 일반적으로 접근이 금지된 영역에 악의적인 접근이 이루어져 시스템이 올바르게 동작하지 못하는 상태를 방지하는 것이다. 서비스 유지 보안이란 시스템에 지나친 부하를 가하는 등의 방법에 의해 시스템이 제공해야 할 서비스가 이루어지지 않도록 하는 공격을 방지하는 것이다. 다른 표현으로 '서비스 거절 공격(Denial of Service Attack)'을 방지한다고 한다. 마지막으로 인증 보안이란 서비스에 대한 접근 권한을 제어하는 방법으로 권한이 없는 사용자가 시스템 내에 접근하는 것을 막는 것이다.

실시간 임베디드 시스템 또한 망에 연결된 시스템 비중이 높아짐에 따라 위의 세가지 보안 문제를 모두 해결할 수 있는 방안을 갖추어야 한다. 인증 보안은, 실시간 제어 시스템의 경우 로그인의 과정을 거치지 않기 때문에 자연스럽게 유지된다. 서비스 유지 보안은 일반 데스크탑이나 서버의 경우와 같은 방법으로 유지할 수 있다. 시스템 무결성 보안은 기존 일반 용도의 시스템에서의 해결 방법을 그대로 적용하기 어렵다. 그 이유는 실시간 임베디드 시스템의 여러 가지 특징 및 제약 조건에 기인하는데 Phillip Koopman은 [15]에서 임베디드 시스템 보안이 일반 시스템 보안 문제와 다른 점을 크게 네 가지로 설명하였다. 첫째는 비용에 민감하다는 점, 둘째는 현실 세계에 직접적인 영향을 준다는 점, 셋째는 전력 소모 제약이 따른다는 점, 그리고 넷째는 응용 소프트웨어의 크기가 대부분 작고 개발 환경도 이러한 응용 소프트웨어 작성에 필요한 필수 기능 중심으로 구성된다는 것이다. 다시 말하면, 실시간 임베디드 시스템의 보안 문제를 해결하기 위해서 동적으로 시스템에 부담을 가중시키는 일반 시스템을 위한 보안 문제 해결 방법은 모두 사용할 수 없다는 것이다. 따라서, 실제 실행 시 비용을 최소화하는 보안 문제 해결 방법이 필요하다.

본 논문에서 대상으로 삼는 시스템의 특징은 인터넷을 통해 외부에서 연결이 가능한 실시간 제어 시스템이라는 것이고, 동적으로 제어 코드 혹은 응용 소프트웨어를 업그레이드할 수 있다는 것이다. 즉, 새로운 기능의 제어 코드 혹은 더 나은 성능의 제어 코드를 수행할 수 있도록 시스템 운용 중에 시스템을 정지시키지 않고 시스템 내 제어 소프트웨어를 업그레이드할 수 있는 시스템을 말한다. 이러한 시스템에서 동적으로 시스템에 전달되는 제어 코드 내에 악의적인 코드를 삽입하여 시스템 보안에 해를 끼치는 상황을 방지하기 위한 방법으로 본 논문에서는 정적인 프로그램 분석 방법을 적용한다. 제어 코드를 시스템에 적용하기 전에 컴파일 단계에서 정적으로 완전하게 안전성을 확인할 수 있는 방법을 통해 망에 연결된 실시간 임베디드 시스템에서의 보안 문제 해결 방안을 제시한다.

본 논문에서 제안하는 정적 프로그램 분석을 통한 망에 연결된 실시간 임베디드 시스템의 보안 기법은 실제 동적으로 제어 코드를 업그레이드할 수 있도록 구축된 실험 환경을 통해 검증된다. 본 연구에서 활용한 동적 업그레이드 가능한 환경은 University of Illinois at Urbana Champaign의 Real Time Systems Laborotary에 설치 되어 있는 Simplex 구조에 기반한 Telelab[2] 환경이다. 본 논문에서는 제안하는 기법이 일반적인 실시간 제어 프로그램의 안전성 검사에 효과적으로 활용될 수 있음을 실시간 시스템을 위한 벤치마크 프로그램을 통해 검증하고, Telelab 환경에 실제로 적용하여 구현한 환경에서 악의적인 제어 코드를 시스템에 전달해 보안이 유지되는 지를 확인함으로써 유용성을 검증한다.

본 논문의 구성은 다음과 같다. 2 장에서는 프로그램의 안전성을 보장하기 위한 관련 연구를 소개하고, 3 장에서는 동적으로 업그레이드가 가능한 시스템에서 발생 가능한 보

안 문제를 설명하고, 4 장에서는 본 논문에서 제안하는 정적 프로그램 분석에 의한 보안 기법을 설명한다. 5 장에서는 본 논문에서 제안하는 기법을 실험을 통해 검증한 결과를 보이고, 6 장에서 결론을 맺는다.

II. 관련 연구

시스템 보안 문제는 전통적으로 크게 다음의 세가지 방법으로 해결되어 왔다. 첫째는 운영체제 자체 내에서 해결해주는 방법[2][10][11][16][17]으로서 전통적인 UNIX 운영체제의 메모리 관리 정책을 하나의 예로 들 수 있다. 이러한 정책의 기본 방향은 각 작업이 사용하는 메모리 영역을 운영체제에 의해 다른 작업들로부터 보호하는 것이다. 최근에는 이러한 메모리 관리 정책을 채용한 운영체제들이 실시간, 내장형 시스템에도 활용되기 시작하는 추세이나 아직 범용 시스템에서의 메모리 보호 기법의 모든 기능이 구현된 운영체제가 실시간, 내장형 시스템의 운영체제로 사용되는 것보다는 보다 간단한 형태의 제한적 메모리 보호 기능을 지닌 운영체제가 사용되는 경우가 많다. 둘째 해결 방법은 컴파일러에 의해 삽입된 코드가 실행 중 대상 응용 프로그램의 메모리 사용의 안전성 여부를 항상 검사하는 것이다[3][4]. 이러한 방법들을 쓰려면 안전성 문제를 야기할 가능성이 있는 모든 메모리 접근 부분에 메모리 안전성 검사 코드를 삽입하고, 삽입된 코드가 프로그램의 실행 중 계속 실행되기 때문에 실행 성능에 심각한 영향을 끼친다. 마지막 방법으로는 정적 프로그램 분석을 활용하는 방법이다[8][12]. 정적 프로그램 분석 방법을 사용한다는 것은 프로그램을 실행하기 전에 컴파일러의 도움, 혹은 별도의 분석 도구를 사용하여 프로그램의 안전성을 검사하는 것이다. 따라서, 실제 프로그램의 실행 시에는 추가적인 성능 비용을 야기하지 않는다. 이러한 성능 상의 이점이 있음에도 정적 프로그램 분석 방법을 범용 시스템을 대상으로 하여 적용하는 경우 밝혀낼 수 있는 시스템 보안 문제의 범위가 상당히 제한적이기 때문에 정적 프로그램 분석만으로 범용 시스템의 보안 문제를 해결하기는 어렵다.

프로그램의 정적 분석을 통한 안전성 검증과 관련해서는 Wagner 외의 방법[13], Rugina의 정수 계획법을 이용한 방법[9], 그리고 Xu 외의 기계어 수준의 코드만을 대상으로 한 방법[14] 등이 있다. Wagner 외의 방법은 고수준 언어 수준에서 프로그램의 흐름과 무관한 분석을 수행함으로써 버퍼 오버플로우 오류의 존재 여부를 검사하는 방법이다. 이를 실제 유닉스 시스템에서 사용하는 시스템 응용 등에 적용하여 추가적인 그동안 알려지지 않았던 추가적인 버퍼 오버플로우 오류를 밝혀내었다. 그러나, 이 방법은 프로그램 흐름과 무관한 방법을 사용하기 때문에 분석의 정확도가 낮아 실제로는 오류가 아닌데도 불구하고 오류로 판단하는 비율이 지나치게 높다는 단점이 있다. Rugina의 방법 또한 프로그램의 흐름에 무관한 방법을 사용함으로써 프로그램 분석의 정확성보다는 정적 분석 방법의 적용 가능성을 검증하는데 초점을 맞추었다. Xu 등의 방법은 기계어 수준의 코드를 대상으로 프로그램 흐름에 입각한 분석을 시도함으로써 기존의 Wagner 등의 방법과 Rugina의 방법에 비해 정확성을 높였다는 장점이 있으나 결국 안전성 분석 과정에서는 고수준의 프로그램 정보, 즉, 구문 구조 정보나 각 변수의 타입 정보 등이 필요하기 때문에 이러한 정보를 모두 사용자로부터 입력 받을 필요가 생기는 문제를 안고 있다.

본 논문에서는 이러한 방법들과 달리 프로그램 흐름에 입각한 방법을 취함과 동시에 컴파일러 내에 구현하는 방식을 취하여 컴파일러의 전반부 컴파일 수행과정에서 얻어지는 프로그램에 대한 각종 정보들을 그대로 이용하는 기법을 사용하였다.

III. 동적 업그레이드 가능한 실시간 시스템의 보안 문제

본 절에서는 본 논문에서 제안하는 정적 프로그램 분석에 의한 실시간 시스템의 보안 문제 해결 기법의 적용 대상 환경인 동적 업그레이드가 가능한 실시간 시스템 환경에 대해 기술하고, 제안하는 기법을 설명한다.

3.1 동적 업그레이드 환경에 대한 가정

본 논문에서는 실시간 시스템에서 사용하는 제어 소프트웨어의 특성을 고려하여 보안 검사의 대상 응용 프로그램의 구조에 대해 다음과 같은 가정을 한다.

첫째, 실시간 제어 응용 프로그램은 C 언어로 작성된다.

둘째, 응용 프로그램 내에 삽입된 어셈블리(inline assembly) 명령어들이 존재하지 않는다고 가정한다. 이는 삽입된 어셈블리 명령어들의 제약 없는 시스템 자원 접근을 막음으로써 문제의 범위를 한정하기 위해서이다.

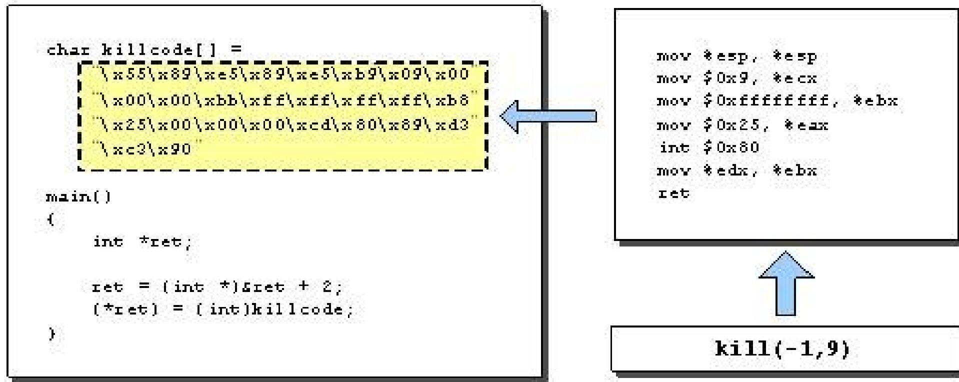


그림 1 버퍼 오버플로우 공격의 예
Fig. 1 Buffer overflow attack example

셋째, 본 논문에서는 응용 프로그램에서 안전성이 보장되지 않는 어떤 종류의 외부 함수 호출도 일어나지 않는다고 가정한다. 안전성이 보장되지 않는다는 것은 그 함수 내부에서 메모리 사용의 안전성이 완전히 검증되지 않은 경우를 말하며 이때에는 해당 함수의 호출이 금지된다. 이러한 가정은 함수 사이의 인자 및 결과물 전달을 고려해야 하는 요인을 제거하여 본 논문에서 다루고자 하는 보안 기법의 범위를 한정하기 위해서이다.

넷째, 본 논문에서는 어떤 종류의 시스템 콜도 허용하지 않는다. 이는 시스템 콜 내부적인 요인에 의한 시스템 보안 문제를 본 논문에서 다루는 문제의 범위에서 제외하기 위함이다.

이상의 가정은 문제의 범위를 제한하는 의미가 있지만 실제 실시간 제어 응용 프로그램의 특성을 고려하면 현장에의 적용 가능성을 크게 제한하지 않는 수준의 가정이다.

본 논문에서 대상으로 하는 원격으로 업그레이드가 가능한 실시간 시스템에서는 원격으로 업로드되어 대상 실시간 제어 시스템에 적용되는 제어 코드가 소스 코드 형태로 업그레이드 서버에 전달되고, 업그레이드 서버 내에서 컴파일 과정을 거쳐 제어 시스템에 전달된다고 가정한다. 아래에서 이러한 환경 하에서 발생할 수 있는 악의적인 제어 코드에 의한 보안 문제를 파악하고, 이를 해결할 수 있는 방안을 제안한다.

3.2. 악의적인 제어 코드에 의한 보안 문제

악의적인 제어 코드라 함은 주어진 제어 기능을 구현한 것처럼 가장하고 있지만, 허용되지 않는 시스템 자원의 접근이나 기타 시스템 파괴를 목적으로 하는 여러 작업들을 수행하는 코드를 포함한 제어

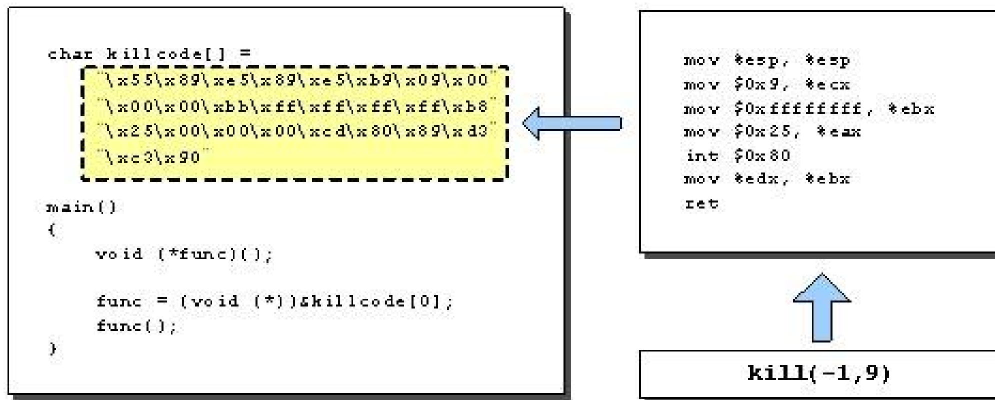


그림 2 데이터 영역에서의 코드 실행 예
Fig. 2 Code execution from data area example

코드를 말한다. 이러한 악의적인 코드들은 대개의 경우 공격 대상 시스템에서 사용하는 자원에 대한 접근 권한 관리가 완벽하지 않은 점을 이용한다. 예를 들면, C 언어로 작성된 응용 프로그램을 실행하는 경우, 컴파일러 및 실행 시스템이 데이터 타입 및 메모리 접근에 대한 유효성 검사를 완벽하게 수행하지 않는 점을 이용하여 악의적인 코드가 실행되도록 한다. (그림 1)과 (그림 2)를 통해 악의적인 코드의 대표적인 예를 보인다.

(그림 1)의 예는 주어진 합법적인 메모리 공간을 벗어나는 메모리 접근을 함으로써 야기되는 시스템 보안상의 문제를 보인다. 이 예는 시스템 해킹의 가장 대표적인 사례가 되고 있는 버퍼 오버플로우 공격 중 스택 오버플로우의 예를 보이고 있다. 가정에 의해 명시적인 시스템 콜의 사용이 불가능한 상황에서도 이러한 스택 오버플로우에 의해 시스템 콜의 호출이 가능할 수 있음을 보인다. (그림 1)의 배열 killcode 내에는 묵시적으로 시스템 콜을 호출하여 특정 프로세스의 실행을 정지시키는 기계어가 포함되어 있다. 문장 ret = (int *)&ret + 2;는 포인터 변수 ret가 스택의 복귀 주소가 저장된 지점을 가리키도록 만든다. 그리고, main 함수의 맨 마지막 문장에 의해 함수의 복귀 주소가 배열 killcode의 시작 주소로 변경됨으로써 main 함수의 실행이 끝나고 복귀하는 경우 이 배열의 내용이 실행된다. 이러한 스택 오버플로우에 의한 공격을 방지하기 위해서는 프로그램 실행 중에 스택의 영역이 보호되어야 한다. 다시 말하면, 각 함수에서 사용되는 스택의 영역이 알려져 있어야 하고, 이 영역을 벗어나는 영역을 접근, 혹은 내용을 변경하는 경우에는 허용되지 않는 작업으로 인식되어 작업이 수행되지 않도록 해야 한다.

(그림 2)에 나타난 둘째 예는 첫째 예와 같은 효과가 나타나지만 스택 오버플로우를 이용하는 것이 아니고 단순히 배열의 시작 번지로 분기를 하는 것이다. 이는 C 언어가 변수의 형식을 엄밀하게 조사하지 않는다는 특성을 이용하는 것이다. 이러한 종류의 시스템 공격을 막기 위해서는 애초에 함수 포인터의 사용을 금지시킬 수도 있으나 때로는 함수 포인터의 사용이 부득이한 경우도 있게 된다. 함수 포인터를 사용하는 상황에서 (그림 2)와 같은 공격을 막기 위해서는 데이터 영역으로 할당된 메모리 공간에 대해 명령어 실행을 금지시키는 방법을 사용할 수 있다. 즉, (그림 2)의 killcode 배열과 같이 데이터의 운용을 위해 할당된 영역으로의 분기가 일어나는 것을 금지시키는 것이다. 이러한 것은 모든 분기를 조사하여 명령어를 위한 메모리 영역을 벗어나는 영역으로의 분기가 일어나는지 파악하는 방법으로 해결해야 한다.

Ⅳ. 프로그램의 안전성 검사를 위한 정적 프로그램 분석 방법

본 절에서는 프로그램의 메모리 안전성을 검사하기 위한 프로그램 분석 방법을 제시한다. 해당 분석 방법은 프로그램

의 실행 전에 프로그램의 소스 코드를 대상으로 하여 컴파일 단계에서 검사하는 정적 프로그램 분석 방법을 사용한다.

4.1 기본 개념

메모리 안전성을 보장한다는 것은 각 개별 프로그램이 인위적으로, 혹은 실수로 메모리 사용의 오류를 범하여 다른 프로그램의 실행이나 시스템의 안전에 위협을 주는 것을 막는다는 뜻이다. 단일 주소 공간(single address space)을 사용하는 소규모 운영체제에서는 다중 프로세스 운영체제에서 볼 수 있는 개별 작업을 위한 메모리 보호 기능이 없기 때문에 각 응용 프로그램의 메모리 안전성을 보장하는 것이 시스템의 안정적 운용에 결정적인 요소가 된다.

이에 더하여, 최근 실시간 및 내장형 시스템에서는 C 프로그래밍 언어를 사용한 응용 소프트웨어 개발의 비중이 커지고 있기 때문에 이러한 메모리 안전성이 더욱 중요한 문제로 대두된다. 그 이유는 C 언어가 가지고 있는 특성 중에는 메모리의 안전한 접근을 개발자가 책임지고 보장해야 하는 측면이 있기 때문이다.

실시간 시스템의 응용 프로그램들은 그 구조가 비교적 간단하고 데이터의 사용 행태가 대부분 일정한 크기의 배열을 대상으로 한 연산이 주류를 이루고 있어서 정적인 분석만으로도 메모리 안전성 검증을 할 수 있다. 본 논문에서는 프로그램 내의 구문들의 메모리 사용을 추적할 수 있는 정적 프로그램 분석 기법을 활용하여 메모리 사용에 있어서의 두 가지 측면을 검사하는 기법을 제안한다.

첫째, 배열 등 일정 크기의 메모리 영역을 할당 받고 그 영역 내에서 연산을 수행하는 프로그램 내에서 그 할당된 영역을 벗어나는 연산이 있는지를 검사하는 것이다. 이는 최근 해킹의 주요 기법으로 드러난 버퍼 오버플로우 여부를 검사하는 것과 유사하다.

둘째, 데이터 영역으로 활용되는 메모리 영역에서 명령어 실행을 시도하는 지 여부를 검사하는 것이다. 다시 말하면 프로그램 내에서 데이터 사용을 위해 할당 받은 배열 등의 메모리 영역 내로 명령어 분기가 발생하여 데이터로 인식되었던 내용들이 실행 가능한 명령어들로 재인식되는 것을 막는다는 것이다.

본 절에서는 이러한 두 가지 측면을 정적으로 분석하는 방법을 보인다.

4.2 정적 메모리 안전성 분석 방법

본 논문에서 제안하는 방법은 기호적 실행(symbolic execution)에 의한 메모리 안전성 분석 방법이다. 기호적 실행이라 함은 분석 대상 프로그램을 고수준 프로그래밍 언어로 작성된 프로그램과 기계어로 표현된 목적 코드 사이의 표현 방법인 '중간 표현'으로 구성된 프로그램을 기반으로 하여 메모리 사용을 추적하는 방법이다. 이렇게 중간 표현으로 작성된 컴파일 과정중의 프로그램을 대상으로 하는 이유는 아래와 같이 정리할 수 있다.

첫째, C 언어와 같은 고수준의 프로그래밍 언어로 작성된 코드 상태에서는 정확하게 어느 구문이 메모리 접근을 야기하는 구문인지 파악하기 어렵다. 따라서, 기계어 표현에 가까운 단계까지 컴파일이 진행된 후에 메모리 사용을 추적해야 한다.

둘째, 완전히 목적 코드로 변환된 후에는 목적 코드가 실행될 프로세서의 기계어 명령어를 모두 알고 있어야 프로그램의 실행을 추적할 수 있다. 다시 말하면, 프로세서 시뮬레이터 수준의 분석 도구가 필요하고, 이러한 분석 도구는 프로세서가 바뀔 때마다 새로 준비되어야 한다.

중간 코드로 구성된 프로그램을 대상으로 분석을 수행하게 되면 프로세서 별로 따로 분석 도구를 구현해야 할 필요가 없어짐과 동시에, 중간 코드 내에는 어떤 구문이 메모리 접근에 해당하는 구문인지 알 수 있도록 충분한 정보를 제공하기 때문에 메모리 접근 추적이 용이하게 된다. 본 연구에서는 GCC가 사용하는 중간 표현인 RTL (Register Transfer Language)를 사용한다.

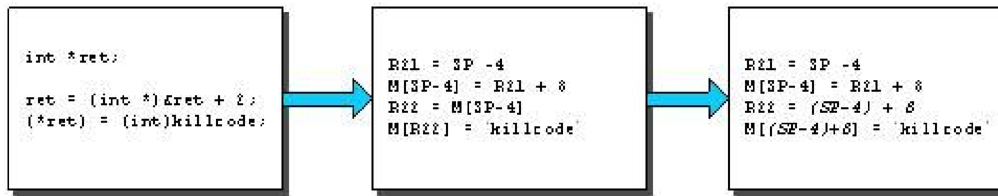


그림 2 RTL을 사용한 중간 표현의 예
Fig. 3 An intermediate representation using RTL

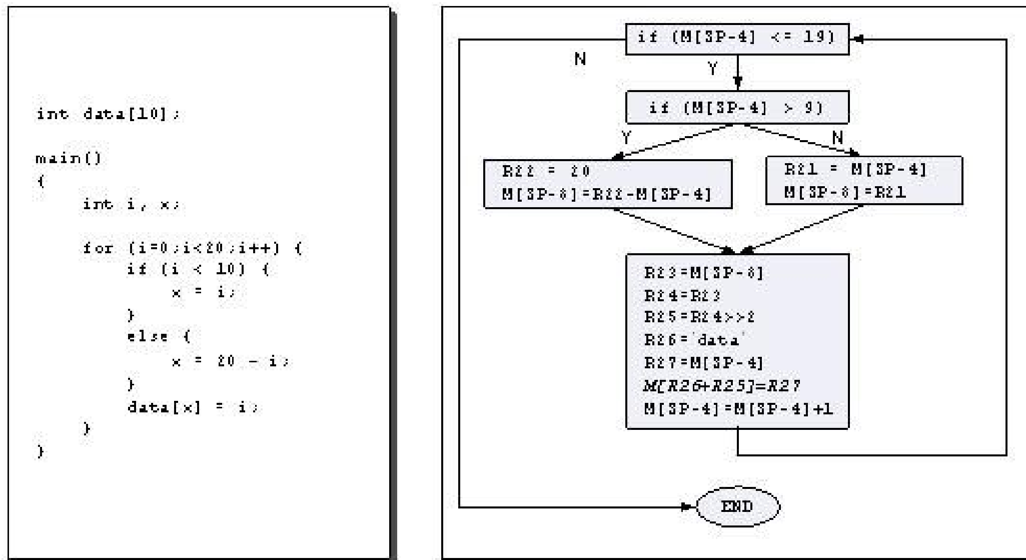
(그림 3)은 본 연구에서 사용하는 중간 표현, 즉, RTL의 예를 보인다. 맨 왼쪽 상자 내에 보이는 프로그램은 C 언어로 작성된 프로그램의 일부이다. 이 프로그램은 (그림 1)에서 보인 버퍼 오버플로우 공격 코드의 일부이다. 두 번째 상자에 있는 코드는 RTL을 사용하여 중간 표현으로 변환된 코드이다. 두 번째 상자의 RTL 코드에서 'M[XX]'로 표현된 항은 XX에 담겨 있는 주소를 통해 메모리 접근이 이루어진다는 뜻이다. 이러한 항이 좌변에 있으면 메모리 쓰기 접근이 이루어지는 것이고, 이러한 항이 우변에 있으면 메모리 읽기 접근이 이루어지는 것이다. 맨 오른쪽 상자에 있는 코드는 각 레지스터(여기서는 가상의 레지스터이다.) 내용을 추적하여 메모리 접근이 이루어지는 주소를 스택 포인터에 대한 상대적인 주소값으로 표현한 결과이다. 예를 들면, 세 번째 행의 R22의 내용은 두 번째 행에 의해 갱신된 '스택 포인터-4' 위치에 있는 값으로 결정되는데 이 위치에는 첫 번째 행에서 결정된 R21의 값, 즉, '스택 포인터 - 4'에 8을 더한 값인 '(스택 포인터 - 4)+8'의 값이 쓰여진다. 이러한 방법으로 마지막 행의 메모리 접근이 일어나는 주소가 추적된다. 이 주소에는 'killcode'라는 심볼로 등록된 주소가 쓰여진다. 여기서, 마지막에 접근되는 주소가 스택의 정해진 영역을 벗어나 접근이 이루어지고 있음을 알 수 있다. 따라서, 메모리 접근의 안전성을 위해서는 마지막 행의 메모리 접근은 실제 실행 시 포함되지 않거나, 이 프로그램 자체가 실행 개시되지 않도록 해야 한다.

이러한 방법으로 본 연구에서는 일단 RTL 표현으로 변환된 프로그램에 대하여 레지스터 및 메모리의 내용을 추적하여 실제 메모리 접근이 이루어지는 주소를 알아낸다. 프로그램에 따라서는 이러한 방법으로 메모리 접근의 주소를 알아낼 수 없는 경우도 있다. 따라서, 본 연구에서는 모든 메모리 접근의 주소를 분석을 통해 '확인 가능한 주소'와 '확인 불가능한 주소'로 나눈다. 분석을 통해 모든 메모리 접근 주소가 '확인 가능한 주소'로 판명되고 해당 주소들이 모두 합법적인 접근 가능한 주소로 판명되어야만 안전한 프로그램이라고 판단한다.

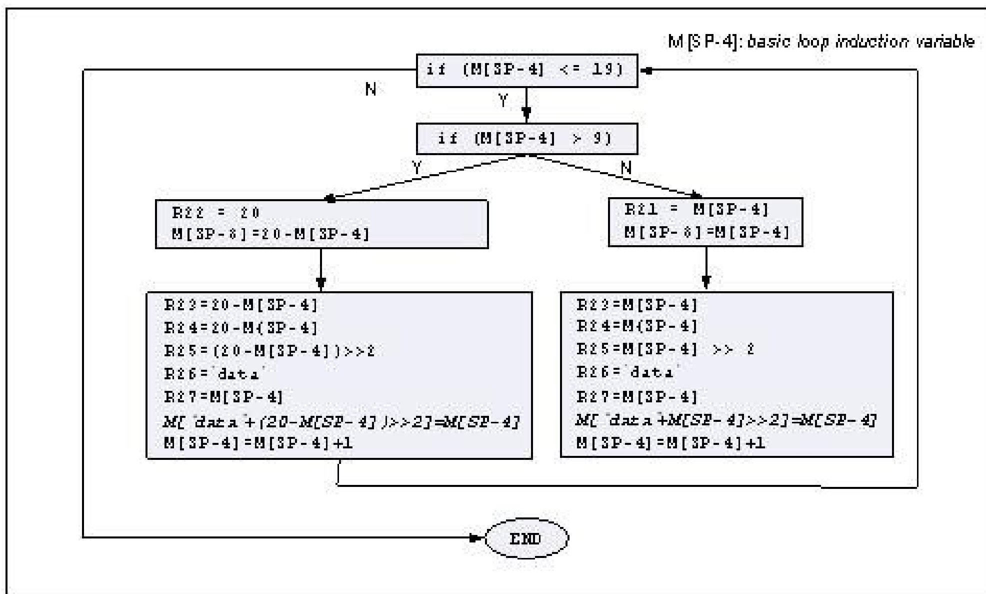
4.3 기호적 실행 방법

본 논문에서 제안하는 기호적 실행에 의한 프로그램 분석 기법은 흐름 제어 그래프 생성 및 RTL 구성 단계, 레지스터 초기화 단계, 그리고 기호적 실행 및 RTL 확장 단계 등으로 구성된다.

첫째, 흐름 제어 그래프 생성 및 RTL 구성 단계에서는 분석 대상 프로그램에 대한 흐름 제어 그래프를 완성하고 모든 프로그램 내의 구문을 RTL 구문으로 바꾼다. RTL 구문은 앞 절에서 소개한 바와 같이 어느 구문에서 메모리 접근이 일어나는지를 알 수 있는 형태의 구문이다. 첫 번째 단계에서 수행하는 중요한 두 가지 작업은 프로그램 내의 순환문 구조를 모두 파악하는 것과전역 변수 및 지역 변수, 그리고 스택 영역을 위한 메모리 크기를 모두 파악하는 것이다. 순환문 구조를 파악하는 목적은 순환문의 순환에 사용되는 베이스 레지스터와 순환 지표 레지스터(loop induction variable)를 파악하기 위해서이다. 이들 레지스터를 파악하면, 순환문 내에서 반복적으로 수행되는 메모리 접근의 대상 주소가 순환에 의해 일정하게 변하는 것을 추적할 수 있게 된다.



(a) (b)



(c)

그림 3 순환문의 순환 추적이 필요 없는 경우
Fig. 5 A case that does not need loop iteration tracking

둘째, RTL 표현 내의 모든 레지스터들 중 초기에 그 값을 알 수 없는 것들을 모두 'unknown' 값을 가지는 것으로 초기화한다. 이 레지스터들은 기호적 실행이 일어남에 따라 점차 확인 가능한 값으로 변경된다.

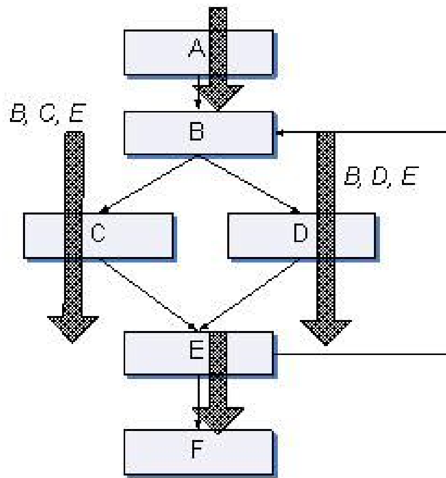


그림 4 순환문을 포함한 프로그램의 실행 경로
Fig. 4 Control flows for a program including loops

셋째, 첫 번째 단계에서 생성된 흐름 제어 그래프를 따라 기호적 실행을 수행하면서 RTL 구문을 '확인 가능한 값'들의 조합으로 확장한다. 즉, 상수나 순환문 베이스 레지스터 및 순환문 순환 지표 레지스터, 그리고 외부로부터 전달되어 온 레지스터 값 등의 선형 조합으로 변경한다.

기호적 실행은 프로그램의 제어 흐름 그래프(control flow graph: CFG)를 따라가면서 각 메모리 접근 주소를 파악하는 시뮬레이션 성격의 분석 작업이기 때문에 순환문을 포함할 경우 매 순환을 따라가야 하는 상황을 피해야 분석의 복잡도를 줄일 수 있다. (그림 4)는 서로 다른 두 개의 실행 경로를 가지고 있는 순환문이 포함된 프로그램의 CFG를 보인다. 이 프로그램의 순환문 내에는 그림에 나타나 있듯이 B, C, E의 순서로 실행되는 실행 경로와 B, D, E의 순서로 실행되는 실행 경로가 존재한다. 순환문의 순환이 거듭됨에 따라 서로 다른 실행 경로의 개수는 기하 급수적으로 증가하게 된다. 따라서, 이 순환문의 모든 실행 경로를 따라 가면서 기호적 실행을 수행하는 것은 순환 회수가 커지게 되면 지나치게 큰 복잡도를 보이게 된다. 따라서, 이러한 순환문 내에 포함된 구문에 메모리 접근이 있을 경우 순환 회수만큼 순환을 직접 따라 가지 않고 메모리 접근 주소를 파악할 수 있는 방법이 필요하다.

(그림 5)에서는 두 개 이상의 서로 다른 실행 경로가 포함된 순환문 내에 메모리 접근 구문이 있는 경우 해당 구문에서 접근하는 메모리 접근 주소를 파악하는 절차를 본 연구에서 사용하는 RTL 표현을 통해 보인다. (그림 5(a))는 순환문 내에 존재하는 메모리 접근을 처리하는 방법을 보이기 위한 예제 프로그램이다. 예제 프로그램은 하나의 순환문으로 구성되어 있고 순환문 내에는 두 개의 서로 다른 경로가 있다. 메모리 접근이 일어나는 구문은 "data[x] = i;" 인데 이 구문에서 메모리 주소를 결정짓는 변수 x의 값은 이 구문에 도달하는 경로에 따라 다르게 할당된다. (그림 5(b))는 (그림 5(a))의 예제 프로그램을 CFG로 표현하고 CFG의 각 기본블록 내에는 RTL 표현으로 바뀐 구문들을 보인다. 굵게 표시된 구문 'M[R26+R25]=R27'이 메모리 접근 구문에 해당한다. 이 외의 구문들은 모두 레지스터를 접근하거나 스택에 할당된 변수를 접근하는 구문들이다. 구문 'M[R26+R25]=R27'이 확인 가능한 주소를 접근하는지, 아니면 확인 불가능한 주소를 접근하는지는 R26+R25의 값이 확인 가능한 값인지 아닌지에 따라 결정된다. 따라서, 레지스터 R26과 레지스터 R25의 값이 프로그램의 시작부터 어떻게 결정되는지 추적할 필요가 있다.

(그림 5(c))는 (그림 5(b))의 RTL 표현을 CFG를 따라 가면서 각 레지스터의 내용을 추적하여 메모리 접근 주소를 확장한 RTL 표현이다. M[sp-4]는 순환문의 순환에 따라 증가하는 순환 지표 변수이다. (그림 5(c))의 메모리 접근 구문의 주소는 RTL 표현에 나타나 있듯이 모두 확인 가능한 변수나 상수 값으로 이루어져 있어서 '확인 가능한 주소'이다. 그리고, 이 예제 프로그램에서는 순환문의 순환 회수를 프로그램 실행 전에 정적으로 파악할 수 있고 메모리 접근 주소 또한 순환문의 순환 진행을 매번 따라가지 않아도 정적으로 주소의 범위를 파악할 수 있다. 따라서, 이러한 형태의 프로그램을 대상으로 할 때에는 순환문이 포함되어 있더라도 순환을 직접 추적하는 기호적 실행을 수행할 필요없이 CFG를 구성하기 위한 한번의 기호적 실행만으로 메모리 접근 주소 범위를 파악할 수 있다.

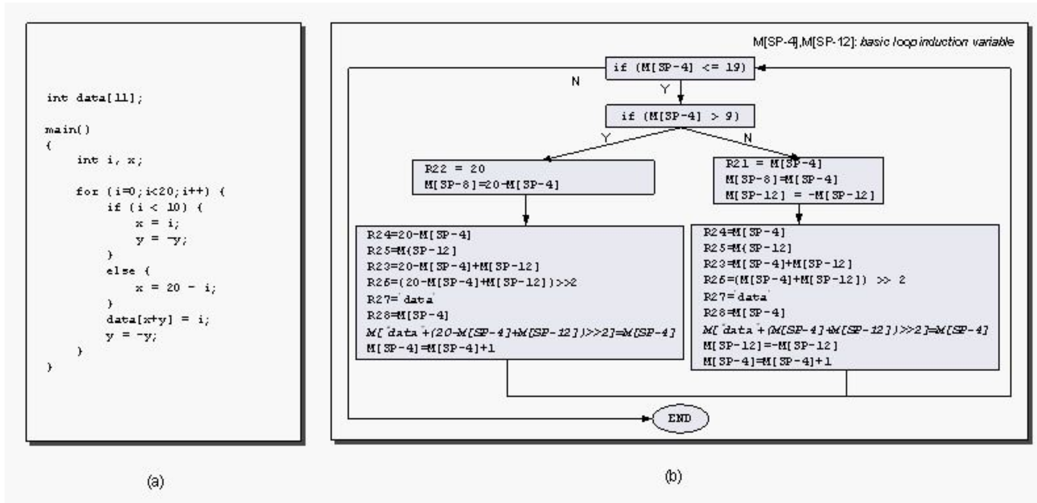


그림 5 순환문의 기호적 실행에 의한 메모리 접근 주소의 파악
Fig. 6 Symbolic execution for a loop to identify memory access addresses

(그림 6)은 (그림 5)와 달리 메모리 접근 구문의 메모리 접근 주소를 순환문의 순환을 직접 진행해보지 않으면 파악할 수 없는 프로그램의 예이다. (그림 6(a))는 C언어로 작성된 프로그램을 보이고, (그림 6(b))는 이 프로그램을 RTL 표현으로 바꾼 CFG를 보인다. 이 프로그램의 순환문 내에서는 순환에 따라 일정하게 변하는 순환 지표 변수인 x와 순환에 따라 변하지만 규칙적으로 변하지 않는 y가 메모리 접근 주소 결정에 영향을 준다. 따라서, 순환문의 순환 시작부터 매 순환마다 변수 x와 y의 변수 값의 변화에 따른 메모리 접근 주소 값의 변화를 추적해야 한다. 그러나, 순환이 진행됨에 따라 주소값이 일정한 범위 내에 존재함이 밝혀지면 순환문의 순환을 중단하고 메모리 접근 주소의 범위를 결정할 수 있게 된다.

V. 응용 및 실험

본 장에서는 본 논문에서 제안하는 기호적 실행에 의한 프로그램의 메모리 안전성 분석을 적용한 웹 기반 실시간 시스템 제어 환경을 소개하고 응용 사례를 설명한다. 또한, 본 논문에서 제안하는 방법을 구현한 컴파일러 및 분석기 환경에서 주요 실시간 응용 프로그램들을 대상으로 제안하는 방법의 효용성을 검증한 결과를 보인다.

5.1 Simplex 기반 Telelab 환경

본 논문에서 제안하는 메모리 안정성 확보를 통한 실시간 시스템의 보안 기법은 미국 University of Illinois, Urbana Champaign의 Real Time Systems Laboratory에서 개발한 인터넷을 통한 실시간 제어 시스템인 Telelab[2]에 적용되었다. Telelab은 실시간 시스템을 위한 제어 프로그램의 동적 업그레이드 아키텍처인 Simplex[10]를 기반으로 하여 구현되었다.

Simplex는 Sha 등에 의해 제안된 동적 시스템 업그레이드 기법으로서 시스템을 정지시키지 않고 시스템을 위한 제어 알고리즘을 변경시킬 수 있는 환경을 제공한다. Simplex에서는 대상 시스템이 안정적으로 동작하는 제어 인자의 범위인 안전 영역(safety envelope)을 정의한다. 이 안전 영역을 벗어난다는 것은 대상 시스템이 제 역할을 수행할 수 없는 위험한 상태가 된다는 것을 의미한다. Simplex에서는 동적으로 시스템에 전달되어 업그레이드되는 제어 코드에 의해 시스템의 각종 인자가 안전 영역을 벗어나는 경우에는 새로 전달된 제어 코드의 실행을 포기하고 자동적으로 기존 안정적인 제어 코드를 실행하게 만든다. 따라서, Simplex 기반 시스템에는 시스템의 안정성을 항상 보장해주는 안정적인 제어 코드가 상주한다.

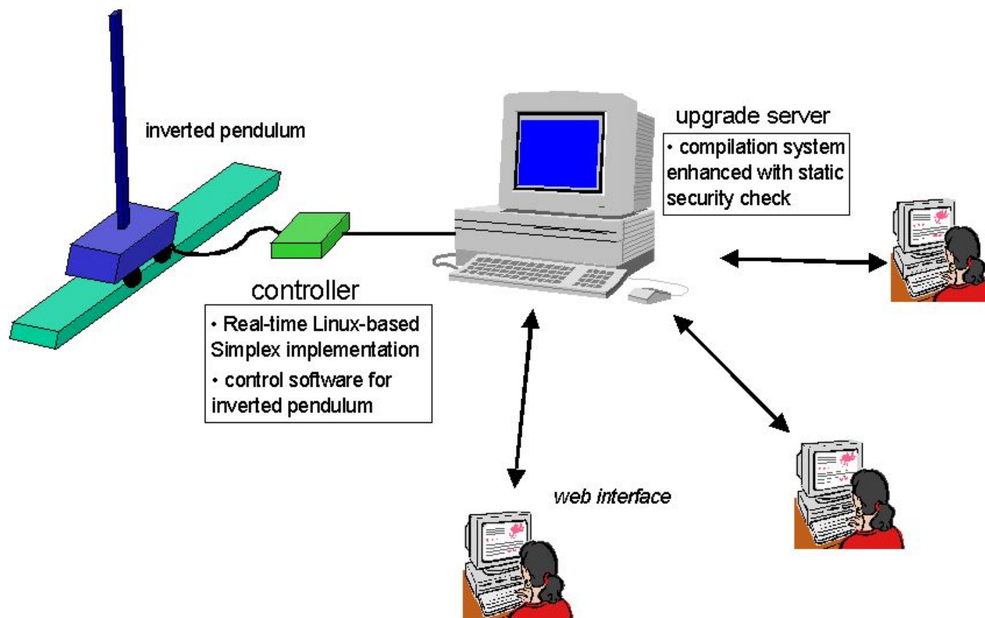


그림 6 Telelab 기반 원격 실시간 시스템 제어 환경
Fig. 7 Telelab-based remote real-time system control environment

Telelab은 Simplex를 실제로 구현하여 실시간 시스템에서의 응용 가능성을 검증하기 위해 구축된 원격 실험 환경이다. Telelab에서는 역진자(inverted pendulum)의 균형을 맞추어주는 시스템을 원격으로 제어하는 시스템을 사용한다.

(그림 7)은 역진자를 이용한 Telelab 환경의 구성을 보인다. 이 실험 환경은 현재 University of Illinois at Urbana Champaign 내 Real Time Systems Laboratory에 설치되어 있으며 인터넷이 연결된 원격지에서 접속하여 역진자를 제어하기 위한 실시간 프로그램을 업그레이드할 수 있다. 역진자를 제어하는 제어 하드웨어는 i486 계열의 AM5x86-133 프로세서를 기반으로 한 단일 보드 컴퓨터이고 AD/DA 컨버터를 장착하고 있어서 이를 통해 역진자를 제어한다. 운영체제는 리눅스 커널 2.2에 실시간 기능을 강화하기 위한 수정을 하여 사용하였다. 실시간 기능 강화를 위한 수정이란 역진자 제어를 무리 없이 수행할 수 있도록 인터럽트 핸들러의 실행 시간을 제한한 것이다. 실시간 기능 강화를 위

해 수정한 리눅스 커널 상에 Simplex를 구현한 Daemon을 구현하여 탑재하였는데, 크기는 약 20KB이다.

5.2 정적 프로그램 분석의 효용성 검증

본 논문에서 제안하는 기법의 효용성을 검증하기 위해서 두 가지 실험을 수행하였다. 첫째는 앞 절에서 설명한 Telelab 환경에서 제어 코드 내에 악의적인 해킹 코드를 여러 가지 방법으로 삽입하여 이를 본 논문에서 제안하는 방법을 구현한 컴파일러에서 차단되는 지에 대한 실험이고, 둘째는 실시간 시스템에 실제로 많이 사용되는 제어 코드들 내에 포함된 메모리 접근 구문 중 '확인 가능한 주소'로 결정되는 구문의 비율을 파악하는 방법이다.

첫째 실험을 위해 Simplex 시스템 내에 본 논문에서 제안하는 분석 기법을 구현한 컴파일러를 탑재하여 인터넷을 통해 전달되는 제어 코드를 컴파일하는 과정에서 메모리 접근의 안전성을 파악한다. 컴파일러는 GCC 2.95.2를 사용하고 RTL 수준에서 기호적 실행 방법에 의해 메모리 안전성을 분석하는 부분을 추가하였다. 기존 GCC의 옵션에 '-fsecurity-check'라는 옵션을 추가하여 이 옵션을 인자로 입력하여 GCC를 수행하면 컴파일 대상 프로그램의 메모리 안전성을 분석하게 된다. 이 분석 부분은 하드웨어에 독립적인 RTL을 기반으로 하고 있기 때문에 다양한 프로세서에

적용될 수 있다. 현재, x86 계열의 프로세서, ARM 계열의 프로세서, 그리고 PowerPC 계열의 프로세서를 위한 GCC에서 테스트를 완료하였다.

본 논문에서는 실시간 시스템에서 주로 사용되는 알고리즘을 중심으로 구성된 UTDSP 벤치마크(University of Toronto DSP benchmark) 및 기타 주요 실시간 시스템을 위한 벤치마크 프로그램들에 포함된 모든 메모리 접근 구문을 대상으로 하여 제안하는 분석 도구로 분석을 수행하였을 때 '확인 가능한 주소'로 판명되는 메모리 접근 구문의 비율을 파악하였다.

<표 1>은 본 연구에서 사용한 벤치마크 프로그램을 보인다. 사용한 벤치마크 프로그램은 UTDSP 벤치마크에서 추출하였다. 본 논문에서 제안하는 기법을 구현한 GCC를 사용하여 <표 1>에 나열된 벤치마크 프로그램을 컴파일할 경우 컴파일 과정 중 수행하는 기호적 실행에 의한 메모리 안전성 분석에 의해 총 메모리 쓰기 동작의 회수와 메모리 쓰기 동작 중 '확인 가능한 주소'로 판별되는 메모리 쓰기 동작의 회수를 출력한다. <표 1>에 나타난 바와 같이 본 논문에서 선정한 벤치마크 프로그램 5가지의 경우에는 모든 메모리 쓰기 동작에 대한 메모리 접근 주소가 '확인 가능한 주소'로 결정되었다. 또한, 해당 프로그램들의 경우 순환문의 순환을 따라가지 않아도 순환문에 포함된 모든 메모리 접근 주소를 파악할 수 있는 경우로서 기호적 실행에 의한 추가적인 비용이 거의 발생하지 않았다. <표 1>에 나타나 있지 않은 다른 프로그램의 경우, 대부분의 실시간 제어용, 혹은 DSP 알고리즘을 구현한 프로그램에 포함된 메모리 접근 주소를 확인 가능하였다.

본 실험 결과로 파악할 수 있는 것은 본 논문에서 제안하는 기호적 실행에 의한 메모리 안전성 분석 기법이 실시간 제어 프로그램에 대해 적용 가능하며 효용성이 있다는 것이다.

표 1 벤치마크 프로그램 리스트
Table 1 Benchmark programs

벤치마크 프로그램	프로그램 설명	메모리 쓰기 동작 회수	확인 가능한 메모리 접근 수
FFT	1024 개 복소수 배열에 대한 FFT 연산	44,084	44,084 (100%)
LMSFIR	64 포인트에 대한 32 tap LMS FIR 필터	12,621	12,621 (100%)
MULT	10 x 10 행렬의 행렬 곱셈	2,448	2,448 (100%)
COMPRESS	128 x 128 이미지에 대한 DCT를 이용한 이미지 압축 알고리즘	2,123,787	2,123,787 (100%)
EDGE_DETECT	256 개의 gray level 128 x 128 이미지에 대한 edge detection 알고리즘	3,847,589	3,847,589 (100%)

본 논문에서 제안하는 기법은 다음 두 가지 개선점이 있다. 첫째, 순환문의 순환을 모두 따라가는 경우 기호적 실행에 걸리는 시간 및 각각의 실행 경로를 모두 따라가는 데 필요한 비용이 상당히 커질 수 있다. 그러나, 순환문의 순환을 거듭하면서 모든 순환문 내에 포함된 경로를 따라갈 필요 없이 제한된 수의 실행 경로만 따라가면 되는 경우가 대부분이기 때문에 기하급수적으로 늘어나는 분석 비용을 초래하지는 않는다. 둘째, 프로그램 내에 여러 개의 함수가 포함되어 있는 경우, 함수의 인자로 전달된 데이터에 의해 메모리 접근 주소가 결정되는 경우, 현재 기법에서는 해당 접근 주소를 '확인 불가능한 주소'로 인식한다. 이는 함수간 인자 전달 시 각 인자별로 이미 확인 가능한 값으로 결정된 인자인지 아닌지를 별도의 정보로 만들어 함께 전달하는 방법으로 함수간 분석 방법을 보완하는 방법으로 해결할 수 있다.

Ⅵ. 결론

본 논문에서는 실시간 제어 프로그램에 대한 정적 메모리 안전성 분석 기법을 제안하고 이를 구현한 분석 도구에 의해 효용성을 검증하였다. 제안하는 정적 메모리 안전성 분석 기법은 기호적 실행에 의해 메모리 접근 주소를 파악하여 메모리 접근의 안전성을 파악하는 것으로 스택 오버플로우에 의한 해킹 및 함수 포인터의 악의적인 사용에 의한 해킹 등의 공격을 차단할 수 있는 방법이다. 제안하는 분석 방법을 실제 인터넷 기반 원격 실시간 제어 시스템에 적용하여 효용성을 검증한 결과 모든 악의적인 메모리 접근 사용을 차단할 수 있었다. 또한, 실시간 시스템용 벤치마크 프로그램을 대상으로 확인 가능한 메모리 접근 주소의 비율을 조사한 결과 대부분의 메모리 접근 주소를 파악할 수 있었다.

참고문헌

[1] UTDSP Benchmark Suite. http://www.eecg.toronto.edu/ conrinna/DSP/ infrastructure/UTDSP.html.

[2] Janek Schwarz, Andreas Polze, Kristopher Wehner, and Lui Sha. Remote lab: A reliable telel aboratory environment. In International Conference on Internet Computing, pages 55 62, 2000.

[3] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single Address Space Operating Systems. ACM Transactions on Computer Systems, 12(4):271 307, Nov. 1994.

[4] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer Overfow Attacks. In Proceedings of the 7th USENIX Security Symposium, pages 63 78, Jan. 1998.

[5] R. W. M. Jones and P. H. J. Kelly. Backwards Compatible Bounds Checking for Arrays and Pointers in C Programs. In Proceedings of the Third International Workshop on Automated Debugging, 1997.

[6] T. Lundqvist and P. StenstrÄom. Integrating Path and Timing Analysis Using Instruction Level Simulation Techniques. In Proceedings of 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, June 1998.

[7] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[8] G. C. Necula. Proof Carrying Code. In Proceedings of the 1997 Symposium on Principles of Programming Languages (POPL97), Jan. 1997.

[9] R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In Proceedings of the SIGPLAN'2000 Conference on Programming Language Design and Implementation, pages 182 195, June 2000.

[10] L. Sha. Dependable System Upgrade. In Proceedings of the 19th Real Time Systems Symposium, pages 440 448, 1998.

[11] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. Operating Systems Review, 34(5):170 185, Dec. 1999.

[12] J. Vochtello, S. Russel, and G. Heiser. Capability based Protection in the Mungi Operating System. In Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems, pages 108 115, 1993.

[13] D. A. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Network and Distributed System Security Symposium, Feb. 2000.

[14] Z. Xu, B. P. Miller, and T. Reps. Safety Checking of Machine Code. In Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'2000), June 2000.

[15] P. Koopman. Embedded System Security, IEEE Computer, 37(7):95 97, July 2004.

[16] 고영웅, 보안 운영체제의 오버헤드 분석, 한국컴퓨터정보학회 논문지, 10(2), 2005

[17] 고영운, 보안 운영체제를 위한 강제적 접근 제어 보호 프로파일, 한국컴퓨터정보학회 논문지, 10(1), 2005

저 자 소 개

임 성 수

2004.3.~현재: 국민대학교 컴퓨터 학부 전임강사

1999~2004.2.: 팜팜테크(주) 기술 이사

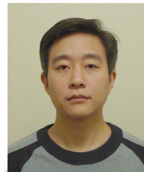
2002.2.: 서울대학교 전기컴퓨터공학 박사

2000.8.~2001.8.: University of Illinois at Urbana-Champaign 방문 연구원

1995.2: 서울대학교 컴퓨터공학 석사

1993.2: 서울대학교 컴퓨터공학 학사

<관심분야> 임베디드 시스템, 이동통신 단말, 실시간 시스템



Kihwal Lee

현재: PhD candidate, Department of Computer Science, University of Illinois at Urbana-Champaign.

1998: BS in Computer Science from Southern Illinois University at Carbondale

<관심분야> 임베디드 실시간 시스템, Robust software system