C 프로그램을 위한 효율적인 버퍼 오버런 분석기의 개발

Development of Cost-effective Buffer Overrun Analyzer for C Programs

김유일 · 전진성 · 한환수
KAIST 전자전산학과 전산학전공
{youil.kim; jinseong.jeon; hwansoo.han}@arcs.kaist.ac.kr

요 약

C 언어로 작성된 소스 코드를 분석하여 버퍼 오버런 취약점을 실행 전에 발견하는 효율적인 정적 분석 도구를 개발하는 과정에서, 속도와 정확도라는 두 가지 상충하는 요구 사항을 모두 만족시키기 위한 분석기의 구조를 제안하고, 현재까지의 실험 결과를 소개한다. 제안하는 분석기의 구조는 포인터 분석과 값 분석을 별도의 단계로 구분하여 정확도를 다르게 설정할 수 있는 구조로, 리눅스 커널 소스에 대해서는 부정확한 포인터 분석과 정확한 값 분석의 조합이 정확한 포인터 분석과 정확한 값 분석의 조합에 비해 빠른 속도로 수행되면서도 정확도 면에서 비슷하다는 실험 결과를 통해 이러한 구조의 가능성을 보인다.

1 서론

버퍼 오버런(buffer overrun) 또는 버퍼 오버플로우(buffer overflow)는 프로그램이 미리 할당된 버퍼의 경계를 지나 데이터를 읽거나 쓰는 것을 말한다. 버퍼의 경계를 지나 데이터를 기록하면 다른 변수의 값에 영향을 미치거나 프로그램의 제어 흐름을 바꿀 수도 있는데, 이는 찾기 어려운 버그의 원인이 되거나 일반적인 수행에서 문제가 발견되지 않더라도 소프트웨어의 심각한 보안 구멍으로 남는다. 실제로 버퍼 오버런 취약점은 소프트웨어가 보안에 취약하게 만드는 대표적인 원인으로서 알려진 소프트웨어 보안 취약점의 상당수가 버퍼 오버런과 관련되어 있다[6]. 예를 들어 1998 년의 Morris 웜은 유닉스 `fingerd` 소프트웨어의 버퍼 오버런 취약점을 이용한 것이었고, 2001 년의 Code Red 바이러스는 마이크로소프트 윈도우의 웹 서버인 IIS 소프트웨어의 버퍼 오버런 취약점을 이용한 것이었다.

본 연구의 최종 목적은 C 언어로 작성된 소스 코드를 정적으로 분석하여 버퍼 오버런 취약점을 조기에 발견할 수 있는 실제적인 분석 도구를 개발하는 것이다. 분석 도구가 실제로 활용 가능한 수준이 되기 위해서는 속도와 정확도라는 두 가지 요구 사항을 모두 만족시켜야 한다. 기존의 연구를 살펴보면 대개 분석의 속도와 정확도는 반비례 관계에 있으므로, 이러한 두 가지 요구 사항은 상충하는 것처럼 보인다. 연구자들은 먼저 기존의 분석 도구[7]가 리눅스 커널 2.6.4의 일부 소스 파일에서 찾아낸 버그들을 살펴보았다. 분석 도구가 찾아낸 버그들은 비교적 간단한 분석 기법만을 사용하더라도 역시 찾아낼 수 있을 것으로 보였는데, 연구자들은 모든 소스 코드에 대해서 복잡하고 정확한 분석 기법을 적용하는 것은 비효율적이라는 판단을 내렸다. 다시 말해, 적어도 일부 소스 코드에 대해서는 간단한 분석 기법만으로도 요구되는 정확도에 도달할 수 있을 것으로 예측할 수 있다. 이러한 예

측에 따라, 간단한 분석 기법으로 분석을 시작하되 점진적으로 정확하고 복잡한 분석 기법을 적용할 수 있는 유연한 구조의 분석기를 개발하기로 결정했다.

본 논문의 이후 구성은 다음과 같다. 제 2 장은 기존의 버퍼 오버런 분석기들의 특성과 장단점을 간략히 소개한다. 제 3 장은 예제를 통해 문제를 조금 더 자세히 살펴본다. 제 4 장과 제 5 장에 걸쳐 제안하는 분석기의 구조와 현재까지의 구현 내용을 설명한다. 제 6 장에서는 실험 결과를 통해 제안하는 구조의 가능성을 보인다. 제 7 장에서 결론을 맺으며 향후 연구 방향에 대해 논의한다.

2 관련 연구

버퍼 오버런 취약점을 발견하는 실제적인 도구로서는 먼저 D. Wagner 등[11]의 연구를 들 수 있다. D. Wagner 등이 개발한 분석기는 3 만 라인 이상의 실제 코드[1]를 15 분 이내에 분석하여 알려지지 않은 취약점을 찾아낼 수 있었다. 그러나 D. Wagner 등의 분석기는 프로그램의 제어 흐름과 함수 호출, 그리고 포인터 연산을 모두 부정확하게 모델링하여 다수의 잘못된 경고 메시지(false alarm)[2]를 출력하며 실제 오류를 놓치는 경우도 있다. 참조한 논문에서는 만일 프로그램의 제어 흐름과 포인터를 정확하게 모델링하면 잘못된 경고 메시지를 63% 이상 걸러낼 수 있는 여지가 있음을 지적하고 있다. 본 연구가 D. Wagner 등의 연구와 차별화되는 점은, 값에 대해서는 프로그램의 제어 흐름을 고려하는 분석 기법을 사용하고, 포인터에 대해서는 프로그램의 제어 흐름을 무시하는 분석 기법을 사용하여 분석 속도를 크게 저하시키지 않으면서도 정확도를 향상시킬 수 있었다는 점이다.

N. Dor 등[6]의 C String Static Verifier (CSSV)는 주어진 프로그램에 버퍼 오버런 취약점이 존재하지 않음을 검증하기 위한 도구를 개발하려는 목표를 가지고 비교적 정확한 분석 기법을 사용한 예이다. CSSV가 기존의 연구와 차별화되는 점은 버퍼를 문자열로 사용하는 경우에는 널 문자의 위치, 즉 문자열의 길이라는 요소가 중요함을 인식하고 이를 정확하게 모델링하여 정확도를 향상시키고 버퍼 오버런 취약점 이외에도 문자열 사용이 잘못된 경우까지 지적해 줄 수 있다는 점이다. CSSV는 각 변수들이 가질 수 있는 값의 범위 뿐만 아니라 변수들 사이의 관계까지도 분석할 수 있는 분석 기법[3]을 사용했다. 이 분석은 복잡도가 높아 많은 분석 시간을 요구하는 것으로 알려져 있지만, CSSV는 주어진 프로그램을 각 함수별로 따로 분석할 수 있도록 설계함으로써 좋은 성능을 얻었다. 그러나 사용자의 개입 없이 수행하는 경우에는 역시 다수의 잘못된 경고 메시지를 출력하는 단점이 남아 있다.

A. Venet 등[10]의 C Global Surveyor (CGS)는 NASA에서 개발하는 크고 복잡한 소프트웨어를 대상으로 하여 버퍼 오버런 오류 검증을 수행하려는 목적으로 개발된 분석기이다. 참조한 논문에서 주장하는 주된 성과는 대상 프로그램에 특화된 분석 기법을 설계하여 기존의 분석기와 동일한 정확도를 유지하면서도 분석 속도를 비약적으로 향상시키는 것이 가능하다는 것을 보인 것이다. 발표된 결과에 따르면 CGS는 28 만 라인에 달하는 코드를 두 시간 이내에 분석할 수 있었다.

최근에 국내에서도 정영범 등[7]이 버퍼 오버런 취약점을 발견할 수 있는 Airac 이라는 분석 도구를 개발했다. 포인터와 값을 동시에 분석하므로 포인터 변수와 일반 변수의 사용이 교묘하게 얽힌 프로그램에 대해서도 정확하게 분석할 수 있을 것으로 보인다. 분석 정보를 병합하는 연산을 최적화하는 등 다양한 최적화 아이디어를 동원하여 좋은 성능을 얻었고, 확률적인 기법을 사용하여 잘못된 경고 메시지를 걸러내는 새로운 방식을 시도하였다. 본 연구에서는 리눅스 커널 2.6.4의 일부 소스 코드에 대해서는 포인터를 정확하게 분석하지 않더라도 Airac과 비슷한 정도의 정확도를 유지한다는 것을 보였다. 또한 본 연구의 방

[1] `Sendmail 8.9.3`

[2] 실제 오류가 아니지만 오류라고 지적하는 것

```
void f(void)
{
    int x[100], i = 0, j = 0;

    while (i < 99) {
        i++;
        j++;
    }
    x[j] = 0;
}
```

```
void g{void)
{
     int x[100], i = 0, j = 100,
     int *p;

     p = &i;
     x[*p] = 0;
     p = &j;
}
```

[그림 1] 분석의 정확도가 중요한 프로그램의 예

향이 Airac과 차별화되는 점은 연구자들은 부분적으로 보다 정확한 분석 기법을 적용하는 방법을 통해 잘못된 오류 메시지를 걸러내려는 계획을 가지고 있다는 것이다.

3 문제의 특성 및 해결 방안

문제를 대략적으로 설명하면 다음과 같다. C 언어로 작성된 프로그램에서 버퍼 오버런 취약점을 찾아내려면 `x[i]` 또는 `*(x + i)`와 같은 형태의 식을 살펴보아야 한다. 이런 형태의 식에서 `x`는 배열 이름일 수도 있고 버퍼의 시작 주소를 가지는 포인터일 수도 있다. 이런 형태의 식이 버퍼 오버런을 발생시키는지의 여부를 파악하려면 `x`가 의미하는 배열 또는 버퍼의 크기를 알아야 하고 실행 중에 `i`가 가질 수 있는 값의 범위를 알아야 한다.

따라서 각 변수들이 실행 중에 가지는 값을 분석하는 방법을 결정해야 한다. 변수들이 실행 중에 가지는 값을 요약하는 방법으로는 인터벌 분석(interval analysis)[1]에서 다각형 분석(polyhedra analysis)[3]에 이르기까지 다양한 방법이 연구되어 있다. 인터벌 분석은 각 변수가 가질 수 있는 최소값과 최대값을 계산하는 방법이고, 다각형 분석은 각 변수들 사이의 모든 선형 관계식을 계산하는 방법이다. 결과의 정확도가 높을수록 분석 시간과 메모리 요구량이 증가한다.

C 언어로 프로그램을 분석하려는 경우에는 포인터를 분석하는 것이 중요하다. 버그를 놓치지 않는 안전한 분석기를 설계하려는 경우에는 포인터를 정확하게 분석해야 잘못된 경고 메시지를 줄일 수 있다. 예를 들어 `x[*p]`라는 식에서 `p`가 가리키는 대상을 모른다면 이 식에서 버퍼 오버런이 발생할 수 있다는 판단을 하게 된다. `*p`에 값을 대입하는 경우는 더욱 심각하다. `p`가 가리키는 대상을 모른다면 프로그램에 존재하는 모든 변수를 잠재적인 대상으로 보아야 한다.

포인터 분석(points-to analysis)은 프로그램 실행 중에 각 포인터가 가리킬 수 있는 대상을 결정하는 단순한 문제이지만, 정확하게 분석하려면 계산의 복잡도가 높아 분석 속도가 문제가 된다. B. Steensgaard[9]는 기존의 포인터 분석 기법이 10 만 라인 이상의 프로그램을 분석하기 어렵다는 것을 지적하고, 정확도를 다소 희생하여 거의 선형 시간 복잡도를 가지는 새로운 분석 기법을 제안하였다.

위에서 언급한 것처럼 값을 분석하는 경우든 포인터를 분석하는 경우든 현재 서로 다른 정확도를 가지는 다양한 분석 기법이 제안되어 있으나, 분석 시간과 분석의 정확도 사이에서 균형을 잡는 것이 어려운 문제이다. 부정확한 분석 기법을 사용하면 잘못된 경고 메시지를 출력할 가능성이 있는 프로그램의 예를 그림 1에 나타내였다. 함수 `f`의 경우 변수 `i`와 변수 `j` 사이의 관계를 무시하는 분석 기법이 축지법(widening) 및 좁히기(narrowing) 기법과

함께 사용되면 변수 j의 값을 올바르게 계산하지 못할 가능성이 있다. 그러나 반복문을 99회 이상 정확하게 추적하거나 변수 i와 변수 j 사이의 관계를 유지하는 분석 기법을 사용한다면 함수 f에 버퍼 오버런 가능성이 없음을 검증할 수 있다. 함수 g의 경우 프로그램의 제어 흐름을 무시하는 (flow-insensitive) 포인터 분석 기법을 사용한다면 포인터 변수 p를 참조하는 위치에서 포인터 변수 p가 가리키는 대상이 변수 i라는 것을 정확히 파악하지 못해 잘못된 경고 메시지를 출력할 가능성이 있다. 그러나 보다 정확한 포인터 분석 기법을 사용한다면 함수 g에 버퍼 오버런 가능성이 없음을 검증할 수 있을 것이다.

그림 1의 코드는 가능성을 보여주기 위하여 의도적으로 작성된 것인데, 연구자들은 실제 소프트웨어 코드에 이러한 패턴이 빈번하게 등장할 가능성에 대해서는 회의적이다. 적어도 리눅스 커널 2.6.4의 일부 소스 코드는 비교적 단순한 분석 기법들을 조합하는 것만으로도 비교적 정확하게 분석할 수 있을 것으로 보인다. 연구자들은 우선 프로그램의 제어 흐름을 무시하는 포인터 분석과 변수들의 관계를 무시하는 값 분석 기법의 조합으로 분석기를 구성하여 실험하였다. 다만 경우에 따라 포인터 분석 기법 및 값 분석 기법의 정확도를 별도로 조절할 수 있도록 분석기를 설계하여, 추후 복잡한 프로그램을 분석해야 하는 상황에 대응할 준비를 갖추었다.

4 분석기의 설계

그림 2는 제안하는 분석기의 전체적인 구조를 나타낸 것이다. Parse 단계에서는 주어진 C 프로그램을 파싱하여 각 함수별로 제어 흐름 그래프를 구성하고, 이후의 단계에서는 제어 흐름 그래프를 바탕으로 분석을 진행한다. 타입 정보를 유지하여 `sizeof` 등을 정확하게 분석하는데 이용한다. 그림 2에 나타낸 것처럼 Points-to Analysis 단계와 Value Analysis 단계를 구분하는 구조의 분석기는 포인터와 값을 동시에 분석하는 분석기에 비해 부정확한 결과를 얻게 될 가능성이 있지만, 연구자들은 아직 실제 소프트웨어에서 그러한 경우를 발견하지는 못했다.

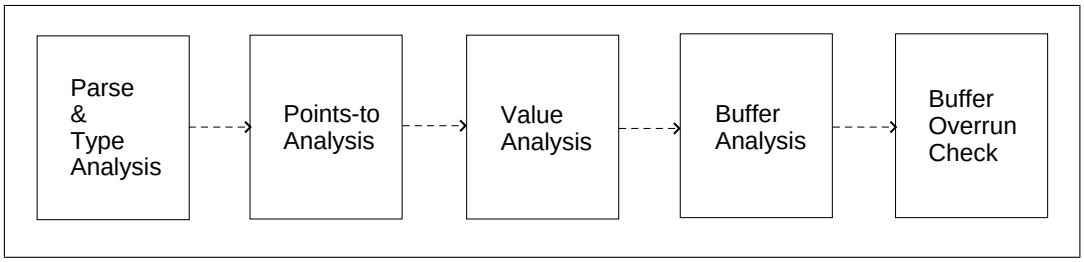
Points-to Analysis 단계는 `*p`, `**p` 등등 가능한 형태의 좌변식에 대해서 좌변식이 의미하는 메모리 영역에 대한 정보를 계산한다. 수학적으로는 요약된 메모리 영역(abstract location)의 집합을 계산하는데, 각각의 요약된 메모리 영역은 하나 이상의 변수 또는 실제 메모리 영역에 대응된다.

Value Analysis 단계는 각 변수가 가질 수 있는 값을 계산한다. 보다 정확하게 말하면 Points-to Analysis 단계의 결과를 사용하여 각 변수 또는 포인터 참조에 대응하는 요약된 메모리 영역을 구하고, 각각의 요약된 메모리 영역이 가질 수 있는 값을 계산하는 것이다. 현재의 구현은 요약 해석 프레임워크[1, 2]에 기반하여 인터벌 분석을 수행하는데, 제어 흐름 그래프의 각 노드에 대해서, 가능한 메모리 상태를 정수 변수를 기준으로 요약한 $\mathsf{Mem_v}$를 계산한다:

$$\mathsf{Mem_v} : \mathsf{AbsLoc} \mapsto \mathsf{Interval}$$

인터벌 도메인 Interval은 정수들의 집합을 최소값과 최대값의 쌍으로써 요약하며 공집합에 대응하는 요소인 $\bot$을 가진다.

Buffer Analysis 단계는 포인터 분석의 일종으로 각각의 포인터가 가리킬 수 있는 버퍼 및 접근 가능한 범위를 계산한다. 예를 들어 x가 크기 10인 배열이라면 `p = &x[1]` 이라는 할당문 이후에는 `p[-1]`부터 `p[9]`까지의 접근이 유효한데, Buffer Analysis 단계는 포인터 p가 배열 x의 둘째 항목을 가리키고 있으며 배열의 크기는 10 이라는 정보를 계산한다. Buffer Analysis 단계의 정확도는 Value Analysis 단계의 정확도에 의존한다. 예를 들어 `p = &x[i]` 라는 할당 이후에 포인터 p가 가리키는 버퍼의 경계를 정확히 알기 위해서는 변수 i의 값을 정확히 알아야 할 것이다.



[그림 2] 분석기의 구조

현재의 구현은 제어 흐름 그래프의 각 노드에 대해서, 가능한 메모리 상태를 포인터 변수를 기준으로 요약한 $\mathsf{Mem_p}$를 계산한다:

$$\begin{array}{lcl} \mathsf{Mem_p} & : & \mathsf{AbsLoc} \mapsto \mathsf{Buffers} \\ \mathsf{Buffers} & : & 2^{\mathsf{Buffer}} + \{\top\} \\ \mathsf{Buffer} & : & \mathsf{Id} \times \mathsf{Size} \times \mathsf{Offset} \\ \mathsf{Size} & = & \mathsf{Interval} \\ \mathsf{Offset} & = & \mathsf{Interval} \end{array}$$

버퍼의 크기 및 오프셋 정보의 단위는 바이트(byte)이고 포인터의 타입에 따라 접근 가능한 인덱스의 범위를 계산한다. 이것은 포인터의 타입 변환(casting)까지도 정확하고 안전하게 추적하기 위한 것이다.

마지막으로 Buffer Overrun Check 단계에서는 Buffer Analysis 결과와 Value Analysis 결과를 이용하여 버퍼에 접근하는 식들이 안전한지를 검사한다. 예를 들어 `p[i]`라는 식에 대해서, Buffer Analysis 결과에 의하면 `p`가 가리키는 버퍼가 10 바이트 크기의 1 바이트 문자 배열이고 Value Analysis 결과에 의하면 `i`가 가지는 값의 범위가 `[0,10]` 인 경우, 분석기는 버퍼 오버런이 발생할 수 있다는 경고 메시지를 출력한다. 경고 메시지는 실제 오류인 경우도 있고, 부정확한 분석으로 인한 잘못된 경고 메시지인 경우도 있다. 향후 연구에서는 Buffer Overrun Check 단계에서 오류 메시지의 원인을 자동으로 분석하고, 경우에 따라 각 단계의 정확도를 조절하여 분석을 반복하는 구조로 분석기를 설계할 계획이다.

5 분석기의 구현

연구자들은 CIL 프레임워크[8] 위에서 분석기를 구현하였다. CIL 프레임워크는 C 언어 프로그램을 파싱하여 C 언어보다 단순한 중간 언어로 변환해 줄 뿐 아니라, 구문 트리 수준에서 프로그램을 다루기 위한 다양한 함수를 제공하므로, C 프로그램을 위한 분석기를 개발하기에 매우 적합한 환경이다. 또한 CIL 프레임워크는 포인터를 한 단계까지 정확하게 분석하는 효율적인 포인터 분석[4, 5]을 기본적으로 제공한다.

현재의 구현은 Points-to Analysis 단계에서 CIL 프레임워크가 제공하는 포인터 분석을 그대로 사용하고 있는데 다음의 세 가지 면에서 잘못된 경고 메시지가 발생할 수 있다. 첫째, 프로그램의 제어 흐름을 무시하는 분석이므로 그림 2의 함수 `g`와 같은 코드의 경우 잘못된 경고 메시지를 출력한다. 둘째, 포인터 분석 시에는 서로 다른 지점에서 발생하는 함수 호출의 결과가 합쳐진다. CIL 매뉴얼에는 제공하는 포인터 분석이 서로 다른 지점에서 발생하는 함수 호출을 구분하여 분석할 수 있다고 설명하고 있지만, 실제로 수행되는 코드는 그렇지 않았다. 셋째, 배열 및 구조체 이루는 모든 메모리 영역을 하나의 요약된 메모

Filename	#Lines	Airac Time	Raccoon Time	#Airac Alarms		#Raccoon Alarms	#Real Bugs
				Buffers	Accesses		
vmax301.c	246	0.28s	0.04s	1	1	1	1
xfrm_user.c	1,201	45.07s	8.77s	2	2	1	1
usb-midi.c	2,206	91.32s	1.04s	2	10	4	4
atkbd.c	811	1.99s	0.68s	2	2	2	2
af_inet.c	1,273	1.17s	0.28s	1	1	3	1
cdc-acm.c	849	3.98s	0.73s	1	3	2	2
mptbase.c	6,158	0.79s	3.34s	1	1	1	1
aty128fb.c	2,466	0.32s	1.06s	1	1	1	1

[그림 3] 실험 결과 (Linux kernel version 2.6.4)

리 영역으로 요약하여 분석한다. 따라서 구조체가 포인터 필드를 포함하고 있다면 두 개의 필드 정보가 합쳐진 부정확한 결과를 얻게 된다.

Value Analysis 단계에서는 프로그램의 각 시점에서 (flow-sensitive) 각각의 요약된 메모리 영역이 값을 저장하는 경우 가질 수 있는 값의 범위를 계산한다. 같은 함수를 호출하는 경우라도 함수를 호출하는 시점에 따라 구분하여 (context-sensitive) 분석하는데, 함수 호출을 어느 정도 수준까지 정확하게 분석할 것인지를 지정할 수 있도록 구현되어 있다. 기본적으로는 구문적 위치가 다른 함수 호출들만을 구분하여 분석한다. Value Analysis 단계가 Points-to Analysis 단계에서 계산하는 요약된 메모리 영역을 그대로 사용하므로, 역시 구조체의 필드를 구분하여 분석하지 못한다.

언급했듯이 Buffer Analysis 단계에서는 프로그램의 각 시점에서 각각의 요약된 메모리 영역이 포인터를 저장하는 경우 가리키는 버퍼들의 정보를 계산한다. 도메인을 제외한 세부적인 부분은 Value Analysis 단계와 유사하게 구현되어 있다.

6 실험 결과

본 연구에서 개발한 분석기의 속도와 정확도를 비교하기 위해 기존의 분석기에 의해 버퍼 오버런 가능성이 검사된 Linux kernel version 2.6.4를 대상으로 실험하였고, 전체 분석 과정에서 걸린 시간과 분석기가 보고하는 경고의 개수 및 경고 내용을 비교해 보았다.

그림 3은 Linux kernel version 2.6.4에 대한 실험 결과를 보여주고 있다. #Lines은 커널 파일의 소스 코드의 길이를 나타내고, Airac Time과 Raccoon Time[3]은 각각 Airac과 연구자들이 개발한 분석기의 분석 시간을 나타낸다. Airac의 분석 시간은 3.2 GHz 인텔 펜티엄 4 프로세서와 4 GB 메모리를 갖춘 리눅스 시스템에서 실험된 결과[7]이고, Raccoon의 분석 시간은 3.0 GHz 인텔 펜티엄 4 프로세서와 2.5 GB 메모리를 갖춘 리눅스 PC를 사용하여 실험한 결과이다. #Airac Alarms는 Airac의 경고개수이다. Airac의 경우엔 사용자로 하여금 살펴볼 경고의 개수를 줄이기 위해, 버퍼 오버런이 발생할 수 있는 버퍼와 버퍼 접근의 수를 따로 보여주고 있다. #Raccoon Alarms는 연구자들이 개발한 분석기의 경고 개수이고, #Real Bugs는 실제로 버퍼 오버런이 가능한 버퍼 접근의 개수이다.

먼저 Raccoon이 보고한 경고를 살펴보면 실제로 버퍼 오버런이 가능한 경우를 빠트림없이 발견하고 있음을 알 수 있다. 분석기의 분석 시간 또한 모두 10 초 이내에 끝나며 대부분의 경우 1 초 이내에 분석 결과를 얻을 수 있었다. 분석기는 `af_inet.c`를 제외한 모든 경우에 정확하게 실제로 버퍼 오버런이 가능한 버그를 찾아내고 있다. 실험 결과를 통해 프로그램 제어 흐름을 무시하는 부정확한 포인터 분석을 사용하더라도 프로그램 제어 흐름

[3]파일 입출력에 소요되는 시간을 포함

까지 고려하는 비교적 정확한 포인터 분석을 사용하는 경우와 큰 차이가 없음을 확인할 수 있다. 분석기가 `af_inet.c`에 대해 보고한 경고 중 잘못된 경고 메시지는 사용하는 포인터 분석이 구조체의 필드를 구분하지 않기 때문에 발생한 것으로 원인이 되는 구조체의 필드를 구분하여 분석하면 발생하지 않을 것으로 예상된다. 경우에 따라 Raccoon이 Airac에 비해 정확한 결과를 얻은 것은 축지법(widening) 및 좁히기(narrowing) 연산을 비롯한 세부적인 구분에서 설계 및 구현이 조금 다를 수 있는 것에 기인한 부수적인 결과로 추측된다.

7 결론 및 향후 연구

본 논문에서는 실제적인 분석기를 제작하면서 맞닥뜨리게 되는 속도와 정확도라는 두 가지 상충하는 목표를 모두 달성하기에 적합한 분석기의 구조를 제안한다. 제안하는 분석기의 구조는 포인터와 값을 한꺼번에 분석하지 않고 별도의 단계로 구분하여 각 분석의 정확도를 다르게 적용할 수 있도록 하는 것이다.

제안하는 구조의 분석기를 시험적으로 구현하여 리눅스 커널 소스에 적용한 결과는 이러한 구조가 가능성이 있음을 보여주고 있다. 포인터를 부정확하게 분석하더라도 포인터를 값과 함께 비교적 정확하게 분석하는 기존의 분석기와 결과의 정확도 면에서 큰 차이가 없었던 반면, 분석기의 속도를 크게 향상시킬 수 있었다. 물론 그림 1에서 보였듯이 특정 프로그램의 경우에는 기존의 분석기에서는 발생하지 않는 잘못된 경고 메시지를 출력하게 될 것이다. 다만 연구자들이 실험 결과에서 다시 한 번 확인할 수 있던 것은 적어도 일부 프로그램을 분석하는 데에는 복잡한 형태의 분석이 필요하지 않으며, 따라서 모든 프로그램, 전체 코드를 일괄적으로 많은 분석 시간과 메모리를 요구하는 복잡한 분석으로 분석하는 것은 비효율적이라는 사실이다.

향후 연구 방향은 단기적으로는 현재의 구현을 안정화하여 리눅스 커널 소스 이외의 커다란 소프트웨어에 대하여 실험하는 것이고, 장기적으로는 프로그램의 각 부분별로 서로 다른 정확도의 분석을 적용할 수 있는 형태의 분석기를 설계하고 구현하는 것이다. 예를 들어 그림 1의 함수 `f`와 같은 코드를 정확하게 분석하기 위해서 저런 패턴의 반복문을 발견하면 그 반복문 부분만을 변수들의 관계를 유지하는 분석 등을 사용하여 보다 정확히 분석할 수 있도록 하려는 것이다. 또 다른 예로 현재 구조체의 필드를 구분하지 않고 분석하는데 필드를 구분하여 분석할 구조체를 지정할 수 있도록 하는 것을 들 수 있다. 이러한 유연한 구조를 기반으로 분석기가 찾아낸 버퍼 오버런 가능성 중에서 부정확한 분석으로 인한 잘못된 경고 메시지를 자동으로 걸러 낼 수 있는 분석기를 개발하려고 한다. 몇 가지 간단한 규칙 또는 일종의 휴리스틱을 통해 각각의 경고 메시지의 원인이 될 수 있는 부분을 파악하고, 그 부분의 정확도를 높여 다시 한 번 분석하는 것으로 다수의 잘못된 경고 메시지를 걸러낼 수 있을 것으로 예상하고 있다.

참고문헌

[1] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[2] Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.

[3] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Principles of Programming Languages*, pages 84–96, 1978.

[4] Manuvir Das. Unification-based Pointer Analysis with Directional Assignments. In *Programming Language Design and Implementation*, pages 35–46, 2000.

[5] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *Static Analysis Symposium*, pages 260–278, 2001.

[6] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *Programming Language Design and Implementation*, pages 155–167, 2003.

[7] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis. In *Static Analysis Symposium*, pages 203–217, 2005.

[8] Geoge C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, pages 213–228, 2002.

[9] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Principles of Programming Languages*, pages 32–41, 1996.

[10] Arnaud Venet and Guillaume P. Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *Programming Language Design and Implementation*, pages 231–242, 2004.

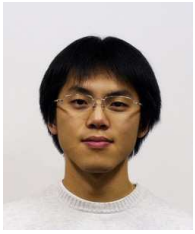
[11] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *the Network and Distributed Systems Security Symposium*, 2000.

김유일

– 2001 KAIST 전산학과 학사
– 2003 KAIST 전자전산학과 전산학전공 석사
– 2003-현재 KAIST 전자전산학과 전산학전공 박사과정

<관심분야> 프로그램 분석, 컴파일러

전진성



– 2005 KAIST 전자전산학과 전산학전공 학사
– 2005-현재 KAIST 전자전산학과 전산학전공 석사과정

<관심분야> 프로그램 분석, 컴파일러

한환수

– 1993 서울대학교 컴퓨터공학과 학사
– 1995 서울대학교 컴퓨터공학과 석사
– 2001 Ph.D., Computer Science, University of Maryland
– 2001-2003 Sr. Engineer, Intel Architecture Group, Intel
– 2003-현재 KAIST 전자전산학과 전산학전공 조교수

<관심분야> 모바일 컴퓨팅, 임베디드 시스템, 컴파일러