# 분석 문맥 조절 기법을 이용한 버퍼 오버플로우 분석의 중복 경보 제거

## (Eliminating Redundant Alarms of Buffer Overflow Analysis Using Context Refinements)

김 유 일 † 한 환 수 ††
(Youil Kim) (Hwansoo Han)

**요 약** 버퍼 오버플로우 취약점을 검출하는 정적 분석 도구에서, 분석 문맥을 조절하는 방법을 통해, 동일한 원인에 의해 발생하는 중복 경보 메시지를 제거하는 방법을 제안한다. 동일한 원인에 의해 다수의 경보 메시지가 출력되는 경우, 첫 경보 메시지만 살펴보아도 나머지 경보 메시지들에 대한 판단을 내릴 수 있으므로, 사용자에게 첫 경보 메시지만을 보여주는 것이 보다 바람직하다. 제안하는 분석 문맥 조절 기법은 기존의 정적 분석 도구에 쉽게 적용할 수 있고, 오픈 소스 소프트웨어를 사용한 실험에서 평균 23%의 경보 메시지를 제거할 수 있었다.

키워드 : 분석 문맥 조절, 중복 경보 제거, 버퍼 오버플로우, 프로그램 분석

*Abstract* In order to reduce the efforts to inspect the reported alarms from a static buffer overflow analyzer, we present an effective method to filter out redundant alarms. In the static analysis, a sequence of multiple alarms are frequently found due to the same cause in the code. In such a case, it is sufficient and reasonable for programmers to examine the first alarm instead of the entire alarms in the same sequence. Based on this observation, we devise a buffer overflow analysis that filters out redundant alarms with our context refinement technique. Our experiment with several open source programs shows that our method reduces the reported alarms by 23% on average.

Key words : Context Refinement, Redundant Alarm, Buffer Overflow, Program Analysis

## 1. 서 론

본 연구의 목표는 오픈 소스 소프트웨어를 분석하여 버퍼 오버플로우 취약점을 검출하는 실제적인 정적 분석 도구를 개발하는 것이다. 오류를 놓치지 않는 안전한 정적 분석 기술에 기반을 두고, 오픈 소스 소프트웨어의 다양성을 고려한, 완전히 자동화된 도구를 개발하려는 점에서 기존의 버퍼 오버플로우 분석 도구들[1-4]과 차별화된다.

본 논문에서 관심을 가지는 문제는 안전한 정적 분석 기법을 사용한 버퍼 오버플로우 취약점 검출이 다수의 거짓 경보(false positive)를 보고하는 점이다. 정적 분석은 모든 수행 가능한 경우의 정보를 함축하여 요약된 정보를 만들어 내는데, 이 과정에서 실제로는 수행 가능하지 않은 실행 경로의 정보가 분석 결과에 포함될 수 있다. 요약된 정보를 사용하여 수행한 분석의 결과로 실제로는 가능하지 않은 상황에 대해서 경보 메시지를 출력하는 경우가 상당 수 발생한다.

거짓 경보가 발생하는 여러 가지 원인 중에서, 본 논문에서 해결하려는 문제는 프로그램 동일한 원인에 의한 경보를 소스 코드의 여러 곳에서 반복적으로 보고하는 문제이다. 일반적으로 프로그램 오류를 프로그래머가 수정하는 방식은 처음 등장한 오류를 먼저 수정하고, 그에 의해 파급되어진 오류들은 첫 오류를 수정한 후에도

· 본 연구는 교육과학기술부 재원으로 한국연구재단(구 한국학술진흥재단)의 지원으로 수행되었음(KRF-2007-331-D00427)

† 정 회 원 : 서울대학교 컴퓨터공학과 연구원
bluewiz@gmail.com
†† 종신회원 : 성균관대학교 정보통신공학부 교수
hhan@skku.edu

논문접수 : 2010년 7월 27일
심사완료 : 2010년 9월 27일

```
/* position_set grps[256]; */
/* ngrps = [-oo, +oo] */
2078:  MALLOC(grps[ngrps].elems, position, d->nleaves);
2079:  grps[ngrps].nelem = 1;
2080:  grps[ngrps].elems[0] = pos;
```

그림 1 Grep 2.5.1에서 추출한 부분 소스 코드

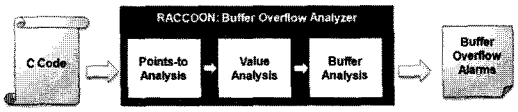


그림 2 정적 분석을 이용한 버퍼 오버플로우 검사 도구 (Raccoon)[8,9]

존재하는 경우에 수정을 시도하는 것이다. 버퍼 오버플로우 경보의 경우에도 역시 비슷한 오류 수정 과정을 수행할 것으로 예상되므로, 동일한 원인에 의해서 다수의 경보가 발생하는 경우에 최초의 경보만을 출력하여 프로그래머가 소스 코드의 오류를 검토 및 수정하는데 들이는 노력을 효과적으로 줄일 수 있을 것이다.

그림 1은 오픈 소스 소프트웨어인 Grep의 2.5.1 버전 소스 코드에서 추출한 부분 소스 코드이다. 코드는 2078줄 이후 배열 grps를 동일한 인덱스 변수 ngrps를 사용하여 세 번 액세스하는데, 정적 분석 도구가 변수 ngrps의 값이 안전한 범위에 들어가는지 정확하게 분석하지 못했다고 가정하자.

이 코드에서 연속된 세 개의 경보 메시지를 모두 출력하는 것은 불필요한 일이다. 인덱스 변수 ngrps의 값이 바뀌지 않으므로, 2078줄에 대한 경보 메시지가 거짓 경보라면 다음 두 줄의 경보 메시지도 역시 거짓 경보이다. 반면 2078줄이 실제 버퍼 오버플로우를 발생시키는 경우에는 그 원인을 수정하면 다음 두 줄의 오류도 함께 수정될 것으로 기대할 수 있다.

이처럼 동일한 원인에 의해 다수의 경보 메시지가 출력되는 경우, 첫 경보 메시지만을 출력하도록 분석 문맥을 조절하면서 분석을 수행하는 방법을 설명한다. 제안하는 분석 문맥 조절 기법은 기존 정적 분석 도구에 쉽게 적용할 수 있다. 우리는 오픈 소스 소프트웨어를 사용한 실험에서 평균 23%의 경보 메시지를 제거할 수 있었는데, 이는 경보 메시지를 살펴보는 노력도 비슷한 정도로 줄어든다는 것을 의미한다.

본 논문에서 제안하는 기법은 통계적인 방법을 이용해 중요한 경보 메시지를 먼저 보여주려는 연구[5-7]들과 차별화된다. 그러므로 통계적인 방법과 분석 문맥 조절 기법을 융합하여 보다 나은 경보 메시지 출력 방법을 설계하는 것이 가능하다. 예를 들어, 문맥 조절을 적용하여 경보 메시지의 수를 줄이고, 남은 경보 메시지들에 통계적인 방법을 적용하여 중요한 경보 메시지를 먼저 보여주는 방법을 생각할 수 있다.

이후 논문의 구성은 다음과 같다. 2장에서는 우리의 버퍼 오버플로우 정적 분석 도구를 간략히 소개하고, 3장에서는 중복된 경보 메시지를 출력하지 않도록 기존의 정적 분석을 수정하는 방법을 소개한다. 4장에서는 오픈 소스 소프트웨어를 사용하여 제안하는 방법의 효과를 실험했고, 5장에서 결론을 맺는다.

## 2. 버퍼 오버플로우 정적 분석

정적 분석 도구 Raccoon[8,9]은 버퍼 오버플로우 분석에 필요한 분석을 그림 2의 구조도와 같이 단계적으로 수행한다. 각 단계에서 분석을 통해 얻어진 정보는 다음 단계로 전달되고, 최종적으로 마지막 단계에서 버퍼 오버플로우 가능성이 있는 프로그램 문장에 대하여 경보를 출력하게 된다. 첫 단계인 포인터 분석 (points-to analysis)에서는 대상 프로그램을 구문 분석하여 제어 흐름 그래프 형태로 변형하고, Steensgaard 스타일의 포인터 분석[10]을 통해 프로그램의 좌변값(L-value)들에 요약된 메모리 주소(AbsLoc)를 할당한다. 이 과정에서 같은 메모리 주소를 가리킬 수 있는 좌변값들에는 동일한 요약된 메모리 주소가 할당된다. 두 번째 단계인 값 분석(value analysis)에서는 요약 해석[11,12]에 기반을 둔 정수 구간 분석과 버퍼 접근에 사용되는 정보 분석을 수행한다. 마지막 단계인 버퍼 분석(buffer analysis)에서는 전 단계에서 얻은 분석 결과를 참조하여 버퍼를 접근하는 프로그램 구문에 대하여 버퍼의 타입 정보에 비추어 정해진 범위 이외의 메모리 주소를 접근할 가능성이 있는 경우 버퍼 오버플로우 경보 메시지를 출력한다.

요약 해석 단계에서 사용하는 분석 도메인은 다음과 같다. 정수 구간 분석은 제어 흐름 그래프의 각 노드에 대해 다음의 요약된 메모리 $\mathsf{Mem}_\mathsf{v}$를 계산하고, 포인터 정보 분석은 제어 흐름 그래프의 각 노드에 대해 다음의 요약된 메모리 $\mathsf{Mem}_\mathsf{p}$를 계산한다. 정수 구간 분석은 요약된 메모리 주소가 정수 변수에 대응하는 경우, 가질 수 있는 정수 값의 범위(Interval)를 최대값과 최소값의 순서쌍으로 나타낸다. 포인터 정보 분석은 요약된 메모리 주소가 포인터 변수에 대응하는 경우, 포인터가 가리키는 대상 버퍼들의 정보(Pointer)를 버퍼의 크기(Size)와 오프셋(Offset)의 순서쌍으로 요약한다.

$$\begin{aligned} \mathsf{Mem}_\mathsf{v} &: \mathsf{AbsLoc} \mapsto \mathsf{Interval} \\ \mathsf{Interval} &= \mathsf{Integer} \times \mathsf{Integer} \\ \mathsf{Mem}_\mathsf{p} &: \mathsf{AbsLoc} \mapsto \mathsf{Pointer} \end{aligned}$$

Pointer : Size × Offset
Size = Interval
Offset = Interval

예를 들어, 프로그램 수행 중에 어떤 포인터가 2 바이트 버퍼의 첫 바이트 혹은 3 바이트 버퍼의 둘째 바이트를 가리킨다면 이 정보를 크기 [2, 3], 오프셋 [0, 1]로 요약하고, 이 포인터를 통해 한 바이트를 읽고 쓰는 것만 안전하다고 판단한다.

Raccoon은 CIL 컴파일러[13]를 기반으로 하여 개발되었다. ANSI C 문법과 GNU C 컴파일러 문법을 지원하여 대부분의 오픈 소스 소프트웨어를 그대로 분석할 수 있다.

## 3. 분석 문맥 조절을 이용한 중복 경보 제거

본 논문에서 분석 문맥이란 프로그램의 실제 수행 중에 나타날 수 있는 모든 메모리 상태를 요약한 정보인 $Mem_v$와 $Mem_p$를 말한다. 이 장에서는 버퍼 오버플로우 경보가 발생하는 상황에서, 버퍼 오버플로우가 발생하지 않는 분석 문맥만을 가정하면서 이후의 분석을 진행하는 것의 의미를 설명하고, 2장에서 정의한 분석 도메인 위에서 그러한 방식으로 분석을 수행하는 방법을 설명한다.

그림 1의 2078 줄의 경보 메시지가 거짓 경보라고 가정하자. 다시 말해, 우리의 분석 도구는 변수 ngrps의 최소값이 $-\infty$, 최대값이 $+\infty$라는 분석 결과를 얻었는데, 이것이 과도한 요약에 의한 부정확한 범위이고, 실제 수행에서는 변수 ngrps가 버퍼 오버플로우가 발생하지 않는 범위의 값만 가진다는 의미이다. 이 경우 변수 ngrps의 값을 버퍼 오버플로우가 발생하지 않는 범위 [0, 255]로 가정하는 것은 실제 수행에서 나타나는 값의 범위를 모두 포함하면서도 보다 정확한 분석 정보가 된다.

이번에는 그림 1의 2078 줄의 경보 메시지가 거짓 경보가 아니고, 실제 오버플로우가 발생한다고 가정하자. C 언어에서 버퍼 오버플로우가 발생하는 경우의 동작은 명확하게 정의되어 있지 않으므로, 수행 환경에 따라서는 다음 두 줄에서 연속적으로 버퍼의 경계를 넘는 상황이 발생할 수도 있다. 이 경우에 변수 ngrps의 값을 버퍼 오버플로우가 발생하지 않는 범위 [0, 255]로 가정하고 분석을 진행하는 것은 버퍼 오버플로우가 발생하지 않는 수행 경로들만을 따라 또 다른 버퍼 오버플로우 가능성을 검출하려고 시도하는 효과가 있다.

그림 1과 같은 코드에서 2079 줄과 2080 줄의 중복된 경보 메시지를 제거하기 위해 분석 문맥 조절을 사용하는 방법은 다음과 같다. 우리의 분석 도구는 구간 분석의 결과로 분석되는 변수 ngrps의 범위를 가지고 배열 접근 식 grps[ngrps]의 버퍼 오버플로우 여부를 판단한다. 구간 분석 과정에서 grps[ngrps]라는 배열 접근 식을 지날 때에 분석 문맥에 저장된 ngrps의 값을 안전한 구간 [0, 255]와의 최대하계(Greatest Lower Bound)로 수정한다. 여기서 안전 구간의 상한인 255는 대상 배열의 선언된 크기에 따라 규정되는 숫자이다. 요약하면, 2078 줄의 배열 접근식을 지나면서, 변수 ngrps의 분석 결과가 [0, 255]로 조정된다.

Raccoon에 이와 같은 아이디어를 적용하기 위해 배열 접근식을 분석하는 함수를 확장하여, 배열 접근식을 지날 때에 배열 인덱스 식에 사용된 변수들의 값이 조절된 새로운 요약 메모리를 계산하도록 했다. 일반적으로는 역방향 분석[13]을 적용하여 배열 인덱스 식이 안전한 범위(0 이상 배열 크기 미만)로 계산되기 위한 초기 상태를 계산할 수 있다. 역방향 분석은 요약 해석 기반 프로그램 분석기에서 조건문의 정확한 분석을 위해 많이 사용되는 기법이다.

본 논문에서는 분석 문맥 조절의 아이디어를 배열 접근 식에서 발생하는 경보 메시지에 적용해 보았다. 포인터를 통해 버퍼에 접근할 때 발생하는 경보 메시지와 문자열 함수에서 발생하는 경보 메시지에도 분석 문맥 조절 방법을 적용하는 것은 향후 연구 과제이다.

## 4. 실험 결과

논문에서 제안하는 아이디어의 효과를 평가하기 위해 Bugbench[14] 벤치마크 중 버퍼 오버플로우를 포함하는 프로그램들에 대해 분석을 수행했다. 실험은 두 개의 3.2 GHz XEON 프로세서와 4 GB의 메모리를 가진 리눅스 PC를 사용하여 수행했다.

표 1은 기존 Raccoon의 분석 결과와 분석 문맥 조절 기법을 적용한 후의 분석 결과를 함께 요약한 것이다. 첫째 열과 둘째 열은 벤치마크 프로그램의 이름과 크기를 나타낸다. 프로그램의 크기는 SLOCCount 도구[15]를 사용하여 주석과 빈 줄을 제외하고 측정했다. 셋째 열은 총 배열 접근식의 개수를 측정한 것이다. 넷째 열과 다섯째 열은 3장에서 제안하는 분석 문맥 조절 기법을 적용하기 전과 적용한 후에 경보 메시지 개수를 측정한 것이고, 여섯째 열은 경보 메시지 개수의 변화를 백분율로 계산한 것이다. 평균 23%의 경보 메시지가 감소했음을 알 수 있다. 일곱째 열과 여덟째 열은 제안하는 방법을 적용하기 전과 적용한 후의 분석 시간을 측정한 것이고, 마지막 열은 분석 시간의 변화를 백분율로 계산한 것이다. 추가적인 분석 시간이 최대 6.7% 이내임을 알 수 있다. 분석 문맥 조절이 배열 접근 식에서 이루어지므로, 배열 접근 식이 비교적 많은 gzip-1.2.4의 분석 시간이 크게 증가하는 것은 예상할 수 있는 결과이다. 다만, 배열 접근 식이 가

표 1 실험 결과

| 프로그램 | SLOC | 배열 접근식 | 적용전 경보 | 적용후 경보 | 감소 비율 | 적용전 시간 | 적용후 시간 | 증가 비율 |
|---|---|---|---|---|---|---|---|---|
| polymorph-0.4.0 | 1,357 | 3 | 3 | 3 | 0% | 0.1s | 0.1s | 0.0% |
| ncompress-4.2.4 | 2,195 | 43 | 21 | 18 | 14% | 1.1s | 1.1s | 0.0% |
| man-1.5h1 | 7,232 | 27 | 3 | 3 | 0% | 8.7s | 8.7s | 0.0% |
| gzip-1.2.4 | 11,213 | 394 | 241 | 213 | 12% | 35.2s | 37.6s | 6.7% |
| bc-1.0.6 | 12,830 | 50 | 41 | 40 | 2% | 138.4s | 140.3s | 1.4% |
| tar-1.13 | 21,891 | 108 | 32 | 29 | 9% | 775.0s | 776.2s | 0.1% |
| 099.go | 47,337 | 9,744 | 8,925 | 6,866 | 23% | 966.6s | 872.1s | -9.8% |
| 120.compress | 5,585 | 35 | 16 | 15 | 6% | 1.0s | 1.0s | 0.0% |
| 합계 | 109,640 | 10,404 | 9,282 | 7,187 | 23% | 1,926.1s | 1,837.1s | -4.6% |

장 많은 099.go는 분석 시간이 오히려 줄어들었는데, 정확한 분석 결과로 인해 분석 방정식의 고정점에 도달하는 시간이 단축되는 정도가 분석 문맥 조절의 오버헤드보다 크기 때문인 것으로 판단된다.

## 5. 결 론

본 논문에서는 버퍼 오버플로우 정적 분석 도구에서 동일한 원인에 의해 연속적으로 경보 메시지들이 발생하는 경우를 보이고, 이를 효과적으로 해결하는 분석 문맥 조절 기법을 소개했다. 제안하는 방법을 배열 접근식에 적용하여 실험한 결과 평균 23%의 경보 메시지를 줄일 수 있었다. 본 논문의 핵심 아이디어는 특정 정적 분석 도구의 구조와 밀접하게 연관되어 있지 않으므로, 다른 방법으로 설계한 정적 분석 도구에 적용할 수 있을 것이다.

## 참 고 문 헌


[1] Y. Xie, A. Chou, D. R. Engler, Archer: using symbolic, path-sensitive analysis to detect memory access errors, ESEC/FSE, 2003.

[2] N. Dor, M. Rodeh, S. Sagiv, CSSV: towards a realistic tool for statically detecting all buffer overflows in C, PLDI, 2003.

[3] A. Venet, G. P. Brat, Precise and efficient static array bound checking for large embedded C programs, PLDI, 2004.

[4] W. Le, M. L. Soffa, Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector, FSE, 2008.

[5] T. Kremenek, D. R. Engler, Z-ranking: Using statistical analysis to counter the impact of static analysis approximations, SAS, 2003.

[6] Y. Jung, J. Kim, J. Shin, K. Yi, Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis, SAS, 2005.

[7] S. Kim, M. D. Ernst, Which Warnings Should I Fix First?, FSE, 2007.

[8] Y. Kim, J. Jeon, H. Han, Development of cost-effective buffer overrun analyzer, KIISE SIGPL Transactions on Programming Languages, vol.19, no.2, pp.1-9, 2005.

[9] Y. Kim, J. Lee, H. Han, K.-M. Choe, Filtering false alarms of buffer overflow analysis using SMT solvers, Information and Software Technology, vol.52, no.2, pp.210-219, 2010.

[10] B. Steensgaard, Points-to analysis in almost linear time, POPL, 1996.

[11] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, POPL, 1977.

[12] P. Cousot, R. Cousot. Systematic Design of Program Analysis Frameworks, POPL, 1979.

[13] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer, CIL: Intermediate language and tools for analysis and transformation of C programs, CC, 2002.

[14] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, Y. Zhou, Bugbench: Benchmarks for evaluating bug detection tools, Workshop on the Evaluation of Software Defect Detection Tools, 2005.

[15] SLOCCount, http://www.dwheeler.com/sloccount/.



김 유 일

2001년 KAIST 전자전산학과 전산학전공 학사. 2003년 KAIST 전자전산학과 전산학전공 석사. 2010년 KAIST 전자전산학과 전산학전공 박사. 2010년~현재 서울대학교 소프트웨어 무결점 연구센터 연구원. 관심분야는 프로그램 정적 분석 등

한 환 수

정보과학회논문지 : 소프트웨어 및 응용
제 37 권 제 10 호 참조