Java 스택 인스펙션을 위한 권한검사 분석 시스템*

김윤경[0] 창병모
숙명여자대학교
{ykkim79[0], chang}@sookmyung.ac.kr

Static Permission Check Analysis System for Java Stack Inspection

Yunkyung Kim[0], Byeong-Mo Chang,
Department of Computer Science, Sookmyung Women's University

요 약

Java 2에서는 자원의 접근관리를 위하여 정책파일에 근거한 스택 인스펙션(stack inspection)기법을 제공하고 있다. 본 논문에서는 스택 인스펙션에 자연스럽게 접근하여 불필요한 권한검사 집합을 구하기 위해 역방향 흐름분석(backward flow analysis)기법을 사용한 권한검사 시스템을 구현하였다. 이를 통해 정책파일의 내용에 근거하여 각 메소드에서 항상 성공하거나 실패하는 권한검사를 결정하여 보여준다. 또한 권한검사에 대해서 스택 인스펙션하는 과정을 추적해볼 수 있다. 본 시스템을 이용하는 사용자는 불필요한 권한검사를 제거하여 스택 인스펙션을 최적화하거나, 자신의 프로그램에 적절한 보안정책을 세우기 위해 정책파일을 수정하는데 이 분석결과를 활용할 수 있다.

키워드 : Java 2, 스택 인스펙션, 접근권한 관리, 보안, 정적분석, 정책 파일

1. 서 론

악의적인 프로그램의 실행으로 발생할 수 있는 문제점으로부터 시스템을 보호하기 위한 적절한 접근권한 관리는 중요하다. 프로그래머는 자신이 원하는 접근권한 관리를 하기 위해서 정책파일과 프로그램을 수동적으로 확인하고 시험해 보아야 한다.

본 논문에서는 프로그램에 적절한 보안정책을 세우는데 편의를 제공하고자 권한검사(permission check) 관련 정보를 정적으로 분석하여 시각적으로 알려주는 시스템을 설계하고 구현하였다.

이 권한검사 분석 시스템에서는 스택 인스펙션 과정을 자연스럽게 모델링하고 [2]에서 제안된 역방향 흐름 분석(backward flow analysis)을 사용하여 불필요한 권한검사 집합을 계산한다. 본 시스템은 정책파일과 프로그램을 분석하여 각 메소드 별로 도달 가능한 권한검사 집합과 항상 성공 또는 실패하는 권한검사 집합을 계산한다. 또한 권한검사가 수행될 때 실행시간 호출스택을 검사하는 가능한 스택 인스펙션 과정을 시각적으로 추적해 볼 수 있다.

본 시스템을 이용하는 사용자는 항상 성공 또는 실패하는 권한검사를 제거하여 스택 인스펙션에 대한 오버헤드를 감소시키거나 프로그램의 각 지점에서 자신이 의도한 대로 권한검사가 이루어지고 있는지 확인할 수 있고, 분석에 적용한 정책파일을 수정, 보완한 후 반복적으로 분석결과를 확인할 수 있다.

이 논문은 관련연구, 정적분석, 시스템의 구현 및 실험결과, 결론 및 향후 연구로 구성된다.

2. 관련연구

2.1 스택 인스펙션(stack inspection)

Java 2에서는 정책파일에 근거하여 접근권한 관리를 한다. 정책파일에 따라 보안관리자(Security Manager)의 checkPermission(p) 함수가 호출될 때, 현재 호출스택(call stack)의 모든 프레임(frame)을 역으로 추적(backward traverse)하여 해당동작에 대한 권한이 있는지 검사한다[1]. 검사하고 있는 퍼미션에 대한 권한이 없는 프레임을 만나면 Security Exception을 발생시키며 프로그램은 종료하게 된다.

Java에서는 AccessController.doPrivileged(A) 메소드를 제공하고 있는데, 스택 인스펙션 과정에서 doPrivileged() 메소드를 호출한 프레임을 만나면 검사를 중지한다. 이 프레임이 권한을 가지고 있으면 스택 인스펙션은 더 이상 진행되지 않는다. 새로운 쓰레드 T가 생성되었을 때 T는 현재 호출 스택의 복사 본을 가지게 된다.

2.2 Barat

Barat은 Java 프로그램의 정적 분석을 지원하는 컴파일러의 전단부(front end)이다[3]. Barat은 Java 소스 코드 파일이나 클래스 파일들을 분석하여 이름 분석 정보(name analysis information)와 타입분석 정보(type analysis information)를 포함하는 추상구문트리(abstract syntax tree, AST)를 구성한다. Barat은 프로그래머가 쉽게 AST의 각 노드를 순회할 수 있는 Descending Visitor, Output Visitor 등의 비지터 패턴(visitor pattern)을 제공하고 있다.

3. 정적분석

본 논문에서 구현한 권한검사 분석 시스템의 기반이 되는 정적분석 기법을 소개한다. 본 논문에서 checkPermission(p) 함수는 *check(p)*로 간략 표현하고, *check(p)*를 메소드 m에서 호출했을 때 *check(p)* $\in$ *m*으로 표현할 것이다. 정책파일에 따라, 각 메소드 *m*에 주어진 퍼미션들의 집합은 *Permi ion(m)*으로 나타낸다.

본 논문에서는 다음과 같은 호출 그래프(call graph)를 기반으로 정적분석(static check analysis)을 정의한다[2].

정의 1. 호출 그래프 $CG=(N,E)$는 방향그래프(directed graph)로 여기서 N은 메소드를 나타내는 노드들의 집합이고, $E = N \times N$은 메소드 호출을 나타내는 에지들의 집합이다.

정의 2. 호출 그래프 상에서 메소드 n에서부터 *check(p)*를 포함하고 있는 메소드 m까지의 경로가 존재하고, 이 경로 상에 있는 모든 메소드가 퍼미션 p에 대한 권한을 가지고 있을 때 메소드 n에서 이 *check(p)*은 may-succeed(성공가능)하다.

다음은 각 메소드에서의 may-succeed check(성공가능 검사)을 결정하는 **May-Succeed Check(May-SC) Analysis**를 정의하는 식(flow equation)이다.

* 본 연구는 한국과학재단 특정기초연구(R01-2006-000-10926-0) 지원으로 수행되었음.

$$May-SC_{exit}(n)=\begin{cases}\varnothing & \text{if } n \text{ is final}\\ \cup\ \{May-SC_{entry}(m)\mid\ n\rightarrow m\in E\ \} & otherwise\end{cases}$$

$$May-SC_{entry}(n)=\{check(p)\mid check(p)\in May-SC_{exit}(n),\ p\in Permission(n)\}\cup gen_{May-SC}(n)$$

$$where,\ gen_{May-SC}(n)=\{\ check(p)\mid\ check(p)\in n, p\in Permission(n)\}$$

위 식은 $F_{May-SC}: L\rightarrow L$을 정의하고, L은 $L_{entry}\times L_{exit}$이다. 여기서 L_{entry}와 L_{exit}은 노드 N에서 2^{check}으로의 함수집합이다. 프로그램 내에 있는 권한검사들의 집합이 유한하므로 L이 유한하고, F_{May-SC}은 단조 증가하기 때문에 함수의 $lfp(F_{May-SC})$는 유한 시간 내에 계산할 수 있다. 따라서 Tarski 정리에 의해 위 식의 해인 (may-sc_{entry}, may-sc_{exit})$\in L$은 유한한 n에 대해서 $lfp(F_{May-SC}) = F^{n}_{May-SC}(\perp)$에 의해 계산된다.

메소드 n에서 must-fail check(항상실패 검사)인 must-fc(n)는 다음과 같이 구할 수 있다.

$$\textbf{must-fc(n) = rc(n) - may-sc}_{entry}\textbf{(n)}$$

rc(n)은 메소드 n에 도달 할 수 있는 reachable check(도달가능 검사)이며, 이를 위한 식(flow equation)은 다음과 같다.

$$RC(n)=\begin{cases}\{\ check(p)\mid check(p)\in n\} & \text{if } n \text{ is final}\\ \{\ RC(m)\mid\ n\rightarrow m\in E\}\cup\{\ check(p)\mid\ check(p)\in n\} & otherwise\end{cases}$$

정의 3. 호출 그래프 상에서 메소드 n에서부터 $check(p)$를 포함하고 있는 메소드 m까지의 모든 경로에 대해서, 이 경로 상에 있는 모든 메소드가 퍼미션 p에 대한 권한을 가지고 있을 때 메소드 n에서 이 $check(p)$은 must-succeed(항상성공)한다.

메소드 n에서 must-succeed check(항상성공 검사)인 must-sc(n)는 다음과 같이 구할 수 있다.

$$\textbf{must-sc(n) = rc(n) - may-fc}_{entry}\textbf{(n)}$$

may-fc_{entry}(n)이란 메소드 n에서부터 check(p)을 포함하고 있는 메소드 m까지 이 퍼미션을 만족하지 않는 경로가 존재하는 것이다.

다음은 각 메소드에서의 may-fail check(실패가능 검사)을 결정하는 **May-Fail Check(May-FC) Analysis**를 정의하는 식(flow equation)이다.

$$May-FC_{exit}(n)=\begin{cases}\varnothing & \text{if } n \text{ is final}\\ \cup\{May-FC_{entry}(m)\mid\ n\rightarrow m\in E\} & otherwise\end{cases}$$

$$May-FC_{entry}(n)=May-FC_{exit}(n)\cup gen_{May-FC(n)}$$

$$where\ gen_{May-FC(n)}=\{\ check(p)\in rc(n)\mid\ p\notin Permission(n)\}$$

위 식은 $F_{May-FC}: L\times L$을 정의하고, 이 식의 해인 (may-fc_{entry}, may-fc_{exit})$\in L$은 유한시간 내에 $lfp(F_{May-FC})$에 의해 계산된다. 프로그램 내에 있는 권한검사의 수와 L이 유한하고, F_{May-FC}은 단조 증가한다. 따라서 유한한 n에 대해서 $lfp(F_{May-FC})$은 $F^{n}_{May-FC}(\perp)$에 의해 계산된다.

이러한 정적분석을 그림1과 같은 어플리케이션에 적용하여 보겠다. 표1은 정책파일에 따라 각 프로텍션 도메인에 주어진 권한집합이다. 위의 예에서 $check(p_{read})$는 CountFile1, PrivExcAction, CountFile2에서는 must-succeed check(항상성공 검사)이고, CountFileCaller2에서는 must-fail check(항상실패 검사)이다.

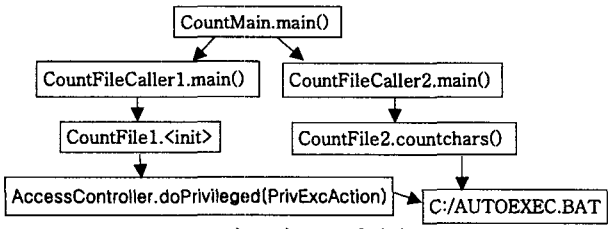

그림 1. 메소드 호출관계

Protection Domain	Granted Permission
CountMain	RuntimePermission "createSecurityManager" RuntimePermission"setSecurityManager"
CountFileCaller1 CountFileCaller2	∅
CountFile1 PrivExcAction CountFile2	FilePermission "C:/AUTOEXEC.BAT","read" =>check(p_{read})

표 1. 정책파일

4. 시스템 구현 및 실험결과

권한검사 분석 시스템은 다음과 같이 세 단계로 구성된다.

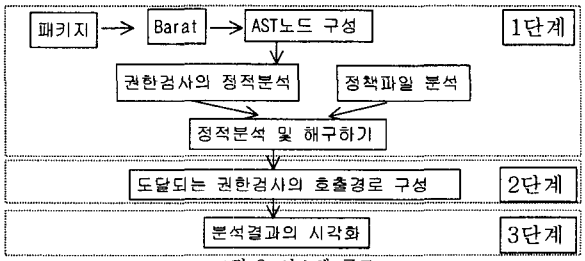

그림 2. 시스템 구조

1단계 : 정적분석 및 해 구하기

입력받은 정책파일(policy file)로부터 각 메소드가 속한 프로텍션 도메인(protection domain)별로 허용된 권한을 permMap에 저장한다.

호출 그래프를 구성하기 위해서 패키지 내의 각 메소드에서 호출하고 있는 메소드 집합을 그림3의 consMap에 저장한다.
consMap은 패키지 내에 있는 메소드를 키 값으로, 각 메소드에서 호출하고 있는 메소드 집합을 값(Value)으로 갖는다. consMap에서 권한검사 집합을 구해 그림 3의 오른쪽과 같은 형태의 맵에 저장한다.

< 권한검사 집합 구하기>

Map은 consMap과 같은 키 집합을 갖도록 초기화한다. Map의 ExitVal과 EntryVal는 각각 메쏘드의 entry와 exit 지점의 권한검사 집합을 저장하며 처음에는 비어있다. consMap과 Map은 키와 값을 쌍으로 갖는 엔트리들의 집합이다.

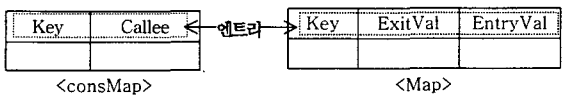

그림 3.consMap and Map

다음은 consMap에서 **may-succeed check(성공가능 검사)을** 구하는 알고리즘이다. Map.ExitVal(k)은 Map에서 k를 key로 갖는 엔트리의 ExitVal 값을 의미한다. *Permi ion(n)*은 메소드 n에 주어진

권한집합이다.

```
initflag=0;
while(initflag==0 || Map의 모든 엔트리에 대해서 ExitVal ≠ EntryVal){
  Map의 모든 엔트리에 대해서 ExitVal을 EntryVal에 복사;
  initflag=1;
  for(consMap의 각 엔트리 e에 대해서){
    k=e의 Key;
    for(Callee안의 각 메소드 m에 대해서){
①     if(m==일반메소드&& m≠doPrivileged()){
          Map.EntryVal(m)중에서 k에서 성공하는
          권한검사를 Map.ExitVal(k)에 추가; }
②     else if(m ==checkPermission(p) &&  p ∈ Permission(n)){
                Map.ExitVal(k)에 m추가; }
    }
  }
}
//doPrivileged()함수 호출을 위해 다시 한번 계산
for(consMap의 각 엔트리 e에 대해서){
  k=e의 Key;
  for(Callee안의 각 메소드 m에 대해서){
③       if(m=doPrivileged()){
                Map.EntryVal(m)중에서 k에서 성공하는
                권한검사를 Map.ExitVal(k)에 추가; }
    }
  Map.ExitVal(k)를 Map.EntryVal(k)에 복사;
}
```

위 알고리즘은 Map의 ExitVal과 EntryVal이 같아질 때까지 반복한다. consMap의 모든 엔트리에 대해 ①과 ②에서 각 메소드에서의 may-succeed check(성공가능 검사)를 구한다. ①에서 k가 가리키는 메소드에서 일반 메소드 m을 호출했을 때(k→m), m의 EntryVal을 k의 ExitVal에 추가한다. 이것은 may-sc_{entry}(m)을 may-sc_{exit}(k)에 추가하는 과정이다. 이때 m이 doPrivileged()함수이면 이 과정을 생략하고 ③에서 계산한다. 그렇지 않으면 k을 호출한 메소드 n의 ExitVal을 계산할 때 doPrivileged()함수 내에서의 권한검사가 추가되기 때문이다.

알고리즘의 ②에서 if문의 두 번째 조건을 $p \not\in Permi\ ion(n)$으로 바꾸어 **may-fail check**(실패가능 검사)를 구한다. 마지막에 각 엔트리에 대해 reachable check(도달가능 검사)중 실패하는 권한검사를 EntryVal에 추가한다. 각 메소드의 reachable check(도달가능 검사)에서 may-succeed check(성공가능 검사)와 may-fail check(실패가능 검사)를 제외하여 각각 must-fc(n)과 must-sc(n)를 구한다.

2단계 : 도달되는 권한 검사의 호출경로 구성

권한검사의 호출경로를 구성하기 위해 각 메소드에 도달 가능한 권한검사를 클래스 checkp 타입의 객체로 생성하여 저장한다. 클래스 checkp는 checkPermission(p)함수의 호출경로를 저장하기 위한 변수, traceInfo[0]와 traceInfo[1]을 가지고 있다. 메소드 n에서 m을 호출했을 때(n→m) 스택 인스펙션이 m에서 n으로 진행됨을 나타내기 위해 checkp객체의 traceInfo[0]=m을, traceInfo[1]=n을 저장한다. 우리는 각 메소드에 도달하는 권한검사에 대해서 traceInfo[0]과 traceInfo[1]을 따라가면 checkPermission(p)함수가 호출된 경로를 따라 스택 인스펙션 하는 과정을 추적할 수 있다.

3단계 정적분석 결과의 시각화

다음은 예로 들었던 CountMain 프로그램을 분석한 결과를 시각적으로 보여주는 시스템의 실행화면으로 각각에 대한 설명은 다음과 같다.

① 선택한 패키지내의 클래스와 메소드의 리스트가 보여진다.

② 선택한 정책파일의 내용이 보인다. 사용자는 필요에 따라 정책파일을 수정한 후, 반복적으로 분석할 수 있다.

③ 메소드 별 정보와 권한검사별 정보를 선택할 수 있는 옵션선택창이 있다.

④ 도달 가능한 권한검사 집합, 항상 성공 또는 실패하는 권한검사 집합에 대한 분석결과가 보인다.

⑤ 선택한 권한검사에 의한 스택 인스펙션 과정을 시각적으로 추적해 볼 수 있다.

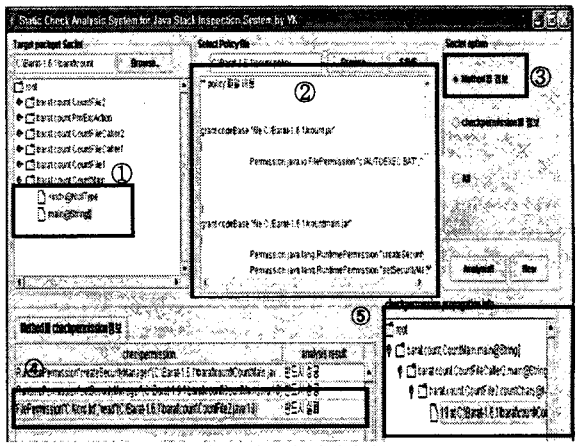

그림 4. 시스템 실행화면

표 2는 이러한 정적분석을 다섯 개의 패키지에 적용한 결과로 정적분석 결과는 각 패키지의 main() 함수에서의 결과이며, 이 결과는 정책파일에 따라 달라질 수 있다.

프로그램 이름	전체	must-fail check 수	must-succeed check 수	reachable check 수
CountMain	5	1	2	3
BankSystem	6	0	0	4
StringSearch	10	5	5	10
getProps	9	2	0	7
Server-Client	3	1	0	1

표 2. 실험 결과

위 실험결과를 통해 프로그래머는 보안 예외가 발생하지 않도록 정책파일을 수정하거나, 스택 인스펙션을 최적화하기 위해서 불필요한 권한검사를 프로그램에서 제거할 수 있다.

5. 결론 및 향후 연구

본 시스템은 권한검사와 정책파일에 대한 전반적인 정적분석 결과를 시각화하여 보여줌으로써 자원에 대한 적절한 접근권한 관리를 하는데 편의성을 제공한다. 향후에는 프로그래머의 선택에 따라 불필요한 권한검사를 자동적으로 제거하거나, 적절한 소스코드로 대체할 수 있도록 시스템을 보완하는 연구가 진행될 것이다.

[참고문헌]

1. M. Bartoletti, P. Degano, G. L. Ferrari. Stack inspection and secure program transformations, International Journal of Information Security Volume 2 , Issue 3 , August 2004

2. Byeong-Mo Chang, Static Check Analysis for Java Stack Inspection, ACM SIGPLAN Notices, To appear.

3. Boris BokoWski, André Spiegel. Barat-A Front-End for Java. Technical Report B-98-99. December 1998.