정적분석을 이용한 자바 언어의 권한검사 시각화 시스템

김 윤 경[†] · 창 병 모[††]

요 약

Java 2에서 자원에 대한 접근 권한관리를 위하여 프로그래머는 프로그램 실행에 필요한 권한들을 정책파일(policy file)에 기술하고, 보안관리자(Security Manager)를 설치하여 프로그램을 수행시킨다. 프로그램 실행 중에 자원에 대한 접근을 시도할 때마다, 보안관리자는 스택 인스펙션을 통해 접근 권한이 있는지 검사한다. 본 논문에서는 프로그래머가 안전한 보안정책을 세우는데 편의를 제공하고자 권한검사 관련 정보를 정적으로 분석하여 시각적으로 보여주는 시스템을 구현하였다. 권한검사 시각화 시스템은 정적 권한검사 분석에 기반하여, 각 메소드 별로 항상 성공하거나 항상 실패하는 권한검사를 계산한다. 프로그래머는 프로그램을 실행시켜보기 전에 권한검사와 스택 인스펙션이 어떻게 수행되는지 시각적으로 확인해 볼 수 있다. 이러한 정보를 토대로 안전한 보안정책을 세우기 위해 프로그램이나 정책파일을 수정할 수 있다.

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

A Visualization System for Permission Check in Java using Static Analysis

Yunkyung Kim[†] · Byeong-Mo Chang[††]

ABSTRACT

In Java 2, to enforce a security policy of a program, programmer writes permission sets required by the code at the policy file, sets Security Manager on system and executes the program. Then Security Manager checks by stack inspection whether an access request to resource should be granted or denied whenever code tries to access critical resource. In this paper, we develop a visualization tool which helps programmers enforce security policy effectively into programs. This system is based on the static permission check analysis which analyzes permission checks which must succeed or fail at each method. Based on this analysis information, programmer can examine visually how permission checks and their stack inspection are performed. By modifying program or policy file if necessary and examining analysis information repeatedly, programmer can enforce security policy correctly.

Key Words : Java, Stack Inspection, Security, Policy File, Static Analysis

1. 서 론

악의적인 프로그램의 실행으로 발생할 수 있는 문제점으로부터 시스템을 보호하기 위한 중요한 자원에 대한 접근 권한 관리는 매우 중요하다. 자원에 대한 접근 권한 관리를 위해서 Java 2에서는 프로그램 실행에 필요한 권한을 정책파일(policy file)에 기술하고 보안관리자를 설치하여 프로그램을 실행시킨다. 프로그램 실행 중에 자원에 대한 접근을 시도할 때마다, 보안관리자가 정책파일에 근거하여 스택 인스펙션에 의해 접근 권한이 있는지 검사한다. 프로그래머는 정책파일과 프로그램을 반복적으로 실행시켜 봄으로써 안전한 보안정책이 잘 수행되는지 확인할 수 있다.

본 논문에서는 안전한 Java 2 프로그램을 개발할 수 있도록 이러한 확인과정을 돕는 시각화 도구를 설계하고 구현하였다. 이 권한검사 시각화 시스템은 [2]에서 제안한 정적 권한검사 분석에 근거하여 권한 검사 관련 정보를 분석하여 시각적으로 보여준다. 권한 검사 시각화 시스템은 먼저 메소드 호출관계에 기반하여 각 메소드 별로 도달 가능한 권한 검사 집합과 항상 성공하거나 항상 실패하는 권한검사 집합을 계산하고, 권한검사 호출 경로를 구성한 후 정적 분석 정보를 시각적으로 보여준다.

사용자는 관심있는 메소드나 권한검사를 선택하여 분석정보를 확인할 수 있고, 권한 검사 호출 경로를 따라 권한검사에 의한 스택 인스펙션이 어떻게 수행되는지 시각적으로 추적해 볼 수 있다. 시스템에서 제공하는 정보를 토대로, 필요에 따라 정책파일과 프로그램을 수정할 수 있으며, 안전

※ 본 연구는 숙명여자대학교 2006년도 교내연구비 지원에 의해 수행되었음.
† 준 회 원 : 숙명여자대학교 컴퓨터과학과 석사
†† 정 회 원 : 숙명여자대학교 컴퓨터과학과 교수
논문접수 : 2006년 6월 9일, 심사완료 : 2006년 8월 30일

한 보안정책이 세워질 때까지 이러한 과정을 반복할 수 있다.

본 논문은 다음과 같이 구성된다. 2장에서는 Java 2의 자원 접근관리 메커니즘인 스택 인스펙션과 Java 프로그램의 정적분석을 지원하는 컴파일러의 전단부인 Barat에 대해 알아본다. 3장에서는 권한검사 시각화 시스템의 기반이 되는 정적 권한검사 분석을 소개한다. 4장에서는 권한검사 시각화 시스템의 설계 및 구현에 대해 설명하고 5장에서는 5개의 Java 패키지를 분석한 실험결과를 제시한다. 마지막으로 6장에서는 결론을 맺는다.

2. 관련 연구

이 장에서는 본 논문의 관련 연구들에 대해 소개한다. 먼저, Java 2의 자원 접근관리 메커니즘인 스택 인스펙션과 Java 프로그램의 정적분석을 지원하는 컴파일러의 전단부인 Barat을 소개한다.

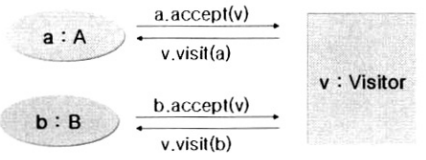
2.1 스택 인스펙션(Stack Inspection)

Java 2에서는 자원에 대한 접근권한 관리를 위해서 정책파일(policy file)에 각 프로텍션 도메인(protection domain)에 대한 권한들을 기술하고, 보안관리자(Security Manager)를 설치하여 프로그램을 수행시킨다. 프로그램 실행 중에 자원에 대한 접근을 시도할 때마다 보안관리자는 checkPermission(p) 함수를 호출하여 호출 스택에 있는 모든 프레임을 역으로 추적하여 권한이 있는지 검사한다[1, 7]. 이러한 Java 2에서의 접근권한 관리 메커니즘을 **스택 인스펙션(stack inspection)**이라 부른다. 자원에 대한 접근권한이 없는 프레임을 만나면 SecurityException을 발생시키며 프로그램은 종료하게 된다. 호출 스택에 있는 모든 프레임이 자원에 대한 접근권한을 가지고 있으면 스택 인스펙션은 종료하고, 자원에 대한 접근은 허용된다.

Java에서는 권한이 없는 클래스에 임시적으로 권한을 허가하여 특정작업을 수행할 수 있도록 하기 위해 AccessController.doPrivileged(A) 메소드를 제공하고 있다[1]. 메소드 M에서 doPrivileged(A) 메소드를 호출하면 A.run() 메소드가 호출되고 특정 동작을 수행할 수 있게 된다. 이것은 메소드 M의 메소드 프레임에 privileged라고 표시하는 것과 같다. 스택 인스펙션은 doPrivileged() 메소드를 호출한 privileged 프레임을 만나면 검사를 중지하며[4], privileged 프레임이 권한을 가지고 있으면 자원에 대한 접근은 허용된다.

2.2 Barat

Barat은 자바 프로그램의 정적 분석을 지원하는 컴파일러의 전단부(front end)이다[3]. Barat은 자바 소스 코드 파일이나 클래스 파일들을 분석하여 추상구문 트리(AST)를 구성하며, 프로그래머가 쉽게 AST의 각 노드를 순회할 수 있도록 Descending Visitor, Default Visitor, Output Visitor 등의 Visitor 디자인 패턴(design pattern)을 제공하고 있다.



(그림 1) Visitor 디자인 패턴

Visitor 디자인 패턴의 예는 (그림1)과 같고, 방문되어진 각각의 객체들은 void accept(Visitor v)에 대한 구현을 포함하고 있다.

3. 정적 권한검사 분석

이 장에서는 본 논문에서 구현한 권한검사 시각화 시스템의 기반이 되는 정적 권한검사 분석(static permission check analysis)을 소개한다[2]. 정적 권한검사 분석에서는 각 메소드 별로 도달 가능한 권한검사 집합과 항상 성공하거나 항상 실패하는 권한검사 집합을 계산한다.

본 논문에서는 checkPermission(p) 함수를 *check*(*p*)로 간략히 표현하고, *check*(*p*)를 메소드 *m*에서 호출했을 때 *check*(*p*) ∈ *m*으로 표현할 것이다. 정책파일에 의해, 각 메소드 *m*에 주어진 퍼미션들의 집합은 *Permission*(*m*)으로 나타낸다. 본 논문에서는 다음과 같은 호출 그래프(call graph)를 기반으로 정적 권한검사 분석을 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.

메소드에서 성공 가능한 권한검사를 다음과 같이 정의하고, **May-Succeed Check(May-SC) Analysis**를 통해 각 메소드의 엔트리에서 성공 가능 검사(may-succeed check)을 계산한다.

[정의 2] 호출 그래프 상에서 메소드 *n*에서부터 *check*(*p*)를 포함하고 있는 메소드 *m*까지 경로가 존재하고, 이 경로 상에 있는 모든 메소드가 퍼미션 *p*에 대한 권한을 가지고 있을 때 메소드 *n*의 엔트리(entry)에서 이 권한검사 *check*(*p*)는 성공 가능(may-succeed)하다.

다음은 **May-Succeed Check(May-SC) Analysis**를 정의하는 흐름 식(flow equation)이다.

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

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

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

$May\text{-}SC_{entry}(n)$은 메소드 n에서 허용되는 퍼미션 p에 대해서, 메소드 n에서의 $check(p)$와 $May\text{-}SC_{exit}(n)$에 있는 $check(p)$

를 포함한다.

위 식은 함수 $F_{May-SC}: L \rightarrow L$을 정의하고, L은 $L_{entry} \times L_{exit}$이다. 여기서 L_{entry}와 L_{exit}은 노드 N에서 2^{check}으로의 함수 집합이다. 프로그램 내에 있는 권한검사들의 집합과 L이 유한하고, F_{May-SC}는 단조 증가하기 때문에 함수의 $lfp(F_{May\ SC})$는 유한 시간 내에 계산할 수 있다[6]. 따라서 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.

만약 어떤 메소드 n에 도달하는 권한검사가 성공 가능 검사(may-succeed check)가 아니면, 메소드 n의 엔트리에서 항상 실패한다. 메소드 m에 있는 $check(p)$가 메소드 n의 엔트리에서 항상 실패한다면, 이것은 퍼미션 p를 만족하는 n에서 m까지의 호출 경로가 존재하지 않는다는 것을 의미한다.

메소드 n에서의 성공 가능 검사인 $may-sc_{entry}(n)$을 계산하면, 메소드 n에서 항상 실패하는 권한검사인 $must-fc_{entry}(n)$는 다음과 같이 쉽게 계산할 수 있다.

$$\textbf{must-fcentry(n) = rc(n) - may-scentry(n)}$$

위 식에서 rc(n)은 퍼미션을 고려하지 않고, 메소드 n에 도달할 수 있는 도달 가능 검사(reachable check)이며, 이를 계산하기 위한 흐름 식은 다음과 같다. rc(n)은 노드 n에서 이 식의 해이다.

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

각 메소드에서 스택 인스펙션을 항상 통과하는 권한검사들을 계산하는 **Must-Succeed Check(Must-SC) Analysis**는 다음과 같이 정의한다.

[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 항상 성공(must-succeed)한다.

만약 어떤 메소드 n에 도달하는 권한검사가 항상 성공 검사(must-succeed check)가 아니면, 권한검사 $check(p)$에서부터 메소드 n까지 퍼미션 p를 만족하지 않는 경로가 존재하는 것이다. 이렇게 메소드에서 실패 가능한 권한검사를 실패 가능 검사(may-fail check)라 정의한다.

각 메소드에서의 실패 가능 검사(may-fail check)를 계산하는 **May-Fail Check(May-FC) Analysis**을 정의하는 흐름 식(flow equation)은 다음과 같다.

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

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

$$where\ gen_{May-FC(n)} = \{check(p) \in rc(n) \mid p \not\in Permission(n)\}$$

$May-FC_{entry}(n)$는 $May-FC_{exit}(n)$에 있는 모든 권한검사 $check(p)$와 메소드 n에 도달하는 권한검사 중에서 n에서 실패하는 $check(p)$를 포함한다.

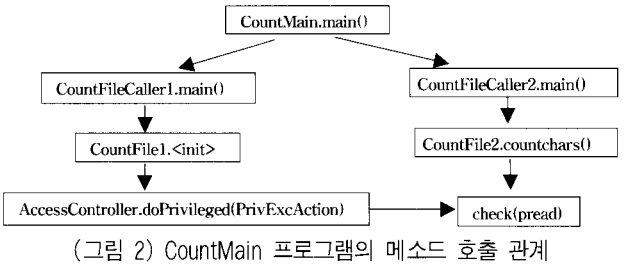
위 식은 함수 $F_{May-SC}: L \rightarrow L$을 정의하고, 이 식의 해인 (may-fc$_{entry}$, may-fc$_{exit}$)$\in L$은 유한시간 내에 $lfp(F_{May-FC})$에 의해 계산된다.

위 식에 의해서 실패 가능 검사(may-fail check)를 계산하고 나면 다음과 같이 항상 성공 검사(must-succeed check)를 계산할 수 있다.

$$\textbf{must-scentry(n) = rc(n) - may-fcentry(n)}$$

(그림 2)와 같은 프로그램에 대해 정적분석을 해보겠다. CountMain은 접근 권한이 있으면 Lord.txt 파일에 있는 단어의 수를 세서 출력하는 프로그램이다. <표1>은 정책파일에 따라 각 프로텍션 도메인에 주어진 권한집합을 나타내고 있다.

다음 예에서 $check(p_{read})$는 Lord.txt 파일에 대한 읽기 권한이 있는지 검사한다. $check(p_{read})$는 권한이 있는 CountFile1, PrivExcAction, CountFile2의 메소드에서는 항상 성공 검사(must-succeed check)이고, 권한이 없는 CountFileCaller2의 메소드에서는 항상실패 검사(must-fail check)이다. 따라서 CountMain 프로그램이 CountMain.main()에서 시작되어 CountFile1.<init>을 실행하게 되면 $check(p_{read})$는 반드시 성공하게 되며, CountFileCaller2.main()을 실행하게 되면 $check(p_{read})$는 반드시 실패하고 SecurityException이 발생한다.



(그림 2) CountMain 프로그램의 메소드 호출 관계

〈표 1〉 정책파일

프로텍션 도메인	권한집합(Granted Permission)
CountMain	RuntimePermission "createSecurityManager" RuntimePermission"setSecurityManager"
CountFileCaller1 CountFileCaller2	∅
CountFile1 PrivExcAction CountFile2	FilePermission "C:/Lord.txt","read"

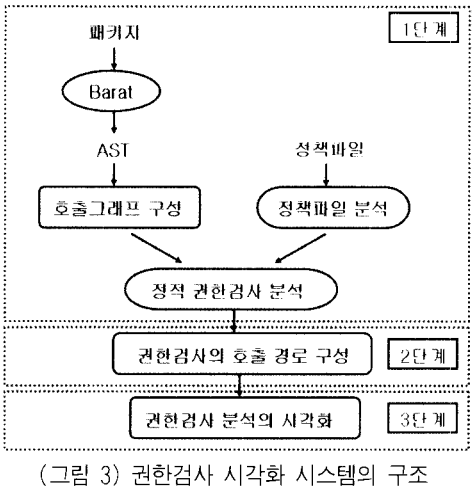
4. 시스템의 설계 및 구현

본 논문에서 구현한 권한검사 시각화 시스템은 (그림 3)과 같이 세 단계로 구성된다. 권한검사 시각화 시스템은 Java 컴파일러의 전단부인 Barat을 기반으로 구현되었다.

Barat은 Java 프로그램에 대해 이름과 타입 분석 정보를 포함하는 AST를 구성하며, AST의 각 노드를 방문하여 필요한 연산을 수행할 수 있도록 Visitor 디자인 패턴을 제공하고 있다[3].

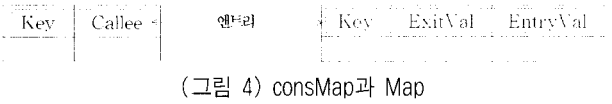
1단계 : 정적분석 및 해 구하기

① **정책파일 분석** : 입력받은 정책파일(policy file)로부터 각 메소드가 속한 프로텍션 도메인(protection domain) 별로 허용된 권한을 저장한다. 이 permMap의 내용을 바탕으로 각 메소드에서 권한검사가 성공하는지 실패하는지를 결정하게 된다.



(그림 3) 권한검사 시각화 시스템의 구조

② **호출 그래프 구성** : 호출 그래프를 구성하기 위해서 패키지 내의 각 메소드에서 호출하고 있는 메소드 집합을 계산하여 (그림 4)의 왼쪽에 있는 consMap에 저장한다.



(그림 4) consMap과 Map

③ **정적 권한검사 분석** : consMap과 퍼미션 정보를 이용하여 권한 검사 집합을 계산하여 (그림 4)의 오른쪽과 같은 형태의 Map에 저장한다.

Map은 consMap과 같은 키 집합을 가지며, ExitVal과 EntryVal는 각각 메쏘드의 entry와 exit 지점의 권한검사 집합을 저장한다.

(그림 5)의 알고리즘에 의해서 각 메소드에서 성공 가능 검사(may-succeed check)을 계산할 수 있다. 여기서 Map.ExitVal(k)은 Map에서 k를 key로 갖는 엔트리의 ExitVal 값을 의미한다. *Permission(n)*은 메소드 *n*에 주어진 권한집합이다.

알고리즘은 Map의 ExitVal과 EntryVal이 같아질 때까지 반복한다. (그림5)의 ①, ②에서는 메소드 k가 메소드 m을 호출했을 때(k → m) k의 exit값을 계산한다. k의 eixt 값(may-sc_{exit}(k))은 m의 엔트리 값(may-sc_{entry}(m))과 k에서 호출한 checkPermission() 중에서 k에서 성공하는 권한 검사를 포함한다. 이때, 메소드 m이 doPrivileged() 함수이면 이 과정을 생략한다. doPrivileged() 함수 내에서의 권한검사는 k의 호출자(caller)까지 도달되지 않기 때문이다. (그림5)의 ③에서는 메소드 k가 doPrivileged() 함수를 호출했을 때, k의 exit값을 계산한다.

```
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 권
                        한 검사를 Map.ExitVal(k) 에 추가;  }
                ② else if(m ==checkPermission(p) &&  p ∈
                                Permission(k)){
                                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
                        권한 검사를 Map.ExitVal(k)에 추가;  }
                }
        Map.ExitVal(k)를 Map.EntryVal(k)에 복사;     }
```

(그림 5) May-Succeed Check을 계산하는 알고리즘

실패 가능 검사(may-fail check)는 (그림5)의 알고리즘을 일부 수정하여 계산할 수 있다. (그림5)의 ①에서는 메소드 k가 메소드 m을 호출했을 때(k → m) m의 엔트리 값(may-sc_{entry}(m))을 k의 eixt 값(may-sc_{exit}(k))에 포함시킨다. ②에서는 k의 eixt 값(may-sc_{exit}(k))에 k에서 실패하는 도달 가능한 권한 검사를 포함시킨다.

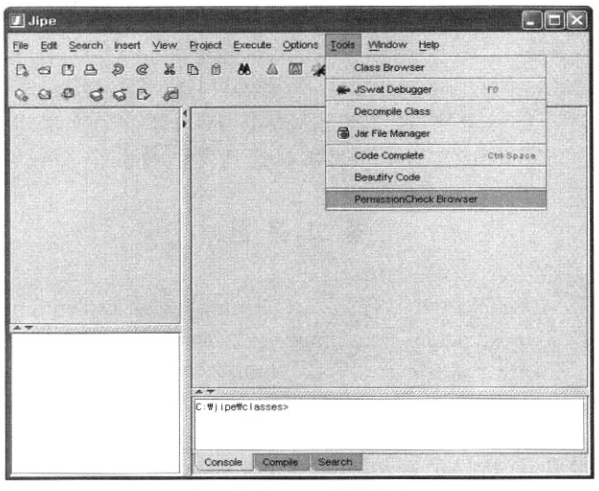
각 메소드의 도달 가능 검사(reachable check)에서 may-sc_{entry}(n)와 may-fc_{entry}(n)을 제외하여 각각 must-fc_{entry}(n)과 must-sc_{entry}(n)를 구한다.

2단계 : 권한 검사의 호출경로 구성

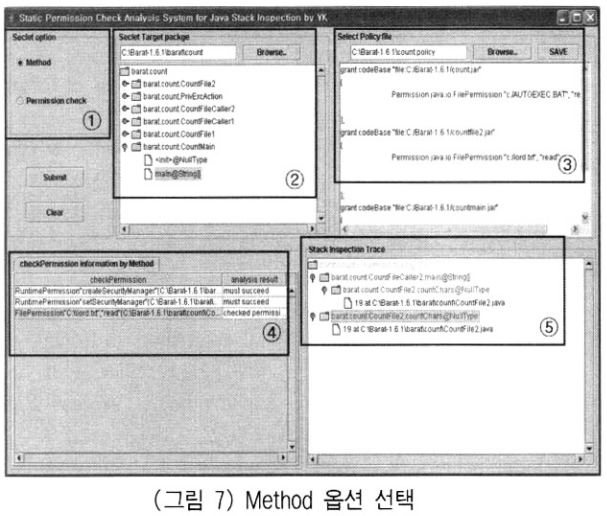
2 단계에서는 가능한 스택 인스펙션 과정을 추적해 볼 수 있도록, 메소드 호출 그래프에 기반하여 checkPermission() 함수가 호출되는 경로를 저장한다.

3단계 : 권한검사 분석의 시각화

본 논문에서는 Java 통합 프로그래밍 환경인 Jipe를 확장하여 권한검사 시각화 시스템을 구현하였다[5]. (그림 6)과 같이 Jipe의 Tools 메뉴에서 PermissionCheck Browser 메



(그림 6) Jipe 실행화면



(그림 7) Method 옵션 선택

뉴 아이템을 선택하면 권한검사 시각화 시스템을 실행할 수 있다.

(그림 7)은 Method 옵션을 선택한 시스템의 실행화면으로 각각에 대한 설명은 다음과 같다.

① 사용자가 옵션을 선택하는 창이다. Method 옵션을 선택하면 ②에서 관심있는 메소드를 선택하여 분석 정보를 확인할 수 있다. Permission Check 옵션을 선택하면 ②에서 관심있는 권한검사를 선택하여 분석 정보를 확인할 수 있다.

② 선택한 패키지 내의 클래스와 메소드들이 보여진다.

③ 선택한 정책파일의 내용이 보인다. 사용자는 필요에 따라 정책파일을 수정한 후, 저장할 수 있다.

④ 선택한 메소드에 도달 가능한 권한검사 집합, 항상 성공 또는 실패하는 권한검사 집합이 모두 보인다.

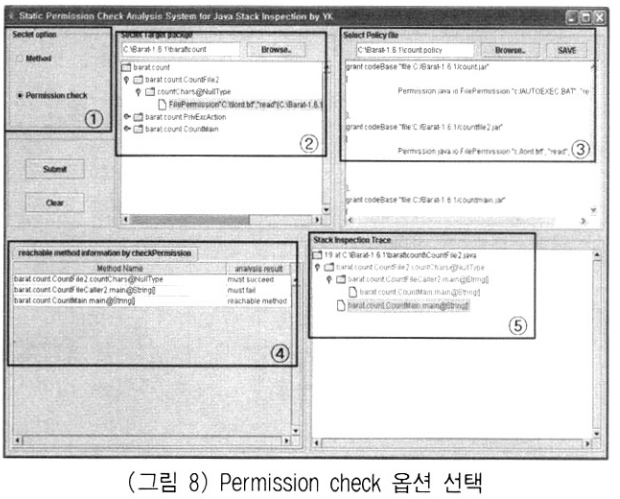
⑤ 선택한 메소드까지 권한검사가 도달되는 경로를 시각적으로 추적해볼 수 있다. 이때 권한검사가 항상 성공하는 메소드는 녹색으로, 항상 실패하는 메소드는 빨간색으로, 실행에 따라 결과가 달라지는 메소드는 오렌지색으로 표시된다.

(그림 8)은 Permission check 옵션을 선택한 시스템의 실행화면으로 각각에 대한 설명은 다음과 같다.

② 선택한 패키지 내의 권한검사 리스트가 보여지고 관심있는 권한검사를 선택할 수 있다.

④ 선택한 권한검사가 도달할 수 있는 패키지 내의 모든 메소드, 항상 성공하거나 항상 실패하는 메소드 집합이 보인다.

⑤ 선택한 권한검사가 스택 인스펙션하는 과정을 시각적으로 추적해볼 수 있다. 선택한 경로에서 권한검사가 성공하는 메소드는 녹색으로, 실패하는 메소드는 빨간색으로 표시된다.



(그림 8) Permission check 옵션 선택

5. 실험결과

이 장에서는 권한검사 시각화 시스템에서 다섯 개의 Java 패키지를 분석한 결과를 소개한다.

<표 2>는 각 패키지를 시스템에서 분석한 결과를 정리한 것으로, 이 결과는 정책파일에 따라 달라질 수 있다. 패키지 내의 전체 권한검사 수와 main() 메소드에 도달되는 권한검사 중에서 항상 성공하거나 항상 실패하는 항상 성공 검사(must-succeed check), 항상 실패 검사(must-fail check) 수를 표에 나타냈다. 패키지 내의 전체 권한 검사 중에서 doPrivileged() 함수 내에서의 권한 검사는 main() 메소드까지 도달하지 않는다.

〈표 2〉 실험결과

프로그램 이름	전체	항상 실패 검사 수(must fail checks)	항상 성공 검사 수(must-succeed checks)	도달 가능 검사 수(reachable checks)
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

실험 대상이 된 각 패키지에 대한 설명은 다음과 같다.

(1) **CountMain 패키지** : SecurityManager 객체를 생성하여 시스템에 등록하고 두 개의 파일에서 단어의 수를 세서 출력하는 프로그램이다. main() 메소드에 도달 가능한 권한 검사 중에서 1개의 권한검사는 실행되면 반드시 실패하여 보안 예외를 발생하는 항상 실패 검사(must-fail check)이고 두 개의 권한검사는 실행되면 반드시 성공하는 항상 성공 검사(must-succeed check)이다. main() 메소드까지 도달되지 않는 두 개의 권한검사는 doPrivileged() 함수 내에서의 권한검사이다.

(2) **BankSystem 패키지** : 은행의 고객이면 고객과 계좌정보를 기록하고 있는 파일에 접근하여 예금, 출금 등의 은행 업무를 수행한 후 새로운 정보를 파일에 기록하는 프로그램이다. main() 메소드에서 Client 클래스의 객체에 의한 파일 접근은 성공하며, Guest() 클래스의 객체에 의한 파일 접근은 실패하고 보안 예외가 발생한다. 따라서 main() 메소드에서 파일에 대한 접근 권한 검사는 실행할 때마다 성공할 수도 있고, 실패할 수도 있다.

(3) **StringSearch 패키지** : 10개의 파일에서 "string"이라는 문자열이 나타나는 횟수를 세는 프로그램이다. 정책파일에 접근권한이 명세되어 있는 5개의 파일에 대한 권한검사는 항상 성공하고, "string" 문자열이 나타나는 횟수를 세서 출력한다. 나머지 접근권한이 없는 5개의 파일에 대한 접근은 항상 거부되고 보안예외가 발생한다.

(4) **getProps 패키지** : user, os, 파일 시스템에 관련된 시스템 정보를 읽어오는 프로그램이다. main() 메소드에 도달 가능한 7개의 권한검사 중에서 접근권한이 없는 시스템 정보를 읽어오려고 했을 때, 접근이 거부되고 보안예외가 발생하게 되는 항상 실패 검사(must-fail check)이다.

(5) **Server-Client 패키지** : 서버와 클라이언트가 소켓을 생성하고 서버에서 파일의 내용을 읽어서 클라이언트에게 보내면 클라이언트가 받은 메시지를 자신의 로컬 파일에 저장하는 프로그램이다. Client 클래스의 main() 메소드에는 Server로부터 받아온 메시지를 저장할 파일에 대한 접근권한이 있는지 검사하여 실패한다. Client 클래스가 자신의 로컬 파일에 메시지를 저장할 수 있도록 하려면 정책파일에 이에 대한 권한을 명세해 주어야 한다.

6. 결 론

본 논문에서 구현한 시스템은 권한검사와 정책파일에 대한 정적분석 결과를 시각화하여 보여줌으로써 프로그래머가 자원에 대한 적절한 접근권한 관리를 하는데 편의성을 제공한다. 프로그램의 각 지점에 도달가능하거나 항상 성공 또는 항상 실패하는 권한검사를 계산하여 보여주고, 권한검사가 실행되었을 때 스택 인스펙션이 어떻게 수행되는지 시각적으로 추적해 볼 수 있다. 프로그래머는 프로그램이나 정책파일을 수정한 후, 반복적으로 분석결과를 확인해 봄으로써 자신의 프로그램에 완전한 보안 정책을 세울 수 있다.

참 고 문 헌

[1] M. Bartoletti, P. Degano, G. L. Ferrari. "Stack inspection and secure program transformations", International Journal of Information Security Vol.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.

[4] http://java.sun.com/j2se/1.5.0/docs/api

[5] S. Koleth, M. Hansen, R. Zsolt. open source GPL license, http://jipe.sourceforge.net.

[6] F.Nielson, H. R. Nielson, C. HanKin, 'Principles of Program Analysis'. pp.363-390, Springer, 2005.

[7] Horstmann, Cay S, G. Cornell, 'Core Java 2', Vol.2, Advanced Features (4th Edition), Sun Microsystems, 2000.

[8] C. Fournet and A. D. Gordon. "Stack inspection : Theory and Variants", Symposium on Principles of Programming Languages, 2001.

[9] M. Bartoletti, P. Degano, and G. L. Ferrari. "Static Analysis for Stack Inspection", International Workshop on Concurrency and Coordination, Vol.54 of ENTCS. Elsevier, 2001.

[10] L. Koved, M. Pistoia, A. Kershenbaum. "Access Rights Analysis for Java", OOPSLA 2002.

[11] Ulfar Erlingsson, Fred B. Schneider. "IRM Enforcement of Java Stack Inspection", IEEE Symposium on Security and Privacy, 2000.

김 윤 경

e-mail : ykkim79@sookmyung.ac.kr

2002년 8월 숙명여자대학교 컴퓨터과학과 (이학사)

2003년~현재 숙명여자대학교 컴퓨터과학과 석사

관심분야 : 프로그래밍 언어 및 시스템

창 병 모

e-mail : chang@sookmyung.ac.kr

1988년 서울대학교 컴퓨터공학과(공학사)

1990년 한국과학기술원 전산학과(공학석사)

1994년 한국과학기술원 전산학과(공학박사)

1994년 한국전자통신연구원 연구원

2002년 IBM Watson 연구소 방문과학자

2003년 Univ. of Pennsylvania 방문과학자

1995년~현재 숙명여자대학교 컴퓨터과학과 교수

관심분야 : 프로그래밍 언어 및 시스템, 유비쿼터스 소프트웨어