자바 바이트 코드에서 정보흐름 보안성 검사

박사천° 이정림 권기현
경기대학교 정보과학부
{sachem°, leejrim, khkwon}@kyonggi.ac.kr

Checking Security of Information Flow on Java Byte Code

Sachoun Park°, Jungrim Lee, Gihwon Kwon
Department of Computer Science, Kyonggi University

요 약

현재 정보보호시스템의 에러로 많은 피해를 입고 있다. 특히 고 보안성이 요구되는 시스템에서의 정보 누출이 심각한 문제가 되고 있는데 이러한 시스템의 예로 스마트카드가 있다. 스마트카드는 오랜 시간 축적된 경험과 연구로써 그 보안성이 특히 우수하다고 할 수 있다. 그러나 애플리케이션 내에서 그리고 애플리케이션 간의 정보흐름의 보안성은 보장되지 않는다. 따라서 개인 정보 누출의 위협이 존재한다. 소스프로그램 차원에서 보안적 정보흐름을 검사하는 연구들이 많이 진행되어 왔고 자동도구인 모델체커를 이용하는 연구들도 점차 증가하고 있다. 이런 연구들의 연장선상에서, 본 논문에서는 SMV 모델체커를 이용해 자바 바이트코드에 대한 정보흐름의 보안성을 검사하는 방법을 보인다.

1. 서 론

인류는 정보보호시스템의 사용으로부터 많은 이점을 얻고 있는 반면에, 또한 정보보호시스템의 에러로 인해서도 많은 피해를 입고 있다. 특히 정보가 누출되면 치명적인 손실을 야기하는 고 보안성(security-critical) 정보보호시스템의 사용이 크게 증대되고 있다. 대표적인 예로서 스마트카드가 있다. 스마트카드란 기존의 플라스틱 카드의 자기 띠를 반도체 칩으로 대체하고 이를 통해 카드의 보안성, 편의성, 경제성 및 다기능성 등을 획기적으로 개선시킨 새로운 개념의 카드를 말한다. 특히, 반도체 칩을 탑재한 칩 카드 중에서도 단지 데이터의 저장 기능만을 강화한 메모리카드보다는 자체적으로 CPU 및 논리연산 회로등을 장착해서 소형 컴퓨터의 기능을 수행할 수 있는 카드이다. 스마트카드의 뛰어난 보안성 및 다양한 기능으로 인하여 금융, 정보통신, 공공부문등에서 그 이용 범위가 광범위하며 보급률이 증대되고 있다. 스마트카드의 보안 문제는 다중 애플리케이션과 카드 발급 후 코드 다운로딩 때문에 점점 더 복잡해지며 중요해지고 있다. 애플리케이션 개발 플랫폼의 보안성은 신뢰적인 서비스를 제공해야 하는 운영체제 보안성을 기반으로 만들어졌다. 그러나 객 애플리케이션 내에서 그리고 애플리케이션 간의 정보흐름의 보안성은 보장되지 않는다. 따라서 개인 정보 누출의 위협이 존재한다[1].

비 보안적 정보흐름이란 보안 레벨이 높은 정보를 담은 변수로부터 보안 레벨이 낮은 변수로의 정보흐름이라고 할 수 있다. 여기에는 크게 두 가지 종류의 정보누수가 존재한다. 첫째로 비 보안적 정보흐름에 기인하는 것으로 명시적 흐름과 묵시적 흐름에 의한 누수가 존재한다. 그 예로 'a := x' 배정문에서 a는 낮은 보안 레벨의 변수이고 x는 높은 보안 레벨의 변수일 때, 위 배정문은 명시적으로 보안성이 누수 되는 것을 알 수 있다. 한편 묵시적인 정보흐름의 예로 'if x=0 then a:=1 else a:=0' 조건문에서 a의 결과 값은 x 값을 미루어 짐작케 한다. 이것은 if 조건부분에 높은 보안 레벨을 갖는 변수가 사용될 때, else 혹은 then 부분에 낮은 보안 레벨의 식이 사용되는 경우에 해당하고 이것은 if문의 모든 영역 안에서 검사되어야 한다.

또한 프로그램의 비 보안적 종료에 의한 것으로, 프로그램이 'while (x > 0) do skip'일 때, 프로그램이 무한이 수행된다면 x가 0보다 크다는 사실과 같이 프로그램의 종료가 보안성이 높은 변수의 값을 추측할 수 있게 하는 것이다[2].

논문의 구성은 2장에서 관련연구, 3장에서는 사용되는 바이트코드의 구문과 의미를 설명한다. 4장에서 바이트코드 프로그램 모델에 대해 기술하고 5장에서는 바이트코드 모델체킹 방법을 보이고, 6장에서 결론 및 향후 연구 과제를 설명한다.

2. 관련연구

본 논문은 크게 두 가지의 관련 연구들이 있다. 우선 바이트코드 검증에 대한 관련연구로는 [1,3,4]등이 있다. J. Posegga를 중심으로 한 그룹에서는 바이트코드의 의미를 ASM(Abstract State Machine)으로 변형하고 이를 다시 SMV로 옮기는 자동화된 단계를 개발했었다. 그러나 몇몇 바이트코드로 제한되는 점이 있었고 하나의 메쏘드에서 타입이나 스택에 대해 검사하였다. PACAP 프로젝트를 수행한 그룹에서는 다중 애플릿간의 상호작용에서 보안 위협을 식별하고 이를 검증하기 위한 노력으로 래티스 구조의 보안 정책모델을 만들고 SMV로 검증하는 단계로 평가를 수행하였다. 마지막으로 G. Chugunov는 바이트코드 레벨의 애플릿 모델 체킹을 위한 JCAVE(JavaCard Applet Verification Environment)라 불리는 프레임워크를 개발했다. 여기서 S. Schwoon는 Moped 모델체커를 개발했는데, 제어흐름 그래프를 푸시다운 오토마타로 변형한 모델과, LTL(Linear Temporal Logic)식을 변형한 부키 오토마타를 속성으로 하여 모델 체킹한다. 이러한 변환 과정에서 각각 ML과 Spin 모델체커를 이용한다.

정보흐름의 보안성 검증에 관한 연구로는 [2,5,6]이 있다. [5]는 명령형 프로그램에서 정보 흐름의 안전성 분석을 모델체킹으로 하였다. 이를 위해 정보흐름의 안정성에 대한 CTL 논리식을 정의하고, 프로그램의 정보흐름만을 표현한 FlowGraph로부터 SMV 검사를 수행했다. 정보누출의 정확성을 위해 순방향과 역방향으로 검사했지만, 이것은 모두 정보누출에 관한 것이고 안전하게 종료하는가 하는 속성은 다루지 않았다. [2]에서도 역시 프로그램의 구문과 의미, 보안성 있는 정보흐름 등을 정의하고 CTL식으로 속성을 표현하고 프로그램 P를 크립키 구조 K(P)로 변형하여 SMV 모델체커로 보안성을 검사하였다.

* 본 연구는 2004년 한국정보보호진흥원 정보보호시스템 보안정책모델 평가방법론 연구 지원에 의하여 수행되었음.

위 두 논문에서 각 변수들에 대한 보안 레벨을 Low와 High로 나누었고 다중 레벨로의 확장이 가능하다. [6]은 요약 해석을 이용해서 자바 바이트코드에서 다중 레벨의 정보흐름 보안성을 검사했다. 그러나 안전하게 종료하는가 하는 속성은 다루지 않았고 일부의 바이트코드만을 다루었다.

3. 바이트코드의 구문과 의미

3.1 프로그램 구문

본 논문에서 사용할 바이트코드와 코드에 대한 간단한 설명은 아래와 같다.

op: 스택에서 두개의 오퍼랜드를 꺼내서(pop) 연산(op)을 수행한 후 그 값을 다시 스택에 넣는다(push). 여기서 연산이란 산술연산 명령어들(add, sub, div, mul)과 논리연산 명령어들(and, or, xor)을 대표한다.

pop: 스택의 최상단(top)의 값을 제거한다.

push k: 상수 k의 값을 스택에 쌓는다. 이 명령어는 상수를 스택에 쌓는 다른 명령어들(bipush, sipuch, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, ldc, ldc_w)을 대표한다.

load i: 지역변수 배열 m의 i번째 값 m(i)를 스택에 쌓는다. 이 명령어는 변수를 스택에 쌓는 모든 명령어들(aload, aload_0, aload_1, aload_2, aload_3, iload, iload_0, iload_1, iload_2, iload_3)을 대표한다.

store i: 스택의 최상단의 값을 빼서 지역변수 배열 m(i)에 그 값을 넣는다. 이 명령은 다음 명령어들을 대표한다: astore, astore_0, astore_1, astore_2, astore_3, istore, istore_0, istore_1, istore_2, istore_3.

if i: 스택의 최상단의 값을 0과 비교해서 i의 위치로 프로그램 카운터를 옮기는 명령들(ifeq, ifne, ifge, ifgt, ifle, iflt, ifnull, ifnonnull)을 대표한다.

if_cmp i: 스택 상단의 두 값을 비교해서 i의 위치로 프로그램 카운터를 옮기는 명령들(if_icmpeq, if_icmpne, if_icmpge, if_icmpgt, if_icmple, if_icmplt, if_acmpeq, if_acmpne, if_acmpge, if_acmpgt, if_acmple, if_acmplt)을 대표한다.

goto i: 단순히 프로그램 카운터를 i의 위치로 옮긴다.

nop: 아무것도 수행하지 않는 명령이다. 실제로 이 명령은 neg, rem, shl, shr, ushr, i2b, i2s, nop 등을 대표한다. 왜냐하면 단항 연산자들은 정보흐름의 보안성에 아무런 영향을 주지 않는다고 판단했기 때문이다.

return: 이 명령어는 실제 메쏘드의 수행이 종료하면 메쏘드를 호출했던 스택 프레임으로 돌아가는 명령이다. 하지만 우리는 단순이 메쏘드의 종료를 알리는 명령으로 사용하겠다. 이 명령어 역시 return, ireturn, areturn의 명령들을 대표한다.

여기서 다루어지는 바이트코드 명령어는 기본적으로 자바카드 가상머신의 명령어 집합을 기반으로 한다[7]. 제외된 명령들은 예외상황을 다루는 jsr/ret 명령과 객체관련 명령들, 배열 관련 명령들이다. 자바카드 가상머신의 모든 명령을 다루는 것은 향후연구과제가 될 것이다.

3.2 프로그램의 의미

우리는 프로그램의 의미를 전이 시스템으로 표현한다. 그 경로는 프로그램의 실행 경로이며 각 노드는 프로그램의 상태변화 즉 스택과 지역변수 배열의 변화 등을 나타낸다. 이 프로그램의 가정 사항은 스택의 언더플로우나 오버플로우가 존재하지 않는다는 것과 *goto*명령어나 if 명령어에 의해서 존재하지 않는 프로그램 카운터로 진출할 수 없다는 것이다.

정의 1. 전이 시스템은 $T = (S, R, s_0)$이다. 여기서 S는 상태들의 집합이고, $s_0 \in S$는 초기 상태를 나타낸 다. $R \subseteq S \times S$은 전이관계이며, $(s, s') \in \rightarrow$으로도 나타내고 $s \rightarrow s'$로 표기한다.

앞 절에서 설명한 구문에 대한 의미가 그림 1과 같다. 상수 값에 대한 도메인 C^e는 데이터(k_1, k_2, ...)와 프로그램 카운터(l, l_1, l_2, ...)를 위해 사용된다. V가 프로그램에서 사용되는 변수의 집합일 때, 변수에 값을 주는 함수 $M^e: V \rightarrow C^e$에는 m, m', m_1 ...등이 있다. 그리고 값에 대한 유한 시퀀스인 스택에 대한 도메인 $S^e = (C^e)^*$은 s, s', s_1, ...등이 있다. 주어진 지역변수 배열 m에 대해서, 그 도메인은 $D(m) \subseteq C^e$로 정의한다.

실행되는 프로그램의 상태는 튜플 $\langle l, m, s \rangle$로 표현한다. 여기서 l은 프로그램 카운터를 의미하고, m은 지역변수의 현재 상태, s는 오퍼랜드 스택의 현재 상태를 가리킨다. *byte*[l]를 l번째 명령어라고 한다면 항상 프로그램의 시작은 *byte*[1]이고, 스택은 비어있는 것으로 가정한다. 지역변수 배열 m에는 첫 번째 원소부터 매개변수가 차례로 저장되고 나머지 원소는 정의되지 않은 것으로 가정한다. Q^e를 상태들의 도메인이라고 한다면, 프로그램의 의미 규칙들은 전이관계 $\rightarrow^e \subseteq Q^e \times Q^e$ 를 정의한다. 표기법 $m[k/i]$은 m의 인덱스 i에 해당하는 값을 k로 변경하는 것을 의미한다. $cmp(k_1, k_2)$는 두 값을 비교하는 술어이다. 주어진 바이트코드 프로그램 *byte*와 지역변수 배열 m에 대해서, 프로그램의 제어 흐름은 전이 시스템 (Q^e, $\rightarrow^e$, 1, m, λ)으로 정의된다. 여기서 λ는 빈 시퀀스를 의미한다. 시퀀스 접합(concatenation) 연산자 $\cdot$는 스택을 처리하기 위해 사용되었다. 프로그램의 마지막 상태는 유일하며 계속해서 자기 자신에게로 전이가 발생한다. 그 명령어는 *byte*[*final*] = *return*이 된다.

$$op \quad \frac{byte[l] = op}{\langle l, m, k_1 \cdot k_2 \cdot s \rangle \rightarrow^e \langle l+1, m, (k_1 \, op \, k_2) \cdot s \rangle}$$

$$pop \quad \frac{byte[l] = pop}{\langle l, m, k \cdot s \rangle \rightarrow^e \langle l+1, m, s \rangle}$$

$$push \quad \frac{byte[l] = push\ k}{\langle l, m, s \rangle \rightarrow^e \langle l+1, m, k \cdot s \rangle}$$

$$load \quad \frac{byte[l] = load\ i}{\langle l, m, s \rangle \rightarrow^e \langle l+1, m, m(i) \cdot s \rangle}$$

$$store \quad \frac{byte[l] = store\ i}{\langle l, m, k \cdot s \rangle \rightarrow^e \langle l+1, m[k/i], s \rangle}$$

$$\text{if}_{false} \quad \frac{byte[l] = \text{if}\ l \ \neg cmp(k, 0)}{\langle l, m, k \cdot s \rangle \rightarrow^e \langle l+1, m, s \rangle}$$

$$\text{if}_{true} \quad \frac{byte[l] = \text{if}\ l \ cmp(k, 0)}{\langle l, m, k \cdot s \rangle \rightarrow^e \langle l, m, s \rangle}$$

$$goto \quad \frac{byte[l] = goto\ l}{\langle l, m, s \rangle \rightarrow^e \langle l, m, s \rangle}$$

$$nop \quad \frac{byte[l] = nop}{\langle l, m, s \rangle \rightarrow^e \langle l+1, m, s \rangle}$$

$$return \quad \frac{byte[l] = return}{\langle l, m, s \rangle \rightarrow^e \langle l, m, s \rangle}$$

$$\text{if}\_cmp_{false} \quad \frac{byte[l] = \text{if}\_cmp\ l \ \neg cmp(k_1, k_2)}{\langle l, m, k_1 \cdot k_2 \cdot s \rangle \rightarrow^e \langle l+1, m, s \rangle}$$

$$\text{if}\_cmp_{ture} \quad \frac{byte[l] = \text{if}\_cmp\ l \ cmp(k_1, k_2)}{\langle l, m, k_1 \cdot k_2 \cdot s \rangle \rightarrow^e \langle l, m, s \rangle}$$

그림 1. 프로그램의 의미

4. 바이트코드 프로그램 모델

보안 속성을 기술하기 위해서 우리는 바이트코드 프로그램의 각 명령들에 보안 레벨을 부여한다. 본 논문에서 보안 레벨은 단순하게 두개의 레벨로 정의한다. 보안 레벨 $L = \{low, high\}$ 은 두개의 원소를 가지는 집합이다. 보안 레벨을 부여하는 함수 *level*은 프로

그램의 의미에 의해서 다음과 같이 정의 된다.

$$level(k) = \begin{cases} \text{if } k=0, & low \\ otherwise & high \end{cases}$$
$$level(m(i)) = \begin{cases} \text{if } m(i)=0, & low \\ otherwise & high \end{cases}$$
$$level(k_1 op k_2) = level(k_1) \sqcap level(k_2)$$
$$level(m[k/i]) = \begin{cases} \text{if } m(i)=0 \wedge k=1, & low \\ otherwise & high \end{cases}$$

여기서 ⊔은 최저 상한(least upper bound)을 의미한다. 이제 바이트코드의 프로그램에 대한 모델을 크립키 구조 K로 정의하면 다음과 같다.

정의 2. 크립키 구조 $K = (Q, q_0, R, Label, AP)$이다. Q는 상태들의 집합으로 프로그램의 구문과 프로그램 카운터 l에 대해서 q_l아래와 같이 튜플 $PC, INS, LEVEL$ 로 정의된다. k_1과 k_2는 각각 스택에 최상단의 값과 그 아래의 값이다.

$$q_l(op) = \langle l, op, level(k_1 op k_2)\rangle$$
$$q_l(pop) = \langle l, pop, \rightarrow\rangle$$
$$q_l(pushk) = \langle l, pushk, low\rangle$$
$$q_l(load\ i) = \langle l, load\ i, level(m(i))\rangle$$
$$q_l(store\ i) = \langle l, store\ i, level(m[k_1/i])\rangle$$
$$q_l(\text{if } i) = \langle l, \text{if } l', level(k_1)\rangle$$
$$q_l(\text{if } \_cmp\ i) = \langle l, \text{if } \_cmp\ l', level(k_1 op k_2)\rangle$$
$$q_l(goto\ l') = \langle l, goto\ l', \rightarrow\rangle$$
$$q_l(nop) = \langle l, nop, \rightarrow\rangle$$
$$q_l(return) = \langle l, return, \rightarrow\rangle$$

$q_0 \in Q$는 초기상태이고, $R \subseteq Q \times Q$ 은 전이관계이며 전체 관계(total relation)이다. 전이관계를 $q_l \rightarrow q_{l+1}$ 로도 표현하고 q_{l+1}은 q_l의 바로 다음에 수행되는 명령이 된다. 전이 관계는 바이트코드 명령에 따라서 결정적으로 형성된다. $Label: Q \rightarrow 2^{AP}$는 상태에 원소 명제의 부분집합을 레이블 하는 함수이다.

5. 정보흐름의 보안 속성 검사

정보흐름의 보안성은 정보 흐름에 관한 보안성과 프로그램 종료에 관계된 보안성으로 나누어 볼 수 있다. 정보흐름에 대한 보안성은 다시 명시적인 흐름과 묵시적인 흐름으로 나뉜다. 고차원의 언어에서 명시적인 흐름에 대한 조사는 배정문이 발생하는 곳을 조사하여 배정문 오른쪽의 보안 레벨이 어떠한가를 알아보는 것으로 해결되었다. 그러나 바이트코드 차원에서는 스택을 사용하기 때문에 우리는 지역변수 배열에 저장하는 store 명령을 중심으로 명시적 정보흐름에 관해서 조사한다. 묵시적 정보흐름은 약간 복잡한 측면이 있다. 고차원의 언어에서는 구조적 정보가 유지되기 때문에 if문의 영역을 쉽게 확인할 수 있고 그 영역 안에서 검사가 완료될 수 있었다. 그러나 바이트코드 차원에서는 if문의 영역을 간단히 판별하기가 어렵다. 왜냐하면 if문에서 명시된 프로그램 카운터와 if 이후에 다른 if 없이 만나는 goto 문의 프로그램 카운터에 의해서 장정적으로 영역이 판별되기 때문이다. 마지막으로 프로그램 종료에 의한 정보누수(covert flow)는 고차원 언어에서 while문에 낮은 보안 레벨의 변수가 사용되는 것을 허용하지 않고 만일 if 문에 높은 보안 레벨의 변수가 사용되었다면 반드시 종료됨을 확인함으로써 막을 수 있다. 그러나 바이트 코드에서는 while문의 구조가 if문과 goto문의 순환되는 구조로 나타나기 때문에 이를 고려해서 검사해야 한다. SMV에서 이를 검사하기 위해서 축약된 바이트코드를 의미에 따라크립키 구조로 변형한 후, 정보흐름 속성을 CTL식으로 정의한다. SMV 모델을 작성할 때, 바이트코드에서 사용된 변수들은 보안등급에 의해 추상화된다.

```
-- explicity flow
 SPEC AG(ins = store -> levelof = high)
-- implicity flow
SPEC AG(pc=5 & ins=if_cmp & levelof=high -
    > !E[!(ins = END_5) U (ins=store &
    m[index] = 0)])
SPEC AG(pc=17 & ins=if_cmp & levelof=high
    & EG AF(pc=17 & ins=if_cmp &
    levelof=high) -> EX(A[(ins=store ->
    m[index]=1)U(pc=17 & ins=if_cmp &
    levelof=high)]))
-- covert flow
SPEC AG(pc=5 & ins=if_cmp & levelof = high
    -> AF (ins=END_5))
SPEC AG(pc=17 & ins=if_cmp & EG EF(pc=17
    & ins=if_cmp) -> levelof = low)
```

그림 2. 정보흐름 속성에 대한 CTL 명세

6. 결론 및 향후 연구

우리는 보안성이 극히 요구되는 분야인 스마트카드에서 애플리케이션에서 정보흐름의 보안성 문제를 다루었다. 본 연구는 정보보호 진흥원의 과제수행의 일환으로 이루어졌으며, 공통평가기준 기반의 고등급을 평가를 위한 자바 바이트코드 검증이다. 실제의 애플릿들은 바이트코드로 제공되기 때문에 앞으로 빈번하게 사용될 스마트카드와 그 위에서 동작하게 될 애플릿에 대한 바이트코드 검증은 시급하게 요구되는 연구 분야이다.

우리는 우선 검증을 쉽게 하기 위해 검증할 바이트코드의 범위를 좁혀서 구문과 의미를 정의하고, 안전한 정보흐름을 정의한 후에 프로그램에 대한 SMV 모델을 만들어 몇몇 예제에 대한 실험을 수행했다.

본 연구는 현재 진행 중에 있으며, 분석 방법의 정확성을 증명하는 문제와 자바카드에서 사용되는 모든 바이트코드로 구문과 의미를 확장하고, 검증을 단계를 자동화하는 연구가 남았다.

참고문헌

[1] P. Bieber, J. Cazin , P. Girard, JL Lanet, V. Wiels , G. Zanon, "Checking secure interactions of smart card applets: extended version," Journal of Computer Security archive Volume 10 , Issue 4 (December 2002) table of contents Special issue on ESORICS 2000. Pages: 369 - 398. Year of Publication: 2002

[2] N. De Francesco, G. Lettieri, Checking security properties by model checking, SOFTWARE TESTING VERIFICATION & RELIABILITY, vol. 13, pp. 181-196, 2003.

[3] J. Posegga and H. Vogt, Java Byte Code Verification Using Model Checking. Proc. OOPSLA Workshop on Formal Underpinnings of Java, 1998.

[4] http://www.sics.se/fdt/vericode/jcave.html

[5] 조영갑, 도경구, 신승철, SMV를 이용한 정보 흐름 안전성의 모델 검사, 프로그래밍언어논문지, 제15권, 제2호, 39-48쪽, 2001년 11월.

[6] M. Avvenuti, C. Bernardeschi, N. De Francesco, Java bytecode verification for secure information flow, ACM SIGPLAN NOTICES, num. 12, vol. 38, pp. 20-27, 2003.

[7] T. Lindhilm and F. Yellin, Java Card™ 2.1 Virtual Machine Specification, Sun Microsystems, Inc., 1999.

[8] R. Barbuti, C. Bernardeschi, N. De Francesco. Checking Security of Java Bytecode by Abstract Interpretation, ACM Symposium on Applied Computing (SAC2002), Madrid 2002.

[9] E.M. Clarke, O. Grumberg, and D Peled, Model Checking MIT Press, 1999.