KVM 상에서 효율적인 검증을 위한 스택맵 분석

설경수°, 진민식, 권오형, 정민수
경남대학교 컴퓨터공학과

Analysis of stack map for efficient verification on the KVM(Kilobyte Virtual Machine)

KyungSu Seol, Minsik Jin, Ohhyung Kweon, Minsu Jung
Dept. of Computer Engineering, Kyungnam Univ.
E-mail : soo026@hanmail.net

요 약

KVM에서의 검증단계는 크게 두가지로 나뉜다. 일반적으로 처리되는 Off-device에서의 사전검증과 실제 VM에 탑재되어 수행되는 in-device에서의 검증이다. 전자의 경우, 실행시 검증을 쉽게 하기 위해서 클래스 파일에 특별한 속성을 가진 스택맵을 추가한다. 이렇게 추가된 스택맵은 KVM상에서 보다 효율적인 검증을 위한 것으로써, 본 논문에서는 이러한 스택맵을 분석하였다.

1. 서론

오늘날의 컴퓨팅 시장은 데스크탑에서의 인터넷 시대를 지나 언제, 어디서든지 네트워크에 연결될 수 있는 휴대용 디지털 기기의 인터넷에 대한 관심이 증대되고 있다. 이를 반영하듯 이동성과 휴대성을 목적으로 주목 받고 있는 디지털기기 가운데 하나가 셀룰러폰, PDA 등과 같은 이동 단말기들이며, 기존의 컴퓨팅 시장에서 큰 변혁을 초래하고 있다. 이러한 변혁을 주도할 기술로, 자바가 대표적이다. 자바는 동적인 응용프로그램을 다운로드, 크로스 플랫폼 호환성, 향상된 사용자 경험과 역동성, 비연결성, 보안 문제등과 같은 이유로 각광받고 있다.

하지만 데스크탑 플랫폼에서 수행되던 자바 기술을 셀룰러폰이나 PDA 와 같은 휴대용 디지털 기기에 그대로 적용할 수는 없다. 왜냐하면, 기존의 컴퓨팅 환경에서 사용되어 오던 자바는 많은 자원을 필요로 하는 반면, 셀룰러폰이나 PDA 와 같은 제한적인 자원을 가지고 있는 기기에서는 자바의 모든 기술을 받아들일 수가 없기 때문이다. 그리하여 제한적인 자원을 가진 이동 디지털 기기에서도 운용될 수 있는 기술을 요구하게 되었고, 썬마이크로시스템즈사는 KVM이라는 제품을 내놓게 되었다. KVM은 'Kilobyte Virtual Machine'의 약어로써, 적은 메모리 용량과 CPU처리능력, 그리고 느린 무선 네트워크에 맞게 설계된 가상 기계이다. 가상 기계의 크기는 45KB~70KB 정도이며, 가상 기계와 힙(Heap) 메모리가 구동되어 실행되기 위해서는 128KB의 메모리 용량을 필요로 한다. 그리고 KVM 검증의 경우 약

† 본 연구는 한국과학재단 산학협력연구(2001-30300-002-1) 지원으로 수행되었음.

10KB 의 이진 코드와 100bytes 의 동적 RAM 만을 요구한다. 이 검증기의 경우 복잡한 반복형태인 데이터 흐름의 알고리즘이 아닌 바이트 코드의 선형적인 스캔만을 수행한다.

본 논문에서는, 자바 가상 기계에서 실시했던 검증의 단점을 보완하고 소형단말기에 적합한 검증알고리즘인 KVM의 검증 알고리즘에 관해 연구하고, 검증과정중 사전검증에서 생성되는 스택맵을 분석하여 KVM의 검증을 보다 효율적으로 한다.

2. 관련연구

2.1 자바가상 머신의 검증

자바 가상 기계(Java Virtual Machine)의 검증 알고리즘은 시스템으로 로드되는 모든 클래스에 인스턴스가 생성되거나 정적 프로퍼티(static property)들이 사용되기 전에 적용된다. 이를 통해 자바 가상 기계는 클래스가 안전성(safety)과 관련된 특성을 갖고 있다고 가정하며, 이러한 가정을 바탕으로 최적화를 수행한다.

검증 알고리즘은 클래스 파일에 관한 몇 가지 질문들을 통해서 동작한다. 질문은 다음과 같이 5개의 일반적인 카테고리로 분류된다.

① 구조적인 관점에서 명확한 class 파일인가?

검증 알고리즘은 매직수(Magic number)인 처음의 4바이트 16진수 값(예, CA, FE, BA, BE)을 통해 잘못된 파일 혹은 클래스 파일이라고 생각할수 없는 파일을 즉시 거부할수 있다.

② 모든 상수들이 정확하게 참조되는가?

파일이 적절한 형식을 갖춘 클래스 파일인지 검사한 후에 검증 알고리즘은 컨스턴트풀(Constant Pool)의 위치와 그 안에 존재하는 상수의 개수를 알게 된다. 또, 모든 상수의 태그가 유효함도 확인한다.

③ 명령어들이 모두 유효한가?

메소드의 몸통 부분을 들여다보고, 메소드 안의 명령어들이 정확한 형태를 갖추었는지 알아본다.

④ 스택과 지역 변수가 항상 적절한 타입의 값을 가지는가?

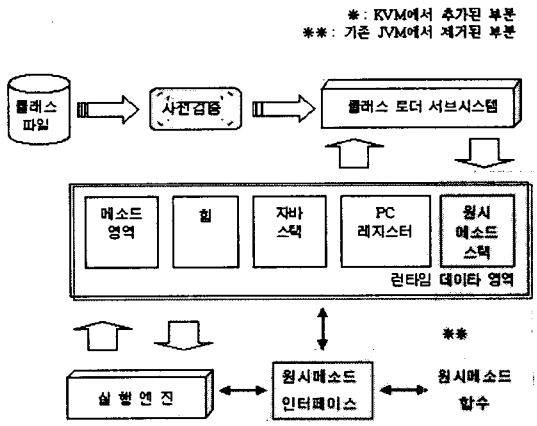
검증 알고리즘은 잘못된 처리를 할지도 모르는 모든 클래스들과 소수의 안전한 클래스까지도 거부한다는 전제를 세워야 한다. 그러므로 검증 알고리즘은 적절한 원소들이 항상 스택의 맨 위에 존재하는지, 특정한 위치에서 실행되는 명령어는 항상 같은 크기의 스택을 사용하게 되는지를 체크하게 된다.

⑤ 외부로의 참조는 검사되었는가?

외부로의 참조란, 현재 검사하는 클래스 외의 다른 클래스의 필드나 메소드를 사용하려는 경우를 말한다. 클래스가 같은 자바 소스 파일에서 정의되었다 하더라도, 이들은 모두 다른 클래스 파일에 존재하게 되므로 여전히 외부로의 참조이다. 외부로의 참조를 검사하는 것은 컨스턴트풀(Constant Pool)에서 참조되는 모든 클래스들을 로딩하는 문제이다.

2.2 KVM(kilobyte Virtual Macine)

K-자바 가상 기계의 구조는 [그림 1]과 같이 클래스 로더 시스템(Class Loader System), 실행 엔진(Execution Engine), 런타임 데이터 영역(RunTime Data Areas)으로 구성되어 있다. 기본적으로 기존의 자바 가상 기계와 유사하다.



[그림 1] KVM의 구조

하지만 K-자바 가상 기계는 클래스 파일을 클래스 로더 서브 시스템에 로드하기 전에 사전검증을 거쳐 미리 검증한다. 그리고 K-자바 가상 기계의 런타임 데이터영역에는 원시 메소드 함수를 지원하기 위한 원시 메소드 스택을 가지고 있지 않다.

① 적재

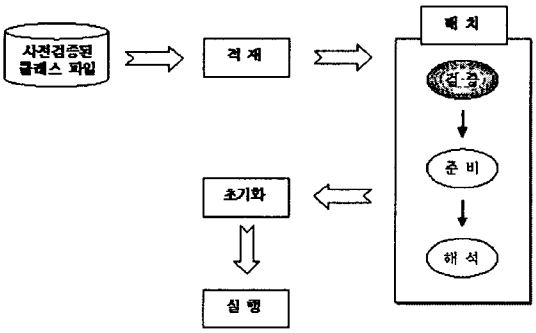
적재(Loading)란 애플리케이션에서 사용하려는 클래스파일을 가상 기계로 전달하는 과정을 말한다.

② 배치

배치(Linking)란 로딩된 클래스파일, 즉 바이너리 코드가 가상 기계에 의해 수행될 수 있는 상태로 만드는 과정이다. 이 과정은 적재된 클래스가 올바른 클래스포맷을 갖고 있는지 실행 단계에서 검증을 하는 검증과정과 메모리 영역의 할당 등의 예비과정(preparation), 심볼릭 참조 주소를 직접 참조 주소로 변환하는 결정과정(resolution)등으로 세분화할 수 있다. 이 때 보안상의 이유와 애플리케이션 관리 소프트웨어의 존재로 인해 클래스 파일의 룩업(Look-up) 순서가 조금 다르며, 사용자 정의 클래스 로더를 만들 수 없다.

③ 검증

가상기계에서의 검증(Verification)은 클래스 파일을 적재(Load)하기 위한 클래스 파일 로더 시스템에서 실행되며 그 실행과정은 [그림 2]과 같다.

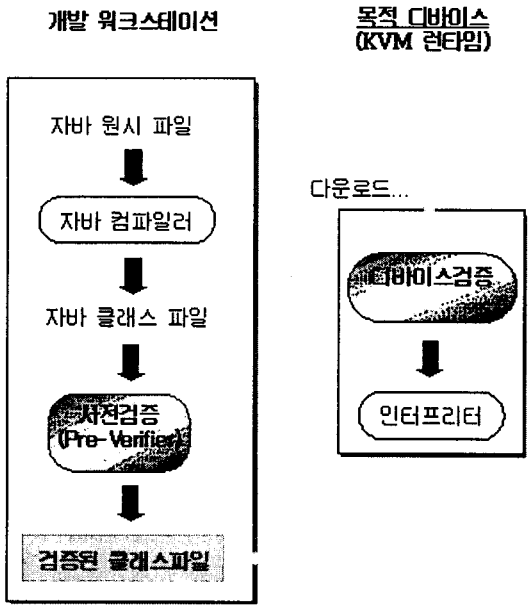


[그림 2] KVM 실행과정

검증은 적재된 데이터의 형이 자바 언어의 의미(semantics)를 따르고 있는지 혹은 가상 기계의 내부 형식을 만족하는지를 검사하는 단계이다. 자바 가상 기계와 K-자바 가상 기계는 기본적으로 동일한 실행 과정을 거치지만 실행방법에서는 다소 차이를 보인다.

K-자바 가상 기계(K-Java Virtual Machine)는 [그림 3]에서와 같이 데스크탑에서 한번의 검증 단계를 수행하는 자바 가상 기계과는 달리 두 번의 검증 단계를 수행한다.

K-자바 가상 기계는 사전검증에서 스택맵(StackMap)이라는 검증정보를 클래스 파일에 추가하여 사용자에서 검증단계에서 검증정보를 이용하여 자바 가상 기계에서의 검증보다 빠르게 수행한다.



[그림 3] 프로그램 실행순서와 KVM의 검증단계

a) 사전검증

K-자바 가상 기계에서는 자바 가상 기계와 동일한 검증을 하지 않는다. 왜냐하면 J2SE(Java 2 Standard Edition)에서 정의된 클래스파일 검증은 소형 디바이스에서의 많은 메모리 크기를 요구하기 때문이다. 그래서 클래스파일을 실행하기 전에 미리 검증 단계를 거쳐서, 올바른 클래스 파일인지 검사하고, 스택맵이

라는 검증정보를 포함하도록 한다.

b) 디바이스 검증

가상 기계의 클래스 파일을 로드하고 난 다음, 검증시 전반적인 클래스 파일을 검사하지 않고 스택맵이라는 검증 정보를 이용하여 검증한다.

3. 스택맵 분석

3.1 스택맵 속성의 형식

사전 검증 과정에서 클래스 파일에 특별한 속성을 추가하는 수정작업이 일어난다. 이 속성값은 지역변수와 오퍼랜드 스택에 대한 부분을 서술하기 때문에 스택맵이라 불린다.

스택맵 속성의 형식은 다음과 같다.

```
StackMap_attribute{
    u2 attibute_name_index;
    u4 attribute_length;
    u2 number_of_entries;

    {
        u2 byte_code_offset ;
        u2 number_of_locals;
        ty types_of_locals[number_of_locals];
        u2 number_of_stack_items;
        ty types_of_stack_items[number_of_stack_items];
    } entries [number_of_entries]
}
```

[그림 4] 스택맵 속성의 형식

3.2 스택맵(StackMap) 분석

본 논문에서는 볼 클래스(Ball Class)라는 예제 프로그램을 이용해서 스택맵을 분석 하였다. 볼 클래스는 크기 조절이 가능한 동그라미를 이동시킬 수 있는 클래스로써 총 4개의 메소드로 구성되어 있으며 그로 인해 4가지 스택맵이 존재하게 된다. 그 중 리사이즈(Resize)메소드의 스택맵을 살펴본다.

리사이즈 메소드는 주어진 사이즈로 볼의 크기를 조절한다.

다음 [그림 5]는 리사이즈 메소드의 소스 코드이다.

```
public void resize(int diff) {
        int prev = radius;
        radius += diff;
        if (radius < 2) radius = 2;
        else if (radius > 20) radius = 20;
        else {
                // size actually changed -> redraw
                int ballSize = prev*2;
                g.drawRectangle(posX,posY,ballSize,
                        ballSize,g.ERASE,0);
                ballSize = radius*2;
                g.drawRectangle(posX,posY,ballSize,
                        ballSize,g.PLAIN,radius);
        }
}
```

[그림 5] Resize() 메소드 소스 코드

리사이즈 메소드는 사전검증 단계를 거쳐 다음과 같은 스택맵을 생성하게 된다.

```
0066 00000023 0003 001f 0003 07 001b 01 01 0000
0031 0003 07 001b 01 01 0000 0065 0003 07 001b
01 01 0000
```

[그림 6] 리사이즈 메소드에 대한 스택맵

스택맵의 구조에 따라, 처음 2바이트 '0066'은 컨스탄트 풀 테이블에 정의되어있는 유효한 인덱스 값이다. "StackMap" 이라는 스트링을 포함한 CONSTANT_UTF8-info 구조여야하고, 다음 4바이트 '00000023'은 네임 인덱스 2바이트와 자신이 4바이트를 제외한 스택맵의 길이를 나타낸다. 그 다음 2바이트 '0003'은 엔트리 배열에서 엔트리의 개수를 나타낸다. 이 엔트리의 개수에 따라 스택맵의 길이가 달라진다. 그 다음부터는 각각의 엔트를 나타낸다. 리

사이즈 메소드에는 세개의 엔트리가 존재한다.

다음 [표 1]은 그 중 첫번째 엔트리를 나타낸것이다.

[표 1] 엔트리 분석

byte	contents	description
001f	byte_code_offset	31번지
0003	number_of_locals	3개
07	ty	ITEM_IninObject
001b	type_name_index	Ball 클래스
01	ty	32 비트 정수형
01	ty	32 비트 정수형
0000	number_of_stack_items	0 개

처음 2바이트 '001f' 는 현재 엔트리에 해당되는 바이트 코드 명령어의 오프셋을 나타내고, 다음 2바이트 '0003'은 타입(ty)이 기록되어 있는 로컬 변수의 개수를 나타낸다.

다음으로 나오는 1바이트는 로컬변수의 타입으로써, type_of_locals[n] 은 로컬 변수 n (0 이상 로컬변수 개수 미만) 의 타입을 나타낸다. 타입은(ty) 1바이트 혹은 3바이트이다. 그리고 아래와 같다.

[표 2] 로컬 변수 타입 정의

Name	Code	Explanation
ITEM_Bogus	0	초기화되지 않았거나 알려지지 않은 값
ITEM_Integer	1	32 비트 정수형
ITEM_Float	2	CDLC 검증기에서는 사용 되지 않음
ITEM_Double	3	CDLC 검증기에서는 사용 되지 않음
ITEM_Long	4	64 비트 정수형
ITEM_Null	5	Null 타입
ITEM_InitObject	6	상세한 설명은 밑에
ITEM_Object	7	상세한 설명은 밑에
ITEM_NewObject	8	상세한 설명은 밑에

① ITEM_InitObject

<init> 메소드에 대한 생성자가 호출 되기 이전에 ITEM_InitObject 타입을 가져야 한다. 현재 객체에서 다른 연산을 수행하기 이전에 <init> 메소드에 대한 생성자가 먼저 호출되어야 한다.

② ITEM_Object

클래스 인스턴스. 처음 1바이트는 타입코드이고 (이 경우 07 해당), 이어 오는 2바이트는 type_name_index 이다. 컨스턴트 풀에서 정의된 CONSTANT_Class_info 의 구조를 띤다.

③ ITEM_NewObject

초기화 되지 않은 클래스 인스턴스. New 명령어에 의해서 새로 생성된 클래스 인스턴스이다. 하지만 <init> 메소드가 호출되지 않아, 아직 초기화는 되어 있지 않은 상태이다. 처음 1 바이트는 타입코드이고 (이 경우 08 해당) , 이어 오는 2바이트는 new_instruction_index 이다. 이 값은 반드시 바이트 코드의 오프셋 값이어야 한다.

여기서 32비트 정수형 두개는 매개변수 int diff 와 지역변수 int prev 를 나타낸다. 그리고 마지막 2바이트 '0000'은 스택에서 아이템의 개수를 나타낸다.

나머지 두개의 엔트리 또한 위의 첫번째 엔트리와 같은 구조를 가진다.

4. 결론

본 논문에서는 KVM의 두 단계검증 과정 중 오프디바이스 상에서의 사전검증에서 생성되는 스택맵을 분석하였다. 이 분석의 결과로 스택맵을 구성하는 엔트리의 값들을 알수 있었다. 이를 바탕으로 향후에는 KVM을 사용하는 응용프로그램 제작시에 도움이 될, KVM용 디버깅툴과 KVM 스택맵 분석기 제작에 관련된 연구와 스택맵 생성시에 보다 효율적인 메모리 사용에 관련된 연구가 필요하다.

[참고문헌]

[1] Joshua Engel 저 / 곽용재 역 '자바가상머신 프로그래밍', '인포북'

[2] 정준영, "자바 플랫폼 지니 기반 소형 자바 가상 기계의 설계 및 구현", 춘계멀티미디어학회 논문, (2001)

[3] 옥재호, "비주얼 자바 클래스 파일 브라우저의 설계 및 구현", 경남대학교 대학원 석사논문, (1998)

[4] http://java.sun.com/, Sun Microsystems, Java Home Page

[5] Sun Microsystems, Inc. The Java™ Virtual Machine Specification, SUN 2002-04-29

[6] Bill Venners, "Inside the Java Virtual Machine second edtion"

[7] http://www.mobilejava.co.kr/