SMT-Solver 를 사용한 자바바이트코드의 배열 오버플로우 검증

이상협*, 김제민**, 박준석*, 유원희*
*인하대학교 컴퓨터 정보공학부
**인하대학교 일반대학원 컴퓨터 정보공학과
e-mail : mpmasllo@gmail.com

A Verification of Array Overflow in Java Bytecode using SMT-Solver

Sanghyup Lee*, Jemin Kim**, Joonseok Park*, Weonhee Yoo*
*Dept. of Computer Science & Information Technology, Inha University
** Dept. of Computer & Information Engineering, Inha University

요 약

자바프로그램 검증은 안전하고 정확한 프로그램을 만들기 위한 필수적인 조건이지만 자바언어로 작성된 프로그램은 바이트코드로 작성되어 있는 클래스 파일로 배포되기 때문에 바이트코드에 대한 검증이 필요하다. 하지만 자바 바이트코드는 가독성이 떨어져 중간언어로 변환을 하고 그 중간코드에서 검증에 필요한 조건들을 작성 해야 한다. 이 논문에서는 새로 정의된 중간언어인 BIRS을 통해 컴파일시 검증이 되지 않는 배열 오버플로우에 대한 정적검증을 설명하고 검증 절차에 필요한 명제의 정의와 검증 시 사용되는 SMT-Solver 인 Z3 의 사용법에 대하여 서술하였다.

1. 서론

자바 언어는 오퍼레이팅 시스템에 독립적으로 사용된다. 그 원리는 .java 파일을 자바언어에서 .class 파일로 컴파일 해 자바 가상 머신 (JVM. Java Virtual Machine)을 이용해 프로그램이 실행이 되기 때문이다. 이는 사용자 입장이나 개발자 입장에서는 어느 오퍼레이팅 시스템을 사용해도 동일한 작업을 할 수 있고 같은 결과를 얻을 수 있는 장점이 있다. 이와 같은 장점으로써 바이트코드가 배포 및 실행되고 있다. 하지만 프로그램의 검증을 하는 입장으로서 개발자가 원 소스프로그램을 공개하지 않는 이상 우리는 바이트코드라는 기계언어를 참조하여 검증할 수밖에 없다. 따라서 자바바이트코드를 검증하는 것이 필요하다.

프로그램을 작성 및 실행 할 경우에는 검증을 하지 않으면 오류가 발생할 수 있다. 여러 오류 중 배열 오버플로우에 대한 검증 또한 필요하다. 특히 우리는 배열 오버플로우 검증시 정적 검증이 필요한데 그 이유로는 자바 가상 머신 검증기 (JVMV, Java Virtual Machine Verifier) 가 여러 가지 오류 가능성을 가진 코드를 검증 하지만 배열 오버플로우에 대한 검증에 대한 부분이 부족하기 때문에 경제적, 시간적 비효율성이 생긴다. 대부분의 자바언어로 작성된 프로그램은 바이트코드 상태로 배포되고 있으며 컴파일시 배열 오버플로우에 대한 검증이 비효율적이기 때문에 바이트 코드를 정적 검증을 해야 한다 [1].

검증 시 사용할 SMT-Solver(Satisfiability Modulo Theories Solver)는 이미 개발이 되어있고 신뢰성 있는 마이크로소프트사에서 개발한 Z3 SMT-Solver[2]를 사용할 것이다. SMT-Solver 를 사용하는 이유는 직접 새로운 검증 프로그램을 만들어 사용하는 것 보다 시간의 절약과 신뢰성을 얻을 수 있기에 사용한다.

2. 관련연구

2.1 Z3

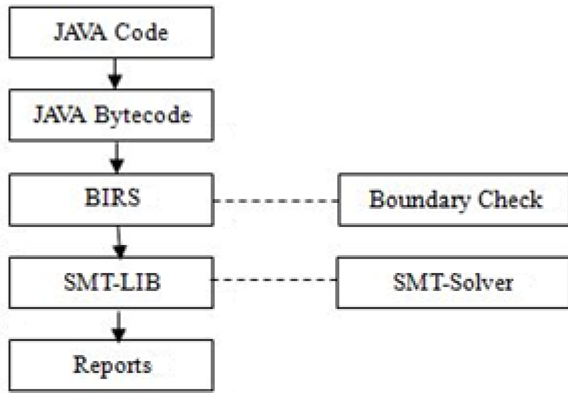
Z3 는 마이크로소프트에서 개발하고 있는 SMT-Solver 이다. 제공하는 기능으로는 Propositional Solving, Relations, Functions, Constants, Arithmetic, Data-type, Bit-vectors, Arrays, Quantifiers, Simplification, Implied Equalities, Unsatisfiable Cores 등의 모듈을 제공하고 있다. 이 논문에서는 인덱스가 배열의 크기를 초과하는지에 대한 정보가 필요하므로 SMT-Solver 의 모듈 중 Arithmetic 모듈을 사용한다.

3. 본론

3.1 배열 오버플로우 검증 과정

자바 코드로 작성된 프로그램을 컴파일 한 자바 바이트 코드를 BIRS 로 변환한다. 그 후 변환된 코드에 배열범위를 검증할 수 있는 술어(predicate)를 정의하고 그 검증식에 BIRS 의 파라매터를 받아서

그 결과를 SMT-Solver 의 Arithmetic 모듈에 입력한 후 결과를 도출한다.



(그림 1) 배열 오버플로우의 검증과정

3.2 BIRS

Bytecode Intermediate Representation With Specification (BIRS)는 스택기반인 자바언어의 단점을 보완하기 위해서 본 연구과제에서 제안한 중간언어이다. 이 언어는 객체지향언어가 가져야 하는 구조물(메소드, 클래스 구조)과 모듈정보를 포함하고 있으며, 명령어 표현은 스택리스 코드로 표현되어 있다.

3.3 소스코드

(그림 2)는 본 논문에서 사용된 소스 코드로서 for 문을 통한 배열에 자동 접근을 하도록 작성하였다. Class 의 이름은 ArrayForExampleOver 이며 배열은 arrayExample 로서 배열의 원소가 정수타입을 가지고 크기는 10 으로 인덱스의 범위는 0≤i≤9 를 가지게 된다. for 문은 인덱스를 정수형 i 로 설정하고 for 문의 지속조건은 i 가 11 미만이 될 때로 작성하였다. 이 코드는 탈출조건이 배열의 범위보다 크기 때문에 인덱스 변수 i 는 배열의 인덱스범위를 넘어선 값인 10 을 가질 수 있다. 그 결과로 오버플로우가 발생한다. 하지만 이 소스를 컴파일을 할 시에는 이러한 오류를 검출 하지 못하고 컴파일이 실행이 된다. 이는 배열 오버플로우의 정적 검증의 필요성을 보여준다.

```
class ArrayForExampleOver
{
        public static void main(String[] args)
        {
                int[] arrayExample = new int[10];

                for(int i=0;i<11;i++)
                {
                        arrayExample[i] = i+1;
                }
        }
}
```

(그림 2) 예제프로그램

3.4 바이트 코드

(그림 3)의 코드는 이 논문에 사용된 (그림 2)의 바이트 코드이다. 바이트코드는 스택기반의 기계언어이기 때문에 가독성이 떨어지고 분석 및 검증에 용이하지 않으며 필요한 정보를 취합하기에 불편하다.

```
Compiled from "ArrayForExampleOver.java"
class ArrayForExampleOver extends java.lang.Object{
ArrayForExampleOver();
  Code:
   0:   aload_0
   1:   invokespecial   #1;   //Method
java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   bipush   10
   2:   newarray int
   4:   astore_1
   5:   iconst_0
   6:   istore_2
   7:   iload_2
   8:   bipush   11
   10:  if_icmpge 25
   13:  aload_1
   14:  iload_2
   15:  iload_2
   16:  iconst_1
   17:  iadd
   18:  iastore
   19:  iinc     2, 1
   22:  goto     7
   25:  return

}
```

(그림 3) 자바 바이트코드 형태의 예제프로그램

3.5 BIRS 로 변환

BIRS 는 바이트코드를 검증하기 위한 본 연구과제에서 정의를 내린 중간언어이다. 스택기반의 자바코드를 스택리스코드로 변경하여 가독성이 향상되고 검증에 필요한 정보를 취합하기에 자바 바이트코드보다 용이 하다. (그림 4)는 (그림 3)의 코드를 BIRS 코드로 변환한 것이다. block0 부터 block9 까지는 기본블록의 내용을 표현하고 그 밑의 행은 이 프로그램의 제어흐름을 보여준다. 컴파일 과정 중 지역 변수의 이름에 대한 정보는 사라지지만 설명의 편의를 위하여 배열의 이름은 arrayExample 로 임의로 정하였다.

```
logicalfunction main:(string array)→unit
computationfunction main:(string array args)→unit{
    read(i) write(i,arrayExample)

block 0 :   0 : check
            1 : newarray arrayExample := newarray int
array[10]
            2: assignval i :=0

block3:     3:ifd i<11 4
block4:     4:check
            5:check
            6:modArray   arrayExample[i]:=i+1
            7:assignVal i:=i+1
            8: goto 3
block9:     9:return
from 0 to 3
from 3 to 4 when i<11
from 3 to 9 when ¬(i<11)
from 4 to 3
returnblock 9}
```

(그림 4)예제프로그램의 중간표현

3.6 배열 범위검증

(그림 4)의 중간코드를 기반으로 배열의 검증을 위해서는 배열 범위의 검증식의 정의가 필요하다. 위의 중간코드에서 필요한 정보인 배열의 크기는 block0 의 1 행에 있는 1: newarray arrayExample:= newarray int array[10]에서 얻을 수 있다. 그리고 필요한 다른 정보로는 block4 의 6 행의 코드인 6:modArray arrayExample [i]:=i+1 의 local1 의 인덱스 부분인 i 의 값이 필요하다. 인덱스변수 i 의 크기가 배열의 인덱스 범위보다 큰 10 을 가지므로 배열의 접근의 오류가 발생한다.
이를 검증하기 위해서는 (그림 4)의 read(i) write (i,arrayExample)다음에 술어를 정의해 주어야 한다. (그림 4)의 코드에 (그림 5)을 삽입함으로써 술어가 정의된다.

```
predicate checkArrayBound (Arrayname, Index, Length) :=
{ 0≤Index<Length}
```

(그림 5)검증을 위한 술어

조건문에 포함된 Arrayname 은 이번 예제에는 배열이 하나만 정의가 되었지만 배열이 하나 이상으로 정의되었을 때 각각의 배열을 구분해주기 위해서 필요하다. Index 는 block4 의 6 행의 i 값으로서 배열에 접근을 하는 인덱스로서 이 크기는 배열의 크기보다 작아야 오버플로우가 발생되지 않는다. Length 는 배열의 크기로서 block0 의 0 번째 행의 int array[10]에서 10 에 해당하는 정보를 가지고 있다.
위의 정의된 술어를 기반으로 검증식을 실행하기 위한 코드로서 (그림 6)을 (그림 4)의 block4 의 5 행 뒤에 삽입을 한다.

```
assert checkArrayBound(arrayExample,i,10)
```

(그림 6)검증식

(그림 7)은 (그림 4)의 검증을 위한 술어와 검증식이 삽입된 코드이다.

```
logicalfunction main:(string array)→unit
computationfunction main:(string array args)→unit{
    read(i) write(i,arrayExample)

predicate checkArrayBound(Arrayname,Index,Length):={ 0
≤Index<Length}


block 0 :   0 : check
            1 : newarray arrayExample := newarray int
array[10]
            2: assignval   i := 0

block3:     3:ifd i<11 4
block4:     4:check
            5:check

        assert checkArrayBound(loca11,i,10)

            6:modArray   arrayExample[i]:=i+1
            7:assignVal i:=i+1
            8: goto 3
block9:     9:return
from 0 to 3
from 3 to 4 when i<11
from 3 to 9 when ¬(i<11)
from 4 to 3
returnblock 9}
```

(그림 7) 술어와 검증식이 삽입된 중간표현

3.7 SMT-SOLVER 에 의한 검증

(그림 5)의 검증 식을 정의 내린 후 검증식이 참인진 거짓인지 판별하기 위해서는 먼저 검증조건(VC, Verification Condition)을 생성해야 한다. 위의 코드에서 생성된 VC 는 (그림 8)과 같다.

$$\forall i\ 0 \leq i < 11 \rightarrow \text{checkArrayBound}(local1, i, 10)$$

(그림 8) 검증을 위한 검증조건

VC 를 생성한 후 프로그램 검증을 위해서 SMT-Solver 를 사용해야 하는데 이 논문에서는 Z3 를 사용한다. (그림 8)의 검증조건을 통해 SMT-Solver 사용하기 위해서는 입력 포맷인 SMT-LIB[3]포맷으로 바꾸어야 한다.

```
(set-option set-param "ELIM_QUANTIFIERS" "true")
(simplify
 (forall (i Int)
  (implies (and (<= 0 i) (< i 11))
   (and (<= 0 i) (< i 10)))))
```

(그림 9) VC 을 참조해 SMT-LIB 포맷으로 변환

(그림 9)의 코드는 변환된 SMT-LIB 포맷으로서 첫 행에서 Z3 의 Quantifier Elimination 기능을 사용하기 위해 옵션을 준다. 2 행에서는 인덱스인 모든 i 가 정수형임을 설정을 하고 3 행과 4 행을 통해서 (그림 8)의 나머지 부분을 SMT-LIB 포맷으로 변환하였다. (그림 9)를 Z3 을 통해 검증을 하면 “false”라는 결과값이 나오므로 (그림 2)의 코드는 배열 오버플로우가 발생함을 알 수가 있다.

4. 결론 및 향후 연구 과제

이 논문은 자바바이트코드 상태의 배열 오버플로우의 정적 검증이 가능하게 만든다. 바이트코드를 BIRS 라는 새로 정의된 중간어로 변환한 후 검증에 필요한 조건을 정의된 후 그 내용을 SMT-Solver 에 입력하여 그 조건이 만족하는지 안 하는지의 대한 검증 결과를 얻을 수 있다.

이 논문의 배열 오버플로우 검증의 제한점은 인덱스는 affine 형식으로서만 검증이 가능하다는 것이다. 그러므로 향후 연구과제로는 인덱스 형태의 다양성에 대한 연구가 필요하다. 예로 들어 배열형태나 포인터로서의 인덱스에 대한 연구가 필요하다.

논문 사사(Acknowledgement)

이 논문은 2010 년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No. 2010-0025660)

참고문헌

[1] Hongwei Xi and Songtao Xia. 1999. Towards array bound check elimination in Java TM virtual machine language. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (CASCON '99), Stephen A. MacKay and J. Howard Johnson (Eds.). IBM Press 14.

[2] Z3-An Efficient Theorem Prover. Microsoft Research. http://research.microsoft.com/en-us/ um/redmond /projects/z3/index.html

[3] SMT-LIB The Satisfiability Modulo Theories Library. http://www.smtlib.org/

[4] Cormac Flanagan and James B. Saxe. 2001. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '01). ACM, New York, NY, USA, 193-205. DOI=10.1145/360204.360220 http://doi.acm.org/10.1145/360204.360220

[5] Mike Barnett and K. Rustan M. Leino. 2005. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (PASTE '05), Michael Ernst and Thomas Jensen (Eds.). ACM, New York, NY, USA, 82-87. DOI=10.1145/1108792.1108813 http://doi.acm.org/10.1145/1108792.1108813