Project 1: Alloy model of JVM array load instructions

Qinxun Bai and Zuojun Zhu

Introduction:
In this project we build an Alloy model to analyse a binary representation of a simple Java method. This Java method is about array load instructions. The model can check the following constrains when an array load instruction is encountered: the stack depth is at least 2 and the stack must contain an array ref and an integer index. For simpleness, the array is 1-dimension and with a statically assigned length.

Description:
We divide our project into the following modules:
1. Simple Java array load methods
2. Bytecodes of the java methods
3. Build Alloy model based on Mark’s previous work
4. Initialization of the Alloy model by analyzing the bytecodes
5. Modify the initializer of the Alloy model to produce the malicious code
6. Test Alloy model with malicious code

Detailed description of each module:

1. Java methods and bytecodes
In order to handle with different type of methods, we write two different Java methods as below:

   public static int iaload(int int1,int[] array)
   {
       array[0] = int1;
   }

   public static double daload(double double1,double[] array)
   {
       array[0] = double1;
   }

The function of these two methods is very clear: push one element of Int type (or Double type) into an array of Int type (or Double type). With this simple method we want to analyse whether the method conform to the constrains we will build in the Alloy model. To build the Alloy model, we must understand how the Java method works in the stack clearly. Only with the Java code we can not get the detail information about it. So we turn the .class file into bytecode with the "javap" tool. The bytecode is made up of opcodes and with the help of the Java documentation published online we can look up each opcode and understand what it actually
does in the stack.

We list the opcodes and explain the meaning of each opcode as below:

```java
public static void iaload(int, int[]);
Code:
0: aload_1
1: icontrol_0
2: iload_0
3: iastore
4: return
```

- `aload_1` means that loads a reference onto the stack from local variable 1
- `icontrol_0` means that loads the int value 0 onto the stack
- `iload_0` means that loads an int `value` from local variable 0
- `iastore` means that stores an int into an array
- `return` means that return void from method

```java
public static void daload(double, double[]);
Code:
0: aload_2
1: icontrol_0
2: dload_0
3: dastore
4: return
```

- `aload_2` means that loads a reference onto the stack from local variable 2
- `icontrol_0` means that loads the int value 0 onto the stack
- `dload_0` means that loads an double `value` from local variable 0
- `dastore` means that stores an double into an array
- `return` means that return void from method

2. Alloy model

Our Alloy model should be able to check the following constraints when an array load instruction is encountered:

- The stack depth is at least 2
- The stack must contain an array ref and an integer index

In order to realize this, at first, we should be able to specify the moment an array load instruction is encountered. Our solution is adding a “flag” relation in the instruction signature, which during model initialization, is assigned to 1 only for ‘load’ instructions (i.e. iload and dload in our function) and 0 for other instructions.

Also, in order to check whether the stack contains an array reference and an integer index, we need to know the type of each object stored in the stack (at least for the two objects pushed
into the stack right before the array load instruction), our solution is maintaining a sequence of
types as a relation of the state signature, which represent the types of objects currently stored in
the stack. In order to construct this sequence of types and update it during state sequencing, we
add a “type” relation to the Instruction signature (i.e. to adhere a type attribute to each
instruction) and update the sequence of types accordingly if an instruction changes the length of
the stack. In account for clarity when adding the above types, we create a new abstract signature
“Type”, and instantiate IntType, DoubleType and RefType from it.

So far, we have already added the following codes as described above to Mark's original
Alloy model:

```alloy
//Add a Type signature
abstract sig Type{}
one sig IntType, DoubleType, RefType extends Type{}

//Add two relations to Instruction signature
abstract sig Instruction { ......
    type: lone Type,
    flag: Int
}

//Add a relation to State signature
sig State { ......
    stack: seq Type
}
```

During the state sequencing process, we need to add the updating of Type sequence: stack,
the basic idea is to add an according Type element to the Type sequence if current instruction
pushes something into the stack, or to delete the last element of the stack sequence if current
instruction pops something out of the stack. The Type sequence is unchanged if the stack is not
changed by current instruction. A predicate for doing this is as follows:

```alloy
//Add update stack sequence
pred nextStack[currentInstruction: Instruction, currentStack, nextState: seq Type]{
  (currentInstruction.mod>0) => (nextStack = currentStack.add[currentInstruction.type])
  else
  (currentInstruction.mod<0) => (nextState= currentStack.delete[#currentStack])
  else
  (currentInstruction.mod=0)=>(nextState= currentStack)
}
```

This predicate is invoked by nextState predicate, and in sameState predicate, equivalence of
the stack relation should be added for s and s'.

Finally, we add the following assertion to check the two constraints, note that precondition
for executing an actual checking is that an array load instruction is encountered, so our ‘flag’
relation is playing its role here, and we only need to check the two previous elements of the type sequence.

// Assertion for constraints checking
assert checkit3{
    all s: State | (s.prog.flag=1) => (gte[#s.stack],2)\&\&
    s.stack[[#s.stack].sub[1]]\&\&
    s.stack[[#s.stack].sub[2]]=RefType
}

3. Extend the Alloy model for type checking

We want to extend the above model such that it can checks the consistence of the array type against the load instruction being executed. Our solution is extending an “NullType” from the Type signature and adding an “atype” relation to the Instruction signature and an “aref” relation to the State Instruction, as follows:

//Extend an NullType
abstract sig Type{
    one sig IntType, DoubleType, RefType, NullType extends Type{

    //Add one more relation to Instruction signature
    abstract sig Instruction {    ......
        atype: lone Type,
    }

    //Add one more relation to State signature
    sig State {    ......
        aref: Type
    }
}

where “atype” indicates array reference type. During initialization, for instructions like “aload_1” and “aload_2”, “atype” will be assigned accordingly to IntType and DoubleType respectively, while for other instructions, to NullType.

“aref” is used to specify the latest encountered array reference type. So when an array load instruction is encountered, we just compare “aref” with the current instruction type. Updating of “aref” is as follows, “aref” is actually updated only when an array reference type is encountered:

//Add update aref
pred updateAref[currentInstruction: Instruction, currentAref, nextAref: Type]{
    (currentInstruction.atype=NullType)\=>(nextAref=currentInstruction.atype)
    else
        (nextAref = currentAref)
This predicate is invoked by nextState predicate, and in sameState predicate, equivalence of the “aref” relation should be added for s and s’.

Finally, we extend the previous assertion for constraints checking, precondition is still the same: when an array load instruction is encountered, we now add one constraint that the load instruction type should be the same as the latest encountered array reference type, which we maintain in “aref”.

// Extended assertion for constraints checking
assert checkit3{
    all s: State | (s.prog.flag=1) => (gte([s.stack],2) &&
                          s.stack[([s.stack].sub[1])] = IntType &&
                          s.stack[([s.stack].sub[2])] = RefType &&
                          s.aref = s.prog.type)
}

4. Initialization of the Alloy model
After building the Alloy model, we need to make the Initialization of the Alloy model according to the opcodes we have. As described above, we add three relations in the Instruction signature: "flag", "atype" and "type".

"flag" is used to help us find the array load instruction. We initialize the value of "flag" for "iload_0" and "dload_0" as one and for other opcodes as zero. Take the “iaload” function as an example, initialization of our added flag relation is as follows:

fact flags{
    flag = startup->0
    + aload_1_1->0
    + iconst_0_2->0
    + iload_0_3->1
    + iastore_4->0
    + return_5->0
}

"type" and "atype" are defined in the field “Type” (including "IntType", "DoubleType", "RefType" and "NullType"). "atype" is used to indicate array reference type and "type" is used to indicate the object type the opcode operates on. As described in the previous Alloy model part, for “iaload” function, they are initialized as follows:

fact types{
    type = aload_1_1->RefType
    + iconst_0_2->IntType
    + iload_0_3->IntType
    + iastore_4->IntType
fact atypes{
  atype =startup->NullType
    + aload_1_1->IntType
    + iconst_0_2->NullType
    + iload_0_3->NullType
    + iastore_4->NullType
    + return_5->NullType
}

Since we add two relations “stack” and “aref” to the State signature, as described previously in Alloy model part, we should also extend the initialState by clearing the sequence of stack types and assigning “NullType” to “aref”.

fact initialState {
  ...
  # (s0.stack) = 0) && (s0.aref = NullType)
}

5. Produce and test the malicious code

When we build an Alloy model and make the initialization successfully, we can click on the execute button and let the Alloy model test whether the Java method fits the rule we build in the Alloy model. In our project our Java method fits our Alloy model, it shows that ”No counterexample found. Assertion may be valid”.

The way we build malicious code is to modify the Alloy model initialization part. For example Below is the benign code:

fact types{
  type =aload_1_1->RefType
    + iconst_0_2->IntType
    + iload_0_3->IntType
    + iastore_4->IntType
}

fact flags{
  flag =startup->0
    + aload_1_1->0
    + iconst_0_2->0
    + iload_0_3->1
    + iastore_4->0
    + return_5->0
}
fact atypes{
  atype = startup->NullType
    + aload_1_1->IntType
    + ifalse_0_2->NullType
    + iload_0_3->NullType
    + iastore_4->NullType
    + return_5->NullType
}

We can change the property of "iconst_0_2" from "IntType" to other type and then run the Alloy model, then it shows "Counterexample found". The reason of turning up "Counterexample" is that when we change the property of "iconst_0_2" from "IntType" to other type, Alloy model will capture the type of "iconst_0_2" and store it in a type sequence. And Alloy model will compare the type of "iconst_0_2" with "IntType". If they do not coincide, there will be a "Counterexample found".

Another way to produce a malicious code is to change the property of "aload_1_1" from "IntType" to "DoubleType". Then reason of turning up "Counterexample" is that when we change the property of "aload_1_1" from "IntType" to other type, Alloy model will compare the opcode's "atype" property and "type" property. If they do not coincide, there will be a "Counterexample found".

**Resources and References:**
Lightweight Modeling of Java Virtual Machine Security Constraints using Alloy by Mark C. Reynolds