Secure Information Flow: from Java Bytecode to NuSMV

Leonardo Cecchelli
Zaccaria Essaid
Riccardo Xefraj

June 2020

Abstract

The objective of this project is to model in NuSMV the Java Bytecode instructions load, store, if, goto, halt to check automatically if a sequence of Java Bytecode instructions respects the secure information flow property.

1 Introduction

To achieve a valid way to check if the given Java Bytecode Instructions respected the secure information flow, we implemented the NuSMV model like a processor that executes a sequence of instructions. We associated to each variable a security level (High or Low). Associating to each variable only the security level allows us to check if some variable during the execution of the Java Bytecode instructions has changed its security level from low to high, and so we are able to detect a data leakage in the program we are analysing.

2 Model implementation

We modelled the Java Bytecode information flow using the abstract semantic rules and translating them inside of NuSMV.

2.1 Variables

Let us first have a look to the variables we decided to use to represent the Java Bytecode instruction flow.

```
VAR

variables:                    array 0..2 of {Ha,Lo};

actual_instruction_index:     0..9;

instruction_pointer:          0..9;

instruction:                  {load,store,if,goto,ipd,halt};

program_instructions:         array of 0..9 of

                              {load,store,if,goto,ipd,halt};

instructions_parameters:      array 0..9 of -1..9;

stack_pointer:                0..2;

stack:                        array 0..2 of {Ha,Lo};
```

```
enviroment_security:          {Ha,Lo};

--

ipd_pointer:                  0..2;

ipd_stack_address:            array 0..2 of 0..9;

ipd_stack_security:           array 0..2 of {Ha,Lo};
```

We decided to represent the Bytecode's variables using the array `variables`, whose dimension has to be selected based on the total number of variables present inside the whole program instruction flow. Since we do not care about the value of each code's variable we just save their security level, in the specific `Ha` means that the variable has an high security level and `Lo` a low security level. One thing to point out is that the index of the array is basically the identifier of a variable, so for example if in our extraced Bytecode we have variables x, y and z, we can represent x with the index 0, y with the index 1 and finally z with the index 2.
Then, like in a processor, we need a sort of instruction memory and a program counter. For this purpose we used `program_instructions` to save the code's instruction in the correct order, starting from the initial instruction representend by the element in postion 0 and the final one represented by the element in postion 9 (in this case). As we previuosly stated, NuSMV doesn't allow dynamic allocation of array's elements so we need to adjust each time the size of the array by looking at the exact number of instructions that are present in our Bytecode. Then of course we need the program counter, which in this case is `instruction_pointer`. This variable simply saves the value of the next operation that has to be executed in the next state of the model. We also added `actual_instruction_index`, which essentially has the same purpose of `instruction_pointer`, althought the values saved inside of it represent the index of the instruction we are currently executing. This could sound like a redundant information but, as you will see next, it will simplify our code. Lastly the variable called `instruction` saves the opcode of the instruction we are currently executing. In our model there are 5 possible opcodes (`load,store,if,goto,halt`), with the addition of an extra one called `ipd` that we will discuss in the next lines.

The opcode `ipd` does not belong to the Java Bytecode's instruction set, instead we added this in order to have the information about the immediate post dominator of a conditional branch. This was necessary because during the model simulation we would not be able to calculate it dynamically everytime we encour into an if operation. This instruction has to be inserted by external program that will parse the original Java Bytecode extracted from an application. We will discuss about the parser program in the *4.1 paragraph*.

After we modelled the instructions we then need to take care of the operand stack. We represented the stack with an array called `stack` that will contain the security level of each variable that has been saved (pushed) into it. We also used `stack_pointer` to save the value of the first free position in the stack; therefore all the values in the stack, whose index is bigger than the `stack_pointer`, can be ignored.

The variable called `enviroment_security` represents, as the name suggests, the enviroment security level, so its values can be `Ha` (high) or `Lo` (low) like for the program variables.

Finally we then added some data structure in order to cope with the addtion of the `ipd` instruction. Basically we need to save the `ipd`'s value when we encour in it during the execution of the program instructions. We decided to use the same approach we used for the JavaBytecode stack since this solution will be very useful when we have nested conditional statements. So we added `ipd_pointer`, `ipd_stack_address`, and `ipd_stack_security` in order to create the same configuration. In few words `ipd_pointer` works like the stack pointer and the `ipd_stack_address` is the stack memory where we save the immediate post dominator value of the conditional branches. Note that the top element of the `ipd_stack_address` is the ipd value of the if branch we are currently executing. Then we added a third and last array called `ipd_stack_security`, that we use to save the value of the enviroment security level at the time we encourred into the if operation. Then this value will be removed off the stack and used to restore the correct enviroment security level once we exit from the conditional branch.

2.2 Assignments

In this paragraph we will briefly analyze how variables assignments works, detailed information can be found inside of the project source code.

2.2.1 Load

The load instruction saves a variable security level at the top of the stack. The saved security level is the upper bound between the environment security level and the variable security level.

2.2.2 Store

Store instruction assigns the value on the top of the stack to the variable specified by the instruction parameter. The stack pointer is then decremented.

2.2.3 If

The if instruction generates two possible states: one with the instruction pointer updated with the jump value of the condition (true condition case) and one that follows the normal instruction flow (false condition case). Once we entered the condition branch, the model saves the environment security level, the immediate post dominator and may also upgrade the security level of the stack and the environment, depending on the level of the variable evaluated in the condition.

2.2.4 Goto

The goto instruction change the value of the instruction pointer with the correspondent value of the instruction parameter.

2.2.5 Ipd

Ipd stores its parameter on the top of the ipd stack for the addresses and the environment security level on the top of the ipd stack for the security levels. The ipd stack pointer is then incremented.

3 Conclusions

At the actual development of this model we are already capable to check for data leakage. Actually, it's possible to automatically analyse SIF violations by specifying the path formulae we want to be respected. This approach that relies on abstract operational semantic is very effective since it allows to explore all the different execution path of the program, without having to test all the possible values that program variables can take. There are although some cons to take in consideration, one is that we have each time to determine a priori the dimensions of all the NuSMV variables, based on the code we want to examine. In the next paragraph we will present an automated solution that allows to parse the code and prepare the NuSMV simulation in a semi-automatic way. Part of this has been already implemented while some parts still require further developments.

4 Future developments

4.1 The Bytecode parser

4.1.1 How to read Bytecode form a .class file

- Download Bytecode Viewer (https://www.bytecodeviewer.com/)
- Execute the .jar file from terminal (java -jar <BytecodeViewer.jar>)
- Drag and drop the .class in the Bytecode Viewer application
- Click on the .class from the Bytecode Viewer application
- Start navigating the content of .class file
- Click the file that you are interested in and the bytecode will be shown

4.1.2 Bytecode content for each method

It is important to notice that each method at the beginning associates an index to each variable called inside it.

```
private clone2(java.io.FileReader arg0) throws java/io/FileNotFoundException,
java/io/IOException { //(Ljava/io/FileReader;)V

<localVar:index=6 , name=i_cloned , desc=Z, sig=null, start=L1, end=L2>

<localVar:index=5 , name=i , desc=I, sig=null, start=L3, end=L4>

<localVar:index=0 , name=this , desc=LPINcloner/PINcloner;, sig=null, start=L
5, end=L6>

<localVar:index=1 , name=PINFile , desc=Ljava/io/FileReader;, sig=null, start
=L5, end=L6>

<localVar:index=2 , name=ex , desc=Ljava/lang/Integer;, sig=null, start=L7, e
nd=L6>
```

```
<localVar:index=3 , name=PIN , desc=Ljava/lang/String;, sig=null, start=L8, e
nd=L6>

<localVar:index=4 , name=j , desc=I, sig=null, start=L9, end=L6>
```

Each java instruction is translated to a set of simple Java Bytecode instructions with an associated name (usually L<Number>):

```
L8 {
            iconst_0
            istore4
    }
```

This instruction corresponds to the java instruction `j=0;` (We can see that istore is invoked on the variable with the index 4 and that the name of the variable associated to that index is “j”).

4.1.3 What the parser should do

Since we are analysing only a few of the instruction that Java Bytecode has at his disposure, a parser must remove and convert some of the instructions.

Our NuSMV model accepts only load, store, if, goto and halt instructions. Since the java Bytecode uses different types of loads and stores, according to the variable that the instruction is loading into the stack, its needed to translate them in a single load and store instruction.

Example: The load instruction has different variants like aaload (to load array reference), iload (to load an integer), lload (to load a long). All these instructions need to be translated into a generic load, without taking care of the type of the variable we are loading (same for the store instructions).

Every if condition should be translated into an “if” instruction with the line of the instruction where that if jumps as parameter.

Example: ifge if_icmpne and the others should be translated in a simple if

The variable index associated with an instruction is written next to the instruction without a space. The parser should add a space between the instruction and the parameter.

Example: istore4 -> store 4 (eliminated the information about the type of the variable we are going to store and added a space between the instruction and the variable index)

In case an instruction if (of any type) is encountered in the java bytecode of our program, an instruction name "ipd" should be inserted before it. This ipd instruction should have as parameter the line number where our branch ends (so when we arrive at the specified instruction the environment security level can be restored). Note that each time we find an instruction without parameters, -1 is added as parameter.

Example:

0 - Load 0	0 - Load 0
1 - Load 1	1 - Load 1
2 - If 5	**2 – ipd 7**
3 - Store 1	3 - If **6**
4 - Goto 6	4 - Store 1
5 - Store 2	**5 - Goto 7**
6 - Halt	6 - Store 2
	7 - Halt

4.1.4 Overcoming NuSMV constraints

At this stage, once we have all the Bytecode parsed, we need a program that initialize our .smv model. In order to do this automatically, we created a Java program that scans the parsed Bytecode and fills the NuSMV variables. At the moment we have not implemented a dynamic assignment of the NuSMV variables dimensions, so we have constraints regarding the number of instructions, parameters and variables based on the dimensions written in the NuSMV file. This is due to NuSMV requirements regarding static dimensions of variables. Further developments need to solve these constraints by creating a parser program that fills the NuSMV file with dynamic dimensions, in order to allow the execution of arbitrary number of instructions and parameters.