Analisys and Verification of Program Project 2014

Gianluca Barbon
gbarbon@dsi.unive.it

Cristian De Zotti
cdezotti@dsi.unive.it

Marco Moscardo
mmoscardo@dsi.unive.it

Abstract

Your Abstract Text Goes Here. Just a few facts. Whet our appetites.

1 Introduction

Vogliamo verificare se delle variabili dell'applicazione contengono dati sensibili, a seguito dell'uso di determinati metodi che in esse hanno salvato i loro risultati.
Nota: utilizzeremo l'abbreviazione "metodi sensibili" per metodi che generano dati contenenti informazioni sensibili e "dati sensibili" per indicare i dati generati da metodi sensibili. Utilizzeremo il verbo "esportare" per indicare tutti i possibili metodi di salvataggio/esportazione di dati sensibili.
La domanda principale diventa quindi: in un'applicazione che genera dati sensibili, sono questi dati esportati?

1.1 Confidential Data Definition

Blabla

2 Safeness Analysis Formal Definition

Our analysis will be based on the Data Flow analysis techniques, and will be named *Safeness Analysis*. As said before, it will identify exported elements that contains confidential data. This analysis will determine exported and possibly confidential variables at the entry and exit point of each node. We will consider the collection of values composed of $\{x, x[y,z,\ldots], n\}$ where x is the exported variable, [y,z,...] is the chain of variables used in the definition of x and n is the involved node. Moreover, we will consider $\{x,y,n\}$ as the variable x equal to y in node n. Please notice that this last statement has different meanings depending on the fact that the variable consist in an object or in a primitive element, but we will see this later in a more precise way.

Thus, variable x and its chain are exported and potentially confidential if:

- a path p from n to m exists
- variable x is exported and potentially confidential in n
- variable x is never re-defined in p

We have to consider that our *Safeness Analysis* is taking place in an Object Oriented environment. Anyway, we will deal later with complex concepts from O. O. programming, like interfaces and interfaces. On the other hand, we will first start from a simple pseudo-code version of an object oriented programming paradigm. That's to say, an o.o. programming language similar to Java that will use only classes, methods and creators, in order to "make it simple".

2.1 Gen/Kill

Let's consider operations that generate and kill information:

$$gen_{SA}[\mathrm{B}] = \{element\ is\ exported\}$$

$$kill_{SA}[\mathrm{B}] = \{element\ is\ not\ created\ by\ a\ confidential\ method\}$$

A more precise definition of *gen* and *kill* could be made considering also variable that are not exported directly, but are used in the definition of exported variable. But we decided to have simple *gen* and *kill* definition, and to improve it later for specific cases (please refer to section 2.3).

In order to collect only variables that are exported, our analysis will start from the end of the program. Thus, it

will be a *Backward Analysis*. This behaviour is similar to the *Live Variables Analysis* [3] one. We can now specify the features of our analysis.

Analysis direction: backward

Confluence operator: union
$\text{out}[B] = \cup \text{in}[S]$, *over the successor S of B*

Initialization: empty set, $\emptyset$

It is now possible to define the equations of the analysis.

2.2 Safeness Analysis equations

$$SA_{exit}(p) = \begin{cases} \emptyset & if\ p\ is\ a\ final\ point \\ \\ \cup \{SA_{entry}(q) \mid q\ follows\ p\ in\ the\ CFG\} \end{cases}$$

$$SA_{entry}(p) = \text{gen}_{\text{SA}}(p) \cup (SA_{exit}(p) \setminus \text{kill}_{\text{SA}}(p))$$

At the end of the analysis we will have two possible states:

- no more variables collected in our tables
- some variables are still present in our tables

The first state means that the examined program is *SAFE*. Indeed, if no variables are present in our analysis memory, all the variables considered possibly unsafe have been removed because they are certified as non confidential data.
On the other side, the second state means that the program is *UNSAFE*. This is due to the fact that some variables remains in our analysis memory, confirming that these variables contain confidential data. Thus, our analysis is also capable of locating unsafe variables.

2.3 Difference between objects and primitive data types

The definitions given so far are not sufficient to describe our analysis. Indeed *gen* and *kill* are more complicated. We have to deal with different problems related to the object oriented paradigm. First of all, we have to consider that also variables used in the definition of elements that will be exported may contain confidential data. So a complete analysis has to consider also these variables. Second, it is important to divide variables into two subsets:

- primitive variables
- objects

This necessary distinction comes from various assumptions. Firs of all, in an object oriented programming language objects are passed by reference, while primitive variables are passed by copy. In example, in Java a variable that uses a class as type does not actually contain an object of that class. Instead, it only stores the position of the object in the system memory [2]. So we present the different cases in a more precise way.

Primitive Variables

`export(x)`
Variable x is exported with a method. In this case the *gen* step will add the record $\{x,n\}$ in the table into the *gen* column.

`x = y`
The variable is copied from another one, and we know that it will be exported (notice that, in order to make this *gen* step sound, we also have to know that variable x will not change its value in the path between the copy and the exportation points). In this case the *gen* step will add the record $\{x,y,n\}$ in the table into the *gen* column. A later *kill* (going backward after this statement) of the copied variable y will remove the information added with this *gen* step, and also the information related to the exportation of the x variable.

`x = method(y)`
The variable is used in the creation of another one, and the last one is exported. In this case the *gen* step will add the record $\{x,x[y],n\}$ in the table into the *gen* column. If the primitive variables involved as parameter are more than one, we will have a record like this: $\{x,x[y,x,...],n\}$. A later *kill* of the referred variable y will remove the information added with this *gen* step, but will not delete the information related to the exportation of the first variable x. This is because, even if y variable is *SAFE*, the method that generates the first one may make use of confidential information, like other methods or variables belonging to its class, and so we cannot trust variable x. In order to check the safeness of variable x, we must expand the *CFG* of this method.

`x = method()`
The variable x is generated. If, by expanding and inspecting the *CFG* of this method, we find out (or we already know) that it's *not confidential*, we can perform a *kill* step. Otherwise, no steps are performed.

Objects

`x.export()`
: Object x is exported with a method. In this case the *gen* step will add the record $\{x,n\}$ in the table into the *gen* column.

`CClass x = y`
: The object is referred from another one, and the last one is exported. In this case the *gen* step will add the record $\{x,y,n\}$ in the table into the *gen* column. A later *kill* of the referred object (y) will remove the information added with this *gen* step, and also the information related to the exportation of the first object (x).

`CClass x = new CClass(y)`
: The object is used in the creation of another one, and the last one is exported. In this case the *gen* step will add the record $\{x,x[y],n\}$ in the table into the *gen* column. A later *kill* of the referred object y will remove the information added with this *gen* step, but will not delete the information related to the exportation of the first object (x). This is because, even if y object is *SAFE*, the creator of the first one may make use of confidential information, and so we cannot trust the x object. In order to check the safeness of the x object, we must expand the *CFG* of its constructor.

`CClass x = new CClass()`
: The object x is created. We cannot make any *kill* step, because we don't know if the constructor of this object is *SAFE*. As said for the previous case, we must expand and inspect its *CFG*.

As we have just seen, primitive variables and objects have different behaviour in the *gen* and *kill* steps. Anyway, the syntax used in the tables of the analysis is quite similar.

2.4 CCFG: Class Control Flow Graph [1]

Blabla

2.4.1 CCFG Constructor Algorithm

```
algorithm: ConstructorCCFG(C):G
input: C a class
output: G a CCFG of C

begin ConstructCCFG

  /* Step1: Construct the class call
  graph for the class */
   G = Construct the class call graph of G

  /* Step2: Replace each call graph node
  with the corresponding CFG */
   foreach method M in C do
    Add M class to C
    Replace M node in G with CFG of M class

  /* Step3: Connect every individual
  CFG */
   Update edges appropriately

  /* Step4: Return the completed
  CCFG of G */
   return G

end ConstructCCFG
```

2.5 Safeness Analysis Algorithm

```
for each n
in[n]:= ; out[n]:=
repeat
for each n
inâĂŹ[n]:=in[n]; outâĂŹ[n]:=out[n]
in[n] := use[n]∪(out[n]-def[n])
out[n]:=
```

3 Examples

We now apply our analysis to some examples written with the proposed simple version of an object oriented programming language.

3.1 Example 1

In this example we will see a very simple application that uses two class.

3.1.1 Code

```
. class MClass{
.
1     void main (String[] args) {
2         Text t  = new Text("Hello!");
3         exportMethod(t);
.     }
.
4     void exportMethod(Text t) {
5         ...
.         //this method exports confidential values
.     }
. }
```

We now have to consider Text class:

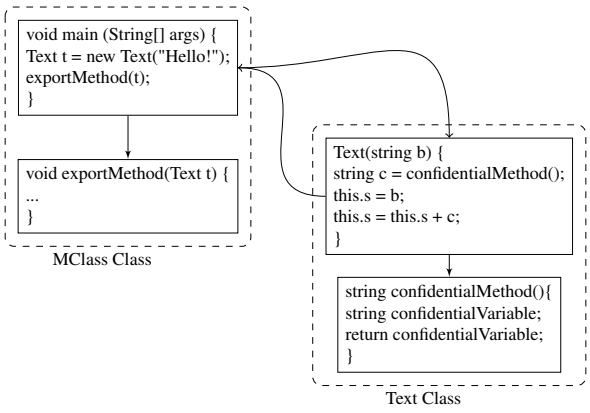
```
. class Text {
1     string s;
.
```

```
2     Text(string b) {  //constructor
3         string c  = confidentialMethod();
4         this.s = b;
5         this.s = this.s + c;
.     }
.
6     string confidentialMethod(){
.         //method with confidential values
7         string confidentialVariable;
8         return confidentialVariable;
.     }
. }
```

3.1.2 Description

By only looking at *MClass* object *t* seems to be not confidential. Anyway, we must investigate also *Text* class, because the constructor may contains methods that create confidential data in the new object. In order to do this, we can expand the CCFG of the *text* constructor. In this way, we will see that the constructor makes use of *ConfidentialMethod*, a method that generates confidential data. So, the object *t* will be considered *UNSAFE* and consequently also the application will be believed as *UNSAFE*.

3.1.3 CCFG



3.1.4 Tables

n	classname	gen	kill
3	MClass	{t,,3, MClass}	
2	MClass	-	-
5	Text	{t,t[c],5, Text}	
4	Text	{t,t[c,b],4, Text}	
3	Text	-	-
8	Text		
7	Text		
6	Text		
3	Text		
2	Text		
2	MClass	{b,2,MClass}	
1	MClass		

n	classname	entry	exit
3	MClass	{t}	
2	MClass	-	-
5	Text	{t, c}	{t}
4	Text	{t, c, b}	{t, c}
3	Text	-	-
8	Text	{t, c, b}	{t, c, b}
7	Text	{t, c, b}	{t, c, b}
6	Text	{t, c, b}	{t, c, b}
3	Text	{t, c, b}	{t, c, b}
2	Text	{t, c, b}	{t, c, b}
2	MClass	{t, c}	{t, c, b}
1	MClass	{t, c}	{t, c}

3.2 Example 2

3.2.1 Code

```
1    CustomClass x = new CustomClass();
2    CustomClass y = x;
3    y.export();
```

3.2.2 Description

Partiamo dalla fine. Gen sull'ultima istruzione ci genera y. Sulla penultima dobbiamo killare y? No. Facciamo gen di x sulla catena di y. Infatti, anche se non viene generato da un metodo sensibile, non sappiamo ancora se x sia sensibile o meno. Nel caso lo fosse, considereremmo anche y sensibile. Quindi per il momento non abbiamo kill. Continuando a risalire nel codice, troviamo il punto dove viene definita la variabile x. Si tratta di un metodo sensibile, quindi non facciamo kill. Inoltre, ora sappiamo che anche y è sensibile. Questo esempio risulta quindi UNSAFE.

Consideriamo il caso in cui x ÃÍ sicuramente non sensibile. In questo caso abbiamo un comportamento simile al precedente, ma la situazione cambia nel momento in cui scopriamo che x non proviene da una funzione sensibile. A questo punto, verranno killate sia x che y. Alla fine avremo l'insieme vuoto, ciò significa che quest'esempio è SAFE.

The last example has been created following the first example of section 2 of this document. We first notice that we can have various possibilities when creating a new object:

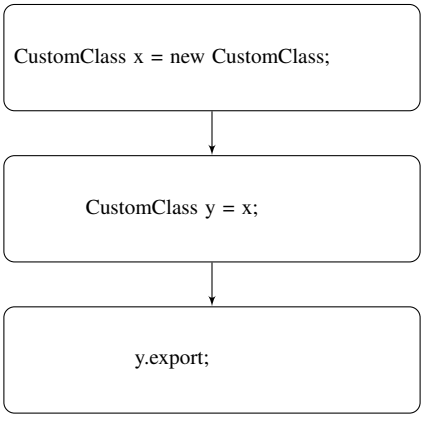
- constructor doesn't make use of confidential data
- constructor USES confidential data
- parameters used in the constructor invocation contain confidential data
- ...

If we apply backward analysis to this example, we will have the following steps:

1. gen step for object y (with export method)
2. x is added to the chain of y
3. the kill step of x and y depends on the results of the analysis of the constructor method and of the CustomClass class (notice that variables declared in the class may be obtained as results of confidential methods)

Furthermore, it's important to remark that a class may declare confidential variables, anyway if we don't have methods that export these variables, the object is SAFE. More precisely, objects may contain methods that exports confidential data, but if these methods are not used in the analysed application, these objects can be considered SAFE.

3.2.3 CCFG



3.2.4 Tables

n	gen	kill
3	$\{y\}$	
2	$\{y,x\}$	
1		$\{x\}$

n	exit	entry
3		
2		
1		

4 Issues with a real Object Oriented Language

Blabla

4.1 Inheritance

Inheritance Blabla

4.2 Extension

Extension Blabla

4.3 Interfaces

Interfaces Blabla

4.4 Overloading

Overloading Blabla

4.5 Overriding

Overriding Blabla

4.6 Recursion

Recursion Blabla

4.7 ...

Blabla

5 Android Environment

Blabla

5.1 Methods and data structure selection

Blabla

5.2 Analysis adaptation to Android Environment

Blabla

5.3 Android App Example

Blabla

6 Conclusions

Blabla

6.1 Future developments

Blabla

References

[1] HARROLD, M. J., AND ROTHERMEL, G. Performing data flow testing on classes. *Department of Computer Science Clemson University, Clemson*.

[2] HORSTMANN, C. S. *Big Java*, 2nd ed. John Wiley & Sons, Inc., New York, NY, USA, 2008.

[3] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999, ch. 2.1.4.

Notes

[1] Remember to use endnotes, not footnotes!