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

The aim of this analysis is to verify if the application variables contain confidential data, as a result of the use of certain methods that they have saved their results. In this paper we purpose an analysis to find all variables which are classified as potential exported variables using the notion of Class Control Flow Graph. First of all we explain the idea of this analysis applying it to simplified object oriented programs and then using it to extended object oriented language which are the same of the language used by the Android platform. Note: we use the abbreviation "confidential methods" for methods that generate data containing confidential information and "confidential data" to indicate the data generated by confidatial methods. We use the verb "export "to indicate all the possible ways to save/export confidential data. Now the main question becomes: in an application that generates sensitive data, are these data exported?

1.1 Data Definition

Data can be classified in three different classes: *personal, confidential or sensitive.* The main difference between these types of data is the kind of information which are related.

Personal data is '*data which relates to a living individual who can be identified from those data OR from those data and other information which is in the possession of or is likely to come into the possession of the data controller (e.g. University) and includes any expression of opinion about the individual*' - Data Protection Act (1998).

Confidential data is data that can be connected to the person providing them or that could lead to the identification of a person referred to, such as names, address, etc. This information, which are not in the public domain, are given in confidence, or with data agreed to be kept '*confidential*' (namely secret) between two parties.

Sensitive data is '*data on a person's race, ethnic origin, political opinion, religious or similar beliefs, trade union membership, physical or mental health or condition, sexual life, commission or alleged commission of an offence, proceedings for an offence (alleged to have been) committed, disposal of such proceedings or the sentence of any court in such proceedings* - Data Protection Act (1998)

In this paper we are only focused on the confidential data that is the most exported data without authorization.

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 (possibly, we can use node number n in association with the name of class, separated by a comma, in case the program has more than one class). 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 inheritance 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
$\mathrm{out}[B] = \cup\, \mathrm{in}[S]$, *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

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

$$SA_{entry}(p) = \mathrm{gen_{SA}}(p) \cup (SA_{exit}(p) \setminus \mathrm{kill_{SA}}(p))$$

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. 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*.

2.4 CCFG: Class Control Flow Graph [1]

For the construction of our Control Flow Graph, we must use a different technique in object-oriented language with respect to an imperative language. This is due to the fact that in the object-oriented language multiple classes exist within the same application, and thus our graph must be able to represent the application in its entirety, including the relationship between the various classes given by method calls.
It has been developed an algorithm that allows the construction of CCFG of the application, considering also the relationships between the various classes. The algorithm consists of four steps:

- The first step is to create the CFG of the class in question.
- In the second step, however, for each method inside the class, the class of the method is added to ours, and we are replacing the node that contains the method invocation, with the corresponding CFG. This operation continues until all the methods contained within the application are covered, including those present in the interior of the classes that are added in the course of work.
- In the third step the update of the various edges of CCFG is made, so that they are connected in an appropriate manner on all sides.
- In the fourth step, the final one, once finished all the above steps, the algorithms returns the CCFG application.

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 C*/
   G = Construct the CFG of C

  /* 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

The following section shows the algorithm used to perform the safeness analysis, which was derived from the equations introduced in Section 2.2, applied to the CCFG of the application:

for each node n in CCFG
  in[n]:= ∅; out[n]:= ∅;
repeat
  for each node n in CCFG **in reverse topsort order**
    in'[n]:= in[n];
    out'[n]:= out[n];
    out[n]:= ∪{in[m] | m ∈ succ[n]};
    in[n]:= gen[n] ∪ (out[n]-kill[n]);
until (in'[n]= in[n] && out'[n]= out[n] for all 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

3.1.1 Code

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

3.1.2 Description

This example has been created in order to show how important can be a Class Constructor in considering a variable confidential or not. 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

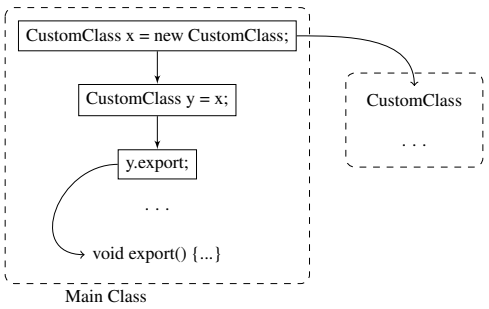
If we apply our analysis to this example, we will have the following steps:

1. gen step for object y (with export method)
2. object y results as a reference to x, so also x is added in the table
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)

We can see that it is important to expand completely a CCFG in order to discover all the interesting path. So, by expanding the CCFG of *CustomClass* we can have two cases:

- constructor USES confidential data, so variables *x* and *y* cannot be killed, thus the application is *UNSAFE*
- constructor doesn't make use of confidential data, so variables *x* and *y* are both killed and the application is *SAFE*

3.1.3 CCFG



3.1.4 Tables

We present here two possible version of tables. The first one show the case in which the constructor is *UNSAFE*.

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

n	entry	exit
3	{y }	
2	{y, x}	{y }
1	{y, x}	{y, x}

While the second one show the case in which the constructor is *SAFE*. As we can see in the tables below, the variables under suspicion are killed and the application is considered *SAFE*.

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

n	entry	exit
3	{y }	
2	{y, x}	{y }
1		{y, x}

3.2 Example 2

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

3.2.1 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
.     }
. }
```

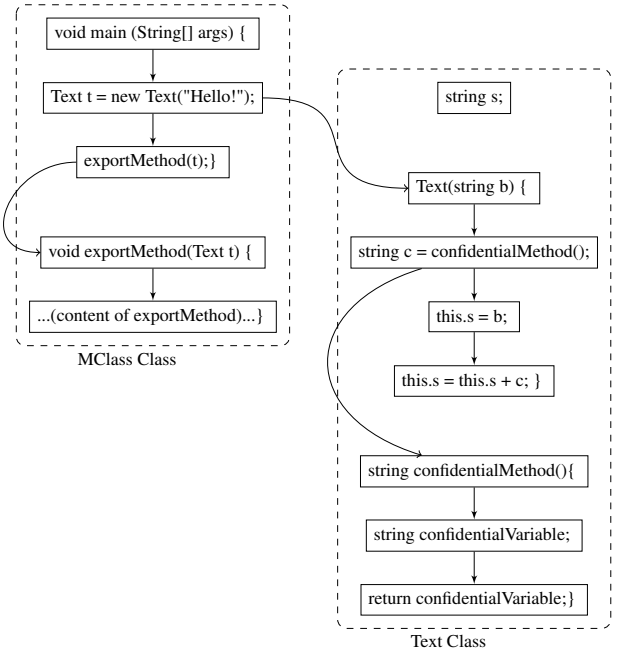
We now have to consider Text class:

```
. 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(){
.         //creates confidential values
7         string confidentialVariable;
8         return confidentialVariable;
.     }
. }
```

3.2.2 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*.

3.2.3 CCFG



3.2.4 Tables

n	classname	gen	kill
3	MClass	{t,,3, MClass}	
5	Text	{t,t[c],5, Text}	
4	Text	{t,t[c,b],4, 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}	
5	Text	{t, c}	{t}
4	Text	{t, c, b}	{t, c}
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}

4 Issues with a real Object Oriented Language

This section will explore some specific features of the object oriented programming languages applied to our analysis, in order to see whether it works or it should be expanded. When necessary and possible, we propose a solution to extend the analysis to these feature. Descriptions may make use of examples with the basic pseudocode already used in section 3, expanded with the required object oriented features.

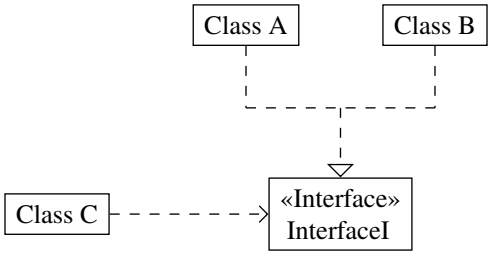
4.1 Interfaces

An interface is used in an object oriented language in order to specify a set of methods that classes must implements. In example, we can have the following interface:

```
public interface InterfaceI {
    int commonMethod();
}
```

It declares a method that will be used by classes that will implement this interface, i.e. classes A and B, that must also implement the given method. Notice that the interface only states the method signature. In this way, all classes can implement the method in a different way, and another class that uses both classes A and B, in example, class C, will use only one method (and it can also use one type for both, the one of the interface).

Graphically, we will obtain this structure:



4.1.1 Polymorphism

In languages similar to Java a problem appears when more than one class implement the same interface. In this case, it seems difficult to choose the right method (declared in the interface) for a given object. This circumstance is related to a principle called *polymorphism*. Applied to interfaces, this concept states that the real type of an object can determine the correct method to call.

As for our analysis, it can be difficult to face this behaviour. Java does not solve it at compiling time, but only at execution time, by choosing the correct method using the *late binding* [2]. Let us take the previous example. Class C may do not want to know the correct type of an object, and use the interface type:

```
public class C {
    InterfaceI x;
    ...
    x = new A();
    x = new B();
    ...
    x.commonMethod();
}
```

What is the right implementation of commonMethod that we should use? The situation is simpler that one might think. Indeed, when the CCFG algorithm executes, it expands every method. If it runs into a variable with type of an interface, we know that sooner or later it will also find the correct constructor (the interface has no constructor). If we look at the examples, when we will meet *new A*() or *new B*() we will expand the CCFG also for the constructor, thus knowing the correct type of the variable. At the end, when we will meet commonMethod(), we will know that it refers to the last constructor used, so we will use the implementation given by class B.

4.1.2 Overloading

In object oriented programming we have principle similar to the polymorphism, called constructor and method overloading. It consists into functions with same name

but different parameters, that are invoked depending on the type of the parameters. Anyway, for us this is easier to face than polymorphism. Indeed, problems may only arise in the selection of the correct method during the construction of the CCFG. The CCFG algorithm must thus choose the right method by looking at its signature.

4.1.3 Inner Classes

Inner classes are classes declared and implemented internally to another one. As for our analysis, everything is solved during the execution of the CCFG algorithm by expanding the used methods in a way similar to normal classes.

4.2 Inheritance

4.2.1 Polymorphism

4.2.2 Overriding

4.3 Recursion

The recursion is used when we can see that our problem can be reduced to a simpler problem that can be solved after further reduction.

Every recursion should have the following characteristics:

- A simple base case for which we have a solution and a return value.
- A way of getting our problem closer to the base case. I.e. a way to chop out part of the problem to get a somewhat simpler problem.
- A recursive call which passes the simpler problem back into the method.

Example:

```
void myMethod(int counter)
{
if(counter == 0)
     return;
else
       {
       System.out.println(""+counter);
       myMethod(--counter);
       return;
          }
}
```

In the case of recursion, our analysis works well and does not need to change, because once analyzed the code of the recursive method and seen if it is safe or not, there are no additional steps.

Caution, however, must be made at the time of the construction of CCFG. This is because in the case of recursion, the control flow graph of the method must occur only once, adding an edge that creates the cycle, without having to replicate the CFG for each call. Indeed, if the method is analyzed only once we do not need to repeat the analysis. In this way we avoid the possibility of incurring into an infinite replication, or otherwise having a CCFG of large dimensions that in the end would be too heavy and useless.

4.4 Data Structures

The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes:

- **Enumeration:** The Enumeration interface isn't itself a data structure, but it is very important within the context of other data structures. The Enumeration interface defines a means to retrieve successive elements from a data structure.
- **BitSet:** The BitSet class implements a group of bits or flags that can be set and cleared individually.
- **Vector:** The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.
- **Stack:** The Stack class implements a last-in-first-out (LIFO) stack of elements.
- **Dictionary:** The Dictionary class is an abstract class that defines a data structure for mapping keys to values.
- **Hashtable:** The Hashtable class provides a means of organizing data based on some user-defined key structure.
- **Properties:** Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Safeness Analysis in Data Structures should only pay attention to the methods applied to the various elements of the structure, and verify that they are safe or not, so as applies to simple variables / objects, it works in the same way for the elements of a data structure.

4.5 Exception Handling

Blabla

5 Android Environment

Blabla

5.1 Methods and data structure selection

Blabla

5.2 Analysis adaptation to Android Environment

Blabla

5.2.1 Differences between our language (simil-Java) and Android

5.3 Android App Example

Blabla

5.3.1 Code

5.3.2 Description

5.3.3 CCFG

5.3.4 Tables

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!