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 Aim of our project

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?

2 Solution Proposal

Utilizziamo Data Flow Analysis. Ipotizziamo di nominare quest'analisi "Safeness Analysis". Quest'analisi determina le variabili che esportano dati sensibili ad ogni entry ed exit point di un nodo. Consideriamo l'insieme di valori formato da (x, x[...], n), dove x[...] è la catena formata da variabili usate nella definizione di x. La variabile x e la sua catena sono esportate e potenzialmente sensibili se:

- esiste un percorso p da n a m
- la variabile x è esportata e potenzialmente sensibile in n
- la variabile x non è mai ridefinita in p

Consideriamo quindi le operazioni che generano e killano l'informazione:

$$gen_{SA}[\mathrm{B}] = \{var\ x\ is\ exported \text{ OR } var\ x\ is\ used\ by\ an\ exported\ variable\ y\ (so\ x\ is\ added\ in\ the\ chain\ of\ y)\}$$

$$kill_{SA}[\mathrm{B}] = \{var\ x\ is\ not\ created\ by\ a\ sensible\ method \text{ OR } chain\ of\ var\ x\ contains\ only\ variables\ that\ are\ not\ sensib$$

Definiamo ora:

- direzione dell'analisi: backward
- confluence operator: unione, $\mathrm{out}[B] = \cup\, \mathrm{in}[S],\ over\ the\ successor\ S\ of\ B$
- inizializzazione: insieme vuoto $\emptyset$

Possiamo formalizzare le equazioni della Safeness Analysis:

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

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

Se alla fine dell'analisi otteniamo l'insieme vuoto, il programma sarà SAFE. Altrimenti il programma sarà UNSAFE e avremo ottenuto l'insieme di tutte le variabili sensibili esportate.
Di seguito alcuni esempi.

2.1 Example 1

```
x = mysensibledata(); $ ... $
```

```
...
y = x + 1; $ No\ kill\ $
...
export(y); $ \text{gen}:\ y\ $
```

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.

2.2 Example 2

```
x = 5;
...
y = x + 1;
...
export(y);
```

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.

2.3 Conclusions

Risulta quindi necessario dover utilizzare una catena/array di riferimenti, per sapere tutta la lista di variabili che ipotizzate sensibili che sono collegate. Ciò ci permetterebbe inoltre di gestire tutte le problematiche relative alle variabili passate per riferimento. Questa bozza prende in considerazione esempi in pseudocodice 'imperativo'. Tutto questo lavoro, se confermato, dovrà quindi essere rivisto per poter essere adattato al paradigma ad oggetti del linguaggio android (considerando quindi le relative problematiche).

3 Next step: Safeness Analysis in Object Oriented programming language

The next step consist in redefine our Safeness Analysis in an Object Oriented environment. 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 that will use only classes, methods and creators, in order to "make it simple". Let's take some examples.

(Notice: for correctness, we here redefine 'sensibile' (sensibile) as 'confidential', as the previous translation from Italian was not correct.)

3.1 Example 1

```
...
Text t = new Text(confidentialText);
...
```

But we also need to consider Text class:

```
class Text {
    string s; //class variable

    Text(string b) {  //constructor
        this.s = b;
    }
}
```

In this example it is easy to see that the object t contains confidential datas.

3.2 Example 2

```
...
Text t = new Text("Hello!");
...
```

We now consider Text class:

```
class Text {
    string s; //class variable

    Text(string b) {  //constructor
        string c  = confidentialMethod();
        this.s = b;
        this.s = c + b;
        //considering string concat allowed
    }

    confidentialMethod(){
        //method with confidential values
        string confidentialVariable;
        ...
        return confidentialVariable;
    }
}
```

In the second example object t seems to be not confidential. anyway, we must investigate also class Text, because the constructor may contains methods that create confidential data in the new object.

3.3 Example 3 (from example 1 of section 2)

```
...
CustomClass x = new CustomClass();
...
CustomClass y = x;
...
y.export();
```

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.

3.4 Formal redefinition of gen and kill

Let's consider operations that generate and kill information:

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

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

We define:

Analysis direction: backward

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

Initialization: empty set, $\emptyset$

Safeness Analysis equations:

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

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

Anyway these definitions are not sufficient. Indeed *gen* and *kill* are more complicated. First of all, it is important to make a distinction between objects and variables. This distinction will include important assumptions: first, in an object oriented programming language, like Java, objects are passed by reference, while primitive variables are passed by copy. Second... blablabla.
So we present the different cases in a more precise way.

Variables

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

`x = y`
: The variable is copied from another one, and we now 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). In this case the *gen* step will add the record $\{x,y\}$ 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 $\{x,x[y]\}$ in the table into the *gen* column. 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 $\{x\}$ 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 $\{x,y\}$ 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 $\{x,x[y]\}$ 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*.

4 CCFG: Class Control Flow Graph [1]

```
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
```

References

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

Notes

[1] Remember to use endnotes, not footnotes!