Taint Analysis of Java Programs using incremental Slicing

Ashwin asokan

1. Introduction:

Taint analysis is a type of information-flow analysis which helps to figure out whether variables and their values from any untrusted methods and parameters flows into security-sensitive operations. Taint analysis can be used to detect many common vulnerabilities such as SQL injections, buffer overflow, cross site scripting and information leakage in Web applications. Since information flow in a program cannot be verified by examining a single execution trace of that system,the results of taint analysis will necessarily reflect approximate information regarding the information flow characteristics of the complete program.

Two important security characteristics of programs can be asserted & verified using taint analysis.
Integrity where untrusted inputs flow into security-sensitive parts and
confidentiality where private information is revealed to public observers
Taint Analysis is generally broken down into figuring out the data flow points in the program boundaries and use them as points to track their flow across the program verifying necessary conditions.

A security rule is a triple <Src,Dwn,Snk>, where Src,Dwn and Snk are patterns for matching sources, downgraders and sinks in the subject program,respectively. A pattern match could be a method call or a field dereference. Depending on whether the problem being solved is related to integrity or confidentiality, a source is kind of a method that injects untrusted or secret input into a program, a sink is a method that performs a security-sensitive computation or exposes information to public observers, and a downgrader is a method that sanitizes untrusted data or declassifies confidential data, respectively

General taint analysis solutions so far statically detects data flows wherein information returned by a "source" reaches the parameters of a "sink" without being properly endorsed by a "downgrader".
A vulnerability is reported for flows extending between a source and a sink belonging to the same rule, without a downgrader from the rule's Dwn set mediating the flow.

Building data flow analysis table, can be a time-consuming task on large code bases, especially if very precise information flow is being tracked. While investing this effort once is reasonable, it won't be cost effective to conduct such analyses on a regular basis, for instance whenever a small update is being made. However, such updates could be frequent and analysis results computed for one version do not necessarily remain valid for the next version. Therefore, there has been studies performed to see what's the correlation between data flow analysis results of subsequent incremental versions of program and how to reuse some of the results from previous analysis effectively on the newer version results.
The objective of this project to study such approaches and implement an incremental analyzer to effectively compute the data flow analysis results.

2. Taint Analysis through Slicing:

```
1: public class CurrencyRate {
2: DataStore dataStore = Bean.getService("CurrencyRateService");
3: protected void doGet(HttpServletRequest req,HttpServletResponse resp) throws IOException {
4: String source = req.getParameter("source");
5: String target = req.getParameter("target");
6: try {
7:        DataStoreRequest request = new DataStoreRequest();
8:        request.set("sourceCurrency", source); // BAD
9:        request.set("targetCurrency", URLEncoder.encode(target)); // GOOD
10:       request.set("date", new String(Date.getDate())); // GOOD
11:       DataStoreResponse response = dataStore.invoke(request);
12:       resp.set(response.get("rate")); // BAD
13: }
14: catch(Exception e) {
15:       e.printStackTrace();
16: }
17: }
18: }
```

Above example illustrates a simple servlet program which takes currency pairs as input and gives back rate relation between them as output. It in turn uses another underlying program to do the conversion. Line 9 & 12 in the example are marked BAD since it's not secure to assume that data being received from either side is safe. On the other hand similar data flow path of target & date variables are marked GOOD just because one originated within the program which is reliable and the other one has been validated and sanitized. So the static analysis model solution for taint analysis should be able to differentiate these two flows distinctively.

Slicing programs is an analysis technique to compute the set of programs statements, the **program slice**, that may affect the values at some point of interest, a **slicing criterion**. Taint Analysis can be modelled as a slicing problem in which we start either from source or sink and try to walk forwards or backwards collecting the statements of interest on our way. Once we have the slice we can query whether there is a downsizer with in it.

The slice is defined for a slicing criterion C=(x,V), where x is a statement in program P and V is a subset of variables in P. A static slice includes all the statements that affect variable v for a set of all possible inputs at the point of interest (i.e., at the statement x). Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies.

Two statements P & Q are said to be **data dependent** P -> Q if value of the variable assigned at P is being referenced in Q

Two statements P & Q are said to be **control dependent** P -> Q if Q can't be reached without reaching P in a program execution.

Below example demonstrates a slice computed for a criterion, [8,sum].

Original Program	**Slice[8,sum]**
1. int i; 2. int sum = 0; 3. int product = 1; 4. for(i = 1; i < N; ++i) { 5. sum = sum + i; 6. product = product * i; 7. } 8. write(sum); 9. write(product);	1.int i; 2. int sum = 0; 4. for(i = 1; i < N; ++i) { 5. sum = sum + i; 7. } 8. write(sum);

3. Slicing as a Data flow Problem

Slicing can be cast as a dataflow problem which involves

1. First computing local data flow relation from each statement - [Referenced, Defined]
2. Next interpret the program either forward or backward starting from source or sink respectively and compute relevance relation between the each statement and the source
3. Iterate till a fixPoint is reached with respect to Relevant set. Any statement which contributed to the relevance statement should be included into the Slice set.

Relevant(j) = (Relevant(i) ∩ Referenced(j) not empty)? Relevant(i) + Defined(j): Relevant(i)

Slice computed by condition (1,i) => {1,5,6}

n	Statement	Defined	Referenced	Relevant
1	Int i = n	{i}	{n}	{i}
2	Int sum = 0	{sum}	{}	{i}
3	Int product = 0	{product}	{}	{i}
5	sum = sum + i	{sum}	{sum,i}	{i,sum}
6	product = product * i	{product}	{product,i}	{i,sum,product}

Above data flow equation has been used to compute the rightmost column Relevant set. Defined and Referenced are local information with in the statement which can be computed independently.

4. Incremental Computation of Program Slices:

An incremental data flow analysis algorithm is one in which we avoid complete reanalysis after each modification, by determining and propagating program changes. The cost of such approaches depends on the program changes made and the size of the affected area. The possible program modifications are insertion and deletion of one or more source-level statements.
The changes made in the source code may be minor and result in changes in local data flow information within a node. In such cases, changes in the local data flow information can easily be propagated and usually affect only a small portion of the solution from the previous analysis. On the other hand, some program modifications may result in changes in the control flow structure. Such changes in general add to the amount of work of the incremental approaches.
Key step in incremental analysis is to compare the revised version with previous version of the program and estimate the dataflow facts which need to be recomputed to bring the overall data flow analysis solution relevant. Generally the type of changes gets categorized into three types and estimation depends on the category

1. Insertion of a new statement: Compute the defined and referenced set for the new statement. Then add all immediate successors of the new statement into the worklist. Both local & relevant data flow facts for them need to be recomputed.
2. Removal of an existing statement: Remove the dataflow facts from the solution. Then add all immediate successors of the existing statement into the worklist. As for the insertion all data flow facts for them need to be recomputed.
3. Changes within the statement: For any small changes within the statement, again the data flow facts need to be dirtied and recomputed. Statement need to be added to the worklist.

Once we collected the changes into the worklist we can process them linearly and on each iteration verify whether there are any changes to the data flow facts and on every change add their successor also to the worklist. If the fixpoint solution stabilizes then the procedure can be stopped.

Basic idea behind this incremental approach is restarting iteration. After iterating to a fixed point solution of a data flow problem,the problem is altered and the old fixed point is used as an initial value for the new problem. For our implementation, incremental changes that could be made to the program has been limited to above three changes, there could be control flow changes such as addition or removal of control flow arcs between statements. Approaches to handle such changes need to be studied further. They aren't considered in current implementation.

5. Implementation Details:

The general approach being followed to implement the incremental slicing is to break it down into two modules,

1.Slice Analysis: Straightforward from scratch implementation of forward data flow slice analysis as described in previous section. In nutshell it involves adding instruction by instruction to worklist and computing the data flow relation information linearly.

Input [initial program's call graph, Seed variables]

Output [slice for given seed variable, data flow facts against each program point]

2.Incremental Analysis: Compute the difference between previous and revised version of the program in terms of instructions. Spot the successor instruction for every difference and add to the worklist which need to be processed. In the incremental module we won't be interpreting the instructions again, instead we will be computing only the limited successor instructions of modifications being figured out from original.

Input [initial program's call graph, seed variables, slice for given seed variables, data flow facts from previous iteration]

Output [Revised Slice, Revised data flow facts from previous iteration]

The final segment will be recomputing the slice from scratch using the first module and compare it with the second module. We will be validating the incremental approach on two measures

1. Correctness: slice should be same as the from scratch computation.
2. Cost: Number of iterations it took to reach the fix point in both the cases.

6. Current Progress:

Both the module implementation has been coded on top of WALA library due to the familiarity being built through class exercises. WALA helps to transform the bytecode of the input program into call graph which in turn is being interpreted directly.

Slice Analysis - Implementation is complete.

Incremental Analysis - Partial. There has been some confusion in determining the difference in the different versions of the call graph after converting bytecode into IR representation particularly due to the conversion of regular variable names into static single assignment. Tried to infer the source code variable names back from IR but wasn't successful and struck for a while.

Then tried to approximate finding the difference between the programs in such a way that there won't be bulk of changes between the subsequent versions of the programs. Tried to restrict the changes to incremental single line additions or deletions or combinations. With the current state it won't be able to handle bunch of lines close together, it assumes that it could be only single line. This restriction need to be relaxed carefully.

The validation part is also pending implementation. Once the validation part come in the way results can be calibrated for analysis.

7. Related Reference work:

The from scratch naive implementation has been based on learnings from series of Taint Analysis work on Java from IBM [1][2]. TAJ [1] suggests comprehensive techniques such as thin Slicing, approach to bound the analysis based on user time requirements. ANDROMEDA [2] builds on top of that with ideas to do analysis on demand and reuse invalided previous version analysis results to incrementally recompute new version results. FLOWDROID [3] incorporates some of the learning from [2] like demand driven alias analysis into Android taint analysis setting successfully. Key contribution from FlowDroid is effective modeling of android lifecycle to increase the precision of the results. Our implementation effectively reiterates the effectiveness of the basic ideas from these works.

The Incremental segment of the implementation is inspired from ideas on incremental program analysis by restarting fix point computation suggested by Ghodssi [6]. There has been much work on improving the precision of incremental data flow analysis, Marlowe and Ryder's hybrid incremental technique uses an elimination-like flowgraph decomposition, a data flow solution factorization on component regions,and solution on regions by fixed point iteration. The algorithm handles all changes for the common distributive intraprocedural and interprocedural data flow problems and can deal with irreducibility[8][9]. Rosen has developed an elimination technique based on finding the flow cover of a flow graph; that technique is applicable in the presence of small changes to the flow graph [10]. There is scope to improve the implementation precision based on these more recent contribution.

8. Future Direction & Conclusion:

The current stage of the implementation is bit fragile since it is built around small code snippets with limited instruction types and it need to be sharpened to handle standard size programs with standard instruction types. It also need to be expanded to handle interprocedural control flows. Extensive recalibration of iteration saves and correctness of the analysis results need to be performed to substantiate the cost effectiveness.

Overall the incremental approach is a two-step process and we use a worklist which contains work to be done. In first phase we deal with the required data flow solution modifications of the immediately affected statements by removing the suspect values from old solution and initializing the worklist with only affected statements which need to be recomputed. During the second phase we propagate the immediate changes resulting from a program modification to all affected areas.

Another observation regarding the incremental approach is that, we have modeled our scenario as forward flow problem, the worklist contains the set of immediate successors of the affected statements, but we just need to go for immediate predecessors of the affected statements in the case of backward data flow problem. Only difference between these two directions is the way in which the worklist is constructed, basic idea remains the same.

Also from the observation it is clear that direction of data flow analysis doesn't matter much in computing the slice, the slice remains to be same. But the data flow information being collected differs and its size seems to depend on the branching of the seed information.

The conclusion that can be drawn from this case study is that the usual testing techniques are not enough for determining the complexities of the incremental approaches.In many cases the effect of a small program change can not be generalized based on limited examples. To precisely analyse the incremental approaches, empirical evidence is needed in the following direction :
(1) Expected types of program modifications.
(2) The effects of modification on the overall control flow structure of the programs.

9. Reference:

1. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. **TAJ**: effective taint analysis of web applications. DOI=http://dx.doi.org/10.1145/1543135.1542486
2. Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. **ANDROMEDA**: accurate and scalable security analysis of web applications. DOI=http://dx.doi.org/10.1007/978-3-642-37057-1_15
3. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. **FlowDroid**: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. DOI=http://dx.doi.org/10.1145/2666356.2594299
4. Mark Weiser. 1981. Program slicing. (ICSE '81). DOI=http://dl.acm.org/citation.cfm?id=802557
5. Frank Tip.1994. A Survey of Program Slicing Techniques. DOI=http://www.franktip.org/pubs/jpl1995.pdf
6. Vida Ghodssi. 1983. Incremental Analysis of Programs. Ph.D. Dissertation.
7. Ryder, B. G., T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: Examples and counterexamples DOI=http://dx.doi.org/10.1016/0167-6423(88)90061-5
8. Thomas J. Marlowe and Barbara G. Ryder. 1989. An efficient hybrid algorithm for incremental data flow analysis. (POPL '90). DOI=http://dx.doi.org/10.1145/96709.96728
9. Michael G. Burke and Barbara G. Ryder. 1990. A Critical Analysis of Incremental Iterative Data Flow Analysis Algorithms. DOI=10.1109/32.56098 http://dx.doi.org/10.1109/32.56098
10. Barry K. Rosen. 1977.High-level data flow analysis. DOI=http://dx.doi.org/10.1145/359842.359849