cFlow Source Code Analysis

Overall

`cFlow` can do static taint analysis on Java-based applications such as `Hadoop`. For each procedure, `cFlow` leverages `Soot` dataflow analysis framework to do intra-procedural taint analysis. When encountering a procedure call, `cFlow` uses summary to transfer the taints. When there is no new taint inserting into the summary, the whole analysis terminates and `cFlow` uses `DFS` to search for path from every source to its corresponding sinks.

- See Process below to check the procedure of running `cFlow`
- See Data Structures below to check the important data structures defined in `cFlow`
- See Limitations below to check the limitations of `cFlow`

Process

This is the main process of `cFlow`'s static taint analysis.

- In method `main()` in `src/main/java/Main.java`, it firstly parse the command line options, then it calls

  ```
  run(considered, use_spark, run_intra);
  ```

to run taint analysis.

- In method `run()` in `src/main/java/Main.java`, it calls

  ```
  driver.runInterTaintAnalysis(srcPaths, classPaths, use_spark);
  ```

  to run inter-procedural taint analysis.

- In method `runIntraTaintAnalysis()` in `src/main/java/taintAnalysis.java`, it initializes the `transformer` for inter-procedural analysis, which calls `internalTransform()` and does analysis.

  ```
  IntraAnalysisTransformer transformer = (IntraAnalysisTransformer)
  PackManager.v().getPack("jtp").get("jtp.taintanalysis").getTransformer();
  ```

- In method `internalTransform()` in `src/main/java/taintAnalysis/InterAnalysisTransformer.java`,

  Firstly, it calls `doAnalysis()` to do analysis.

  - In method `doAnalysis()` in `src/main/java/taintAnalysis/InterTaintAnalysis.java`,

    Firstly, it collects all methods and their corresponding method bodies.

    Secondly, for each body, `cFlow` constructs `analysis` of the class`TaintFlowAnalysis` and calls `analysis.doAnalysis()` to do intra-procedural analysis.

    - class `TaintFlowAnalysis` in `src/main/java/taintAnalysis/TaintFlowAnalysis.java` extends `ForwardFlowAnalysis` in `Soot` framework. It implements method `flowThrough()` to get the `out-set` based on `in-set` and the current node. For the current node, it calls different method to deal with different cases(such as current node represents an assign statement, invoke statement...)

    Thirdly, `cFlow` executes in a loop: For every method in a loop, `cFlow` leverages `summary` to get the `entryTaint` and does the analysis for that method again. If there is no new `entryTaint` added into `summary` for any method, then the loop terminates.

    Now the taint analysis is finished.

  Secondly, after the analysis is finished, it executes on `SourceSinkConnectionVisitor` with several threads to search for paths from sources to sinks.

  - class `SourceSinkConnectionVisitor` in `src/main/java/taintAnalysis/SourceSinkConnectionVisitor.java` implements the `dfs()` method to search the path from a `source` to all of its reachable `sinks`.

  At last, it print the paths from sources to sinks into the file. And the whole process is finished.

Data Structures

Taint abstraction

Description

Class `Taint` contains the implementation of the field-sensitive taint abstraction. It is defined in `Taint.java`.

Member

it has the fields of

- `plainValue`
- `field`
- `stmt`
- `method`
- `successors`
- `transferType`

,which describes the object(`plainValue` and `field`) and its context(`stmt` and `method`).

Method

- `getTransferredTaintFor`
    - Generates a taint object for a given pair of value and context.
    - Marks that it is the successor of its predecessor, and records the transfer type.
- `taints`:
    - Checks whether a value/expression/reference is tainted in this taint abstraction(By comparing the `plainValue` and `field` with the given parameters).

SourceSinkManager

Description

Class `SourceSinkManager` checks whether a statement is source or sink.

Method

- `isSource`

  It leverages the `isGetter` method of `ConfiureInterface`, which checks whether an `InvokeExpr` is a getter method.

- `isSink`

  It checks whether an invoke expression is an external library call(except the logger calls and some basic Java language calls)

TaintWrapper

Description

Contains the interface for defining library modeling rules and a default implementation using heuristic-based rules.

It has the list for methods of the type:`taintBoth`, `taintReturn`, `taintBase`, `kill` and `exclude`.

Actually, I don't know where it is used.

Method

- `TaintWrapper(String file)` as a constructor

  Given a file of `TaintWrapperSource.txt`, it appends the method name in that file into the corresponding lists according to the label.

- `genTaintsForMethodInternal()`

  Checks an invocation statement for black-box taint propagation. This method can allow the wrapper to propagate taints without analyzing the methods.(just like `SummaryEdge` in `IFDS` algorithm)

  Given the `in-set` of taint and the invoke statement, it gets the `base` of callee and the `return value` according to the invoke statement. Then, for each taint in the `in-set`, it checks whether it taints the `base` and the `parameter` of the called method. And later we can add new taint into the `gen-set` and `kill-set` according to different method types.

- `supportsCallee()`

  As long as the method is not from application class, the method of the callee is supported by the taint wrapper.

TaintFlowAnalysis

Description

The implementation of intraprocedural taint analysis.

It extends the framework of `ForwardFlowAnalysis` of `Soot`.

Member

- `Body body`: the method body of the procedure to analyze.
- `SootMethod method`: the method to analyze.
- `ISourceSinkManager sourceSinkManager`: to identity whether the statement is a `source`.
- `ITaintWrapper taintWrapper`: just like a summary. **???**
- `Taint entryTaint`: the taint at the entry point.
- `Map<SootMethod, Map<Taint, List<Set<Taint>>>> methodSummary`
- `Map<Taint, Taint> currTaintCache`
- `Map<Taint, Taint> methodTaintCache`
- `sources`
- `sinks`

Method

- `TaintFlowAnalysis()`

  it initializes`methodSummary`, `methodTaintCache` and `currMethodSummary` for current method.

- `flowThrough()`

  Uses flow function to calculate out-set according to in-set.

  - If `stmt` is assign statement(which may also contains invoke statement, calls `visitAssign()` to solve.
  - If `stmt` is invoke statement, calls `visitInvoke` to solve.
  - If `stmt` is return statement, calls `visitReturn()` to solve.
  - If `stmt` belongs to `sink`, calls `visitSink()` to solve.

- `visitAssign()`

  The process to calculate `out-set` according to `in-set` for assignment statement `l = r`.

  - Kills the original taints of the left operand `l`.
  - Generates the taints for the left operand `l` for the following cases:
    - The assignment `stmt` has invoke statement and that statement belongs to `source`.
    - For each taints `t` from `in-set`, if `t` taints right operand `r`, then we should add a new taint.
- `visitInvoke()`

  The process to calculate `out-set` according to `in-set` for invoke statement `retVal = base.m(a1,a2,...,an)`.

  - Try to use `taint wrapper` to get the `killSet` and `GenSet` of the called method. If so, we don't need to do anything else.
  - Collect all possible callees for this call site.
  - Get the base object `base` and return value `retVal` of the invocation in the caller.
  - Initialize `summary` for the called method
  - Compute `kill` and gather `summary` info for this invocation
    - if taint `t` in `in-set` taints `base`, then it will kill `t`, and calls `genCalleeEntryTaints()` to transfer the taint to `calleeThisLocal` of callee.
    - if taint `t` in `in-set` taints `parameter`, then it will kill `t`, and calls `genCalleeEntryTaints()` to transfer the taint to `calleeParam` of callee.
  - Compute `gen` from the gathered summary information. For `base`, `retVal` and parameters, calls `getTaintsFromInvokeSummary()` and adds that taint into `genSet`.
  - Kill the Intersection of all `killSet` and generate the Union of all `GenSet`.
- `visitReturn()`

  Visit the return statement `stmt`.

  for the taint `t` in `in-set`,

  - if `t` taints the base object, then it calls `getTransferredTaintFor` to transfer the taint at that return site and add that new taint to the summary.
  - if `t` taints the return value, then it does the same thing.
  - if `t` taints the object-type parameter, then it does the same thing.

  When we have done any of the change above, we will set `change` to `True`.
- `visitSink()`

  Visit the statement`stmt` which invokes a sink as the form `base.m(a1,a2,...,an)`.

  For each taint `t` from `in-set`, if `t` taints the base object

  or the parameter, than we will transfer the taint at that site and add the new taint into `sinks`.
- `genCalleeEntryTaints()`

  Generates taints at the entry of callee for `calleeVal`.
- `getTaintsFromInvokeSummary()`

Given the taint set `taints` from summary, the Value from the caller `callerVal` and the call statement `stmt`, we can call `getTransferredTaintFor()` to transfer that taint to `callerVal` and return them at last.

InterTaintAnalysis

Description

The implementation of interprocedural taint analysis

Member

- `sourceSinkManager`
- `taintWrapper`
- `sources`
- `sinks`
- `methodSummary`
- `methodTaintCache`

Method

- `doAnalysis()`

  Do interprocedural taint flow analysis by

  - collecting all `method`s of all application `class`es.
  - for each `method`, get its `body` and construct the analyzer `analysis` of `TaintFlowAnalysis`, and then call `analysis.doAnalysis()` to do intraprocedural taint analysis.
  - get `changed` from `analysis` to check whether we have reached a fix point. If so, stop iterating.

  The source code is like:

```
// Bootstrap
int iter = 1;
logger.info("iter {}", iter);
List<Body> bodyList = new ArrayList<>();
for (SootMethod sm : methodList) {
    Body b = sm.retrieveActiveBody();
    bodyList.add(b);
}
for (Body b : bodyList) {
    TaintFlowAnalysis analysis = new TaintFlowAnalysis(b, sourceSinkManager, Taint.getEmptyTaint(),
                    methodSummary, methodTaintCache, taintWrapper);
    analysis.doAnalysis();
    sources.addAll(analysis.getSources());
}
iter++;

boolean changed = true;
while (changed) {
    changed = false;
    logger.info("iter {}", iter);

    for (SootMethod sm : methodList) {
        Body b = sm.retrieveActiveBody();
        Set<Taint> entryTaints = new HashSet<>();
        entryTaints.addAll(methodSummary.get(sm).keySet());
        for (Taint entryTaint : entryTaints) {
            TaintFlowAnalysis analysis = new TaintFlowAnalysis(b,
sourceSinkManager, entryTaint,
                    methodSummary, methodTaintCache, taintWrapper);
            analysis.doAnalysis();
            sinks.addAll(analysis.getSinks());
            changed |= analysis.isChanged();
        }
    }

    iter++;
}

logger.info("Found {} sinks reached from {} sources", sinks.size(),
sources.size());
```

SourceSinkConnectionVisitor

Description

Searches the path from `source` to `sink`.

Member

- `Taint source`: the `source` that is searched from
- `long threshold`: maximum number of paths from `source` to `sinks`
- `Set<Taint> sinks`: stores the sinks that `source` can reach
- `List<List<Taint>> paths`: stores the paths from `source` to each sink in `sinks`.

Method

- `visit(Taint t)`

  Visit taint `t` and search the path from `t` to sinks.

  Initialize some stacks:

  - `intermediatePath` to record the taints along the path
  - `callerStack` to record the statement along the path
  - `MethodSet` to record the sets of methods
  - `VisitedStack` to record the set of taints that have been visited(to know whether there is repetitious search)

  Call `dfs()` to search the path.

- `dfs()`

  ```
  private void dfs(Taint t, Stack<Stmt> callerStack, Set<SootMethod> methodSet,
                   Stack<Set<Taint>> visitedStack, Set<Taint> sinks,
                   Stack<Taint> intermediatePath, List<List<Taint>> paths) {
  ```

  Use depth-first search to get the path from `t` to `sinks`.

Firstly, add `t` into `visitedStack` to avoid repetitious search and push `t` into `intermediatePath` to build the path.

Secondly, check the current statement of `t`:

- **The statement is a call statement**.

  For each `successor` in callee, push the new method, call statement into the stack and run `dfs` on the `successor`. After returning from `dfs`, pop those methods and call statements.

  (Actually, the source code hints that `cFlow` does not process direct/indirect recursive function call. If `methodSet` contains callee, then it will not continue searching)

- **The statement is a return statement**.

  For each `successor` in caller, if the caller has been explored, we can pop the method and call statement from the stack and then call `dfs` to explore the node in caller; If the caller has not been explored, we can append that caller into `methodSet` and then call `dfs` to explore.

- **The statement is a normal statement**

  Just calls `dfs` on `successor` to continue searching.

Thirdly, pop `t` from `intermediatePath` before exiting from this method.

Limitations

There are several points to improve

1. Prof. Xu T. and Zijie Lu said there exists indeterministic taint path. Yes, that really freaks me out because it is really hard to debug. I have found that it is the reason of the taint propagation graph itself, not about
2. `cFlow` cannot deal with direct or indirect recursive calls.
3. `cFlow` does not consider aliasing problem.
4. `cFlow` has a too simple heuristics to detect source and sink.