My basic understanding of cflow

fanwen USTC

This is the note of the paper of "cFlow:Flow-based Configuration Analysis Framework for Java Bytecode Based Cloud System" from UIUC Xlab

Abstract

1. modern software systems are customizable via configuration, and misconfiguration may cause system failure.
2. Existing attempt of static taint analysis is deeply coupled with upstream analyses and does not support taint propagation path recovery.
3. The author builds a static taint analysis framework to output the taint propagation paths for upstream configuration analyses.

Background

taint propagation:
Follow untrusted data and identify points where they are misused.

Sources: instrument methods that introduce input to set taint markers

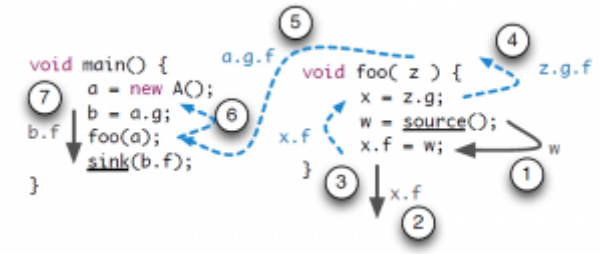
Sinks: instrument sensitive methods to check for taint marker before executing

from https://www.blackhat.com/presentations/bh-dc-08/Chess-West/Presentation/bh-dc-08-chess-west.pdf

domain specific analysis:
The domain means a type of software application?

FlowDroid:
FlowDroidFlowDroid is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications.

When we are trying to analyze the connection between source to the link, we need to analyze forward and backward.



for example, in the figure above,(3) is the reverse analysis.

(https://blogs.uni-paderborn.de/sse/tools/flowdroid/)

non-deterministic taint flow

- Still not sure what it means

Approach

Overview: What Cflow does

- **Preprocess:** java bytecode --(soot)--> Jimple IR & call graph
- **sink manager:** to identify the configuration of loading points and sink points
- **taint propagation engine:**
  - each tainted object is represented with a taint abstraction.
  - when there is an assignment s $\leftarrow$ t, we can create a new taint abstraction for s, and draw a directed edge from t to s.
  - Therefore, the output of the taint propogation phase is a directed taint propogation graph

Field sensitive taint abstraction

- How to model field access?
  - **field based approach:** treat fields as entities independent of base objects(overtainting)
  - **field sensitive approach:** dependent of the base objects
- we adopt `field sensitive approach`. we trade off precision for scalability and simplicity in implementation and limits the field access path length to 1. (which means there is no nested fields such as `b.f1.f2`)

taint policy

- **Taint sources:** After loading configuration into program variables, those variables are served as initial taints for the analysis and all the other variables are initialized as untainted.
- **Taint Sinks:** CFlow allows users to specify how to identify sink statements.
- **Taint propagation:**
  - forward dataflow analysis in **interprocedual control flow graph(ICFG)**
  - we only track explicit information flow(ignore implicit flow)

pointwise flow functions

Assignment:

- **receive** the incoming taints from the previous statement of the assignment
- **pass** the taints to the next statement of the assignment
- for the form $x.* = y.* \mid UnOp(y.*) \mid Binop(y.*, z.*) \mid Cast(y.*) \mid Instanceof(y.*)$
  - if incoming taint $T = x.*$, then the new taint set $T' = \emptyset$. (That's because the tainted object of $x.*$ is flushed by untainted object of $y.*$ or $z.*$)
  - if incoming taint $T = y.* \mid z.*$, then the new taint set $T' = T \cup x.*$.
  - else, $T' = T$.(just pass them on)

Call Flow Function:

- **receive** the incoming taints from the previous statement in the caller.
- **pass** the taints to the entry node of the caller.
- for the call form $o.m(a_0, a_1, \ldots, a_n)$:
  - if incoming taint $T = a_i.*$ in caller, then in callee entry, the taint set $T' = p_i.*$, where $p_i.*$ is the variable that stores the i-th parameter of the method m.
  - if incoming taint $T = o.*$, then in callee entry, the taint set $T' = this.*$, where $this$ is this object in the callee

    - if incoming taint $T = S$ where S is a static field, then the taint set in callee entry $T' = S$.
    - else, $T' = \emptyset$.

Return Flow Function:

- **receive** the incoming taint from the exit node of callee
- **pass** the taints to the next statement of the caller
- for the call form $o.m(a_0, a_1, \ldots, a_n)$
    - if incoming taint $T = p_i.*$ in callee, then the next taint set $T' = a_i.*$, where $p_i.*$ is the variable that stores the i-th parameter of the method m.
    - if incoming taint $T = this.*$, then the next taint set $T' = o.*$, where $this$ is this object in the callee
    - if incoming taint $T = S$ where S is a static field, then the next taint set $T' = S$.
    - else, $T' = \emptyset$.
- for the call form $x = o.m(a_0, a_1, \ldots, a_n)$
    - if incoming taint $T = x.*$ in callee, then the next taint set $T' = \emptyset$.
    - if incoming taint $T = r.*$ in callee where $r.*$ is the return object in the callee, then the next taint set $T' = x$(in the document, it is $x.*$, maybe that's because field length is assumed as 1)

Scalable Interprocedural Analysis

HOPE: We want to scale cflow to large software systems.

Summary-based Analysis

- Since many methods are called multiple times, we want to cache the separate analysis results for different calling contexts(passed taint abstractions)
- Each of the results of different taint abstrations should be analyzed.

Source-Independent Taint Abstraction

Question: Whether to store the source information in a taint abstraction.

- If we store, then two taint abstractions that taint the same objects from different source are different and should be propagated separatedly(**So we have multiple configurations, and they can init the same objects?**)
  (**Actually, I don't know why it hurts the reusability of nethod call summaries**)
- So we don't store currently, so redundant taint abstractions can be merged during propogation.
  (**what does redundent mean?**)
- However, there exists another graph reachability problem: **whether there exists a path between a pair of source and sink?**

Library Modeling

Problem: How to analyze library calls

It is not feasible to precisely directly analyze the libraries.

- `FlowDroid`: models common library calls using a few heuristic-based rules
- We also provide some rules to handle common strings,collection and math operations and a few common third-party libraries in Hadoop.

Call graph consideration

Problem: how to handle dynamic polymorphism and reflection in Java?

- Cflow uses `call graph`, but it will not build an accurate call graph due to the consideration of running time and memory consumption. **So cflow does not consider polymorphism or reflection by default.**

Path Reconstruction

Problem: How to generate a directed taint propagation path from source to sink?

- Cflow traverse the taint propagation graph and reconstruct the propagation graph.
- Unrealizable path: method `A` and method `B` call method `C`, but why the path from taint `A1` to method `C` and from taint `C` to method `B` is unrealizable?(unrealizable means can't be searched?)

- Use stack(or DFS) to solve.
- For cycles introduced by loops, **we keep track of all visited taint abstractions for each active method invocation(so we can unroll each loop only once)???**
- For recursive calls, we keep track of the whole calling stack and stops traversing when a recursive call is detected.

Evaluation

To test whether cFlow can

- track taint inter- and intra-procedurally?
- support field-sensitivity?
- overall performance on real world application

RQ1: Intra- and inter-procedural taint tracking

Here is the example in list1

In this example, variable `duration` in method `getShutdownTimeOut` is the source of taint.

Then, the taint is propagated to variabel `shutdownTimeout` in method `shutdownExecutor`

The taint tracking terminates prior to `LOG.error` or `awaitTermination` because these functions are external to `Hadoop`.

RQ2. Field-sensitivity in CFlow

Here is the example in list1

However, cFlow still has some limitations:

- does not support implicit information flow
- does not support tracking static and nested field access
- take loops and recursion too easily
- hard to make a fine-grain heuristic rules
- aliasing, dynamic polymorphism and reflection

infrastructure of Cflow

```
java bytecode
 |
 | (Soot)
 |
Jimple IR & call graph
 |
taint analysis & taint propogation
 |
directed taint propogation graph of taint abstractions
```

field-sensitive

field-based:

- treat fields as **independent** of the base objects they belong to.
- may cause over-tainting()

field-sensitive:

- treat fields as **dependent** of their base objects
- limit the field access length to 1 for simplicity

taint policy

taint sources:

- program variables at configuration loading site: initial taints
- other variables: untainted

taint sink

Any taint abstractions that flows into a sink statement will be reported later with the corresponding propagation path.

taint propagation

The author only tracks explicit information flow resulted from sequence of **assignments**

call graph function

get the taint from the caller and send those to callee
e.g. in o.m(a0,a1,...,ai), ai.* => pi
    in x.* = o.m(a0,a1,...,ai), set the return value as r.*, then r => x.*

scalable interprocedural analysis

summary-based analysis
To maintain context-sensitivity of the analysis, separate analysis results for different calling contexts(taint abstractions) are cached.

It suffice to compute a summary of the method for each possible incoming taint abstractions and union them.

Source-Independent taint abstraction

If we store the source information in a taint abstraction, we can easily know whether there is a taint propogation path between each pair of sources and links

Else, taint abstractions are independent of any information irrevelant to the actual taint abstraction.

Library modeling

It is not feasible to directly analyze the libraries.

The authors provide a set of predefined rules that handle the common string

call graph consideration

To handle dynamic polymorphism and reflection in JAVA, we need an accurate call graph
But cflow can use SPARK to generate a simple call graph.

path reconstruction

The main challenge for cFlow to reconstructs the propogation path is to maintain context-sensitivity during the traversal to avoid reconstructing unrealizable taint propogation paths

We can keep track of the call sites in a stack when traversing the taint propagation graph, to make it maintain context-sensitive property

for loops, we keep track of all visited taint abstractions for each active method invocation so we can unroll each loop only once

for recursive calls, we keep track of the whole calling stack and stops traversing when a recursive call is detected