Accurate Interprocedural Null Dereference Analysis for Java

Mangala Gowri Nanda and Saurabh Sinha

IBM India Research Lab

ICSE 2009

Introduction

- Null deference Analysis is useful for detecting bugs caused by interprocedural interaction.
- Some popular tools in this regard are JLINT,FINDBUGS and Escape/JAVA.
- *Loginov et al*(2008) presented a sound analysis based on abstract interpretation.
- The aforesaid analysis results in a large number of false positives.
- *Tomb et al*(2007) presented an execution-based analysis which serves more as a verification tool.
- The aforesaid analysis reduces the number of false positives while simultaneously reducing the number of true positives.
- Thus, the primary aim of this analysis is to reduce the number of false positives while reporting a sufficient number of different true positives.

Approach

- This analysis introduces a new tool, XYLEM, that performs the mentioned functions.
- This analysis is **interprocedural**, **context sensitive** and **flow sensitive.**
- This analysis is *inquisitive* in nature, that is, it answers a question which it asks about a null dereference at a particular point.
- This analysis produces a set of all paths ending at a statement S to show that a reference r may be null at this point via the shown paths,
- This analysis is *demand-driven* and **backward** in nature.
- This analysis utilises parameterisation to limit the extent of exploration.
- A natural consequence of such parameterisation is a time-accuracy tradeoff.

Salient Features

- The programming is modelled a simple but powerful abstract program model.
- Descent across the call graph occurs only as far as necessary. Thus,call depth will not limit the analysis.
- After each method is processed, a summary is stored on the cache. This helps retrieve information at a call site.
- A *null consistency check* is performed on the predicates at specific points when no more analysis can be done. This decides if the current path should be explored further of terminated.
- The analysis can be run in two modes:
  1. Demand Mode: To analyse a specific dereference at a specific program point. This is useful for online debugging.
  2. Batch Method: To analyse every dereference in a method. This is used for offline analysis and benchmarking purposes.

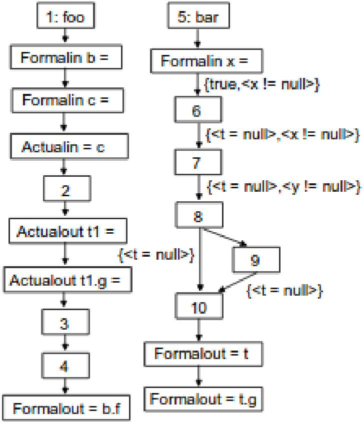
Some useful Definitions

We shall visit some useful terms which the analysis makes use of.

- Predicates are logical constraints which the program imposes to perform a consistency check, which shall be described later.
- A Control Flow Graph (CFG) is a graph consisting of nodes representing a set of statements and edges depicting control flow among them.
- An *escape-in* variable of a method M is a direct or indirect field of a variable, or a formal parameter, that is used before definition.
- An *escape-out* variable of a method M is the direct or indirect field of a formal parameter or return value of the method.
- For each escape-in variable, a *Formal-In* node is added in the CFG.
- For each escape-out variable, a *Formal-Out* node is added in the CFG.

An example

```
[1]  foo( B b, C c ) {
[2]    T t1 = bar( c );
[3]    r = c;
[4]    b.f = r.m();
     }
[5]  bar( C x ) {
[6]    T t = null;
[7]    C y = x;
[8]    if ( y == null )
[9]      t = new T();
[10]   t.g = 10;
[11]   return t;
     }
```



Predicates: In more detail

- To facilitate analysis, the program introduces a constraint at the specific program point using a predicate. We introduce $\langle\ r = null\ \rangle$ to check if dereference r can be null.
- Using backward analysis, we will check if the set of all predicates persisting at a program point cannot be explored are consistent, without which the path is deemed invalid.
- The grammar rules for predicates are shown below:

$$
\begin{array}{lll}
pred & ::= & \langle refVar\ \ refOp\ \ \texttt{null}\rangle \mid \langle refVar\ \ refOp\ \ refVar\rangle \mid \\
 & & \langle refVar\ \ refOp\ \ strConst\rangle \mid \langle boolVar\ \ refOp\ \ \texttt{true}\rangle \mid \\
 & & \langle intVar\ \ intOp\ \ intVar\rangle \mid \langle intVar\ \ intOp\ \ intConst\rangle \\
pred & ::= & \neg pred \\
refOp & ::= & = \\
intOp & ::= & < \mid \leq \mid =
\end{array}
$$

Predicates: Continued

- We start by defining a root predicate R_p and a root reference R_r.
- We introduce the predicate $\langle r = null \rangle$ at the specified program point to check if the Root reference r may be null at that point.
- We update the root predicate and the root reference as the program flows backward, resulting in a new predicate and root reference.
- We move backwards until we reach a statement which is outside the scope of analysis (this could be an entry function or an outside node).
- A consistency check performed at such a statement reveals if the chosen path is suitable or not.

State Transformations

- State transformations describe how the predicate changes as we backtrack. They are similar to transfer functions.
- The state transform generates a new predicate using the appropriate substitution.
- The following figure shows the state transformations used by the algorithm.

Statement	State transformation		
`x = y`	$\Gamma' = \Gamma[x/y]$		
`x = r.f`	$\Gamma' = \Gamma[x/r.f] \cup \{\langle r \neq \texttt{null} \rangle\}$		
`r.f = x`	$\Gamma' = \Gamma[r.f/x] \cup \{\langle r \neq \texttt{null} \rangle\}$		
`if x op y`	$\Gamma' = \Gamma \cup \{\langle x \; op \; y \rangle\}$		(true branch)
	$\Gamma' = \Gamma \cup \{\langle \neg(x \; op \; y) \rangle\}$		(false branch)
`x = y op z`	$\Gamma' = \Gamma \setminus \Gamma[x]$		
`x = new`	$\Gamma' = \Gamma \cup \{\langle x \neq \texttt{null} \rangle\}$		
`x = r.m()` (ext)	$\Gamma' = \Gamma[x/\texttt{null}] \cup \{\langle r \neq \texttt{null} \rangle\}$		(1)
		if $x = \mathcal{R}_r \wedge null(x)$	
	$\Gamma' = \Gamma \cup \{\langle x \neq \texttt{null} \rangle, \langle r \neq \texttt{null} \rangle\}$		(2)
		if $x = \mathcal{R}_r \wedge \neg null(x)$	
	$\Gamma' = \Gamma$	if $x \neq \mathcal{R}_r$	(3)
`x = r.m()` (app)	$\Gamma' = \sigma(r.m, \Gamma) \cup \{\langle r \neq \texttt{null} \rangle\}$		

Null Consistency Checks

- A set of predicates are not satisfiable if they contain conflicting terms. This forms the basis to decide which null flow paths are invalid.
- The algorithm makes two approximations in this regard:
  1. If a reference *r* is dereferenced along all paths or has to direct or indirect null check performed, it is assumed to be non null.
  2. If a reference *r* receives its value from outside the analysis scope and has an explicit null check performed in the code, it is treated as potentially null.
- The FINDBUGS tool uses this assumption but does not perform interprocedural and indirect null checks.

The Algorithm

algorithm `NullDerefAnalysis`
input s_d: dereference statement; r_d: variable dereferenced at s_d
output (path, state) pairs to dereference
declare $\mathcal{R}_r$: root reference; $\mathcal{R}_p$: root predicate; CS: call stack
$null(r)$ true if a null check is performed on reference r
$visited(s)$ set of states with which statement s is visited
$\mathcal{S}(M, \Gamma)$ summary information for method M for state Γ; consists of (path, state) pairs
begin
foreach method M in reverse topological order **do**
  foreach reference r that receives a value externally **do**
    if there is a direct or indirect null check on r **then**
      $null(r)$ = true
    else $null(r)$ = false
$\Gamma = \{\langle r_d = \texttt{null}\rangle, \langle \texttt{this} \neq \texttt{null}\rangle\}$; path $\rho = s_d$
return `analyzeMethod`(s_d, ρ, Γ)
end

function `analyzeMethod`
input s: statement to start analysis from; ρ: path to s; Γ: state at s
output T: (path, state) pairs from method entry to s
declare $worklist$: list of (s, ρ, Γ) triples; T, T_p: set of (ρ, Γ) pairs
$addWlist(s, \rho, \Gamma)$ adds (s, ρ, Γ) to $worklist$; updates $visited(s)$, $\mathcal{R}_r$
begin
$worklist = (s, \rho, \Gamma)$; $T = \emptyset$
while $worklist \neq \emptyset$ **do**
  remove (s, ρ, Γ) from $worklist$
  foreach predecessor s_p of s **do**
    if ($s_p \neq$ call/entry) $\vee$ (s_p calls an external method) **then**
      compute Γ' for the state transformation induced by s_p
      if (Γ' is consistent) $\wedge$ ($\Gamma' \notin visited(s_p)$) **then**
        $addWlist(s_p, \rho \cdot s_p, \Gamma')$
    else if s_p calls application method M **then**
      if $\mathcal{S}(M, \Gamma) = \emptyset$ **then** // no summary exists
        push M onto CS // analyze called method
        s_x = exit node of the CFG of M; Γ' = map Γ to s_x
        T_m = `analyzeMethod`(s_x, s_x, Γ'); pop CS
        foreach $(\rho_m, \Gamma_m) \in T_m$ **do**
          add (ρ_m, Γ_m) to $\mathcal{S}(M, \Gamma)$
      foreach $(\rho_m, \Gamma_m) \in \mathcal{S}(M, \Gamma)$ **do** Γ' = map Γ_m to s_p
        $addWlist(s_p, \rho \cdot \rho_m \cdot s_p, \Gamma')$
    else add (ρ, Γ) to T // s_p is entry
if $CS \neq \emptyset$ **then return** T
$T_p = \emptyset$
if this method is an entry method **then**
  foreach $(\rho, \Gamma) \in T$ **do**
    if $null(\mathcal{R}_r) \vee \mathcal{R}_p = \langle true \rangle$ **then** add (ρ, Γ) to T_p
else foreach call site s_c that calls this method **do**
  foreach $(\rho, \Gamma) \in T$ **do** Γ' = map Γ to s_c
    T_m = `analyzeMethod`$(s_c, \rho \cdot s_c, \Gamma')$; add T_m to T_p
return T_p
end

Observations

- The algorithm characterises each program point by a tuple (ρ,Γ) which denotes the path and the current state of predicates.
- The call stack *CS* declared allows processing of methods in a context sensitive fashion.
- The visited set of each statement also prevents reanalysis of the same state at a given program point.
- The summary information for each method lists all possible pairs (ρ,Γ) which may lead to a null reference at its exit basic block.
- Path validation by consistency check occurs at entry or calls to external methods, which are outside the current scope.
- It is not immediately clear how the indirect null check is made in the initialisation step.

Parameterisation to limit path exploration

- There are three principal parameters which can be introduced.
  1. *Traversal Time*: The analysis is aborted if the time taken to process a particular procedure call exceeds a predefined limit.
  2. *Number of predicates*: The analysis does not add any new predicates beyond a certain predefined limit, and continues as such.
  3. *Number of True paths*: The analysis does not explore any new paths in the current method, but extends currently computed paths into callers.
- We note that parameter 1 results in *aborted traversals*, while parameters 2 and 3 may result in *incomplete traversals*.
- For benchmarking tests, the parameters chosen were **2 seconds, 80 predcates and 7 true paths**.

Limitations of the Analysis

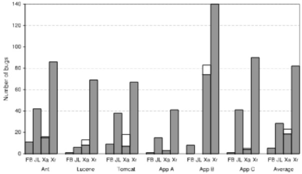
- The algorithm does not cover concepts of Java reflection such as Dynamic Class Loading.
- Concurrency is not handled by the algorithm.
- Integers and arithmetic operations themselves may result in false positives/
- The *instanceOf* operator in Java can be handled by introducing another null predicate.
- As a general mitigation to arithmetic limitations, dual predicates may be used, but they result in too many predicates to enforce correctness.

Evaluation and Performance

- The algorithm initially performs a points-to and escape analysis.
- The CFG and summary information are stored on disk, after which the algorithm itself is run.
- The analysis can be evaluated based on its accuracy, efficiency, paramterisation and the relevance of the detected bugs.

Accuracy

- The algorithm detects 16 times as many bugs as FINDBUGS and 3 times as many as JLINT.
- Bugs can be grouped into equivalence classes to study them more effectively.
- A comparison of Interprocedural and Intraprocedural analysis is also made below.

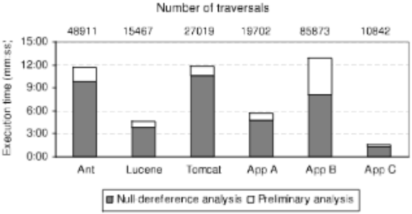



```
[506] public void execute() throws BuildException {
[509]   preconditions();
[514]   boolean useCtrlFile=(ctrlFl != null);
[615]   if (updIctrl) {
[626]     ctrlFl.getAbsolutePath());

[655] private void preconditions()
[671]   if (updIctrl == true && ctrlFl == null) {
[672]     throw new BuildException("...
```

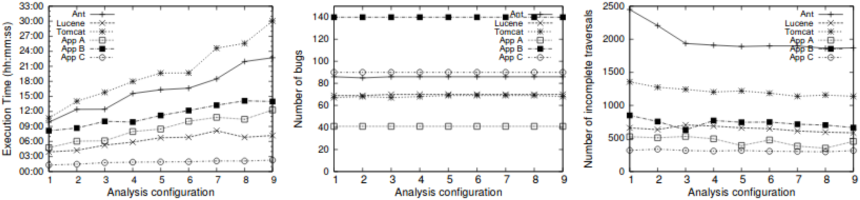
Efficiency

- Traversals are called either Complete, Partial or Aborted based on what happens while processing them.
- The chosen 2 second time limit causes less than 1 percent of traversals to abort.
- Almost 97 percent of traversals were complete.
- This analysis has been able to analyse almost 1,009,000 lines of code in 3.5 hours, which is a good improvement.



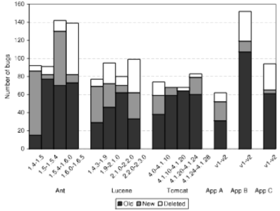
Effects of parameters

- Incomplete traversals reduce with an increase in all parameters.
- Execution time also increases with an increase in parameters.
- The number of bugs detected on the sample datasets remained almost constant.
- It is important to note that the traversal time is mentioned in terms of real time and not system time. Thus,**high CPU loads** will not cause unnecessary aborts.



Relevance of Detected Bugs

- XYLEM performs bug detection as opposed to sound verification.
- XYLEM has been used at IBM Labs and has been updated to fix around 31 percent of detected bugs designated as "must fix".
- The analysis proposed by *Tomb et al* is able to perform a verification to detect invalid type casts as well, but results in lower false and true positives.
- There are well developed tools for bottom-up and forwarding flowing analysis in C as well.



Summary

- The analysis contributes a useful tool, XYLEM that can be used for online and offline bug detection and empirical analysis.
- The analysis is context sensitive,flow sensitive and backward flowing, while remaining inquisitive in nature.
- Predicates and State transforms are used to determine if a path can lead to null dereference.
- Parameterisation of the algorithm by a bound on the execution time, number of predicates and number of paths introduces a time-accuracy tradeoff.
- The analysis is useful for future work such as effects of alteration of class hierarchy.

Future Work

- The effects of unintentional changes that alter detected bugs (such as changes in class hierarchy) can be studied in detail.
- Sound analysis on the lines of a tool like SALSA can be taken up.
- Prioritisation of True Positive Bugs can be taken up.
- A combination of static and dynamic tools to perform predictive analytics on the detected bugs is a possible domain to be explored.

Thank You!