Local Reachability Analysis for Better Debugging

Project Report

Horatiu Jula
Dependable Systems Laboratory
EPFL, Switzerland

June 22, 2009

Abstract

The goal of this project is to perform local reachability analysis, for Java programs, in order to find the conditions under which a given target program position can be reached from a start program position. Our tool can be used as a framework for learning the conditions under which a bug manifests in a Java application, once the bug is detected. The information we learn is useful for debugging Java programs, but also for improving failure immunity techniques like deadlock immunity [5].

1 Introduction

The goal of this project is to perform local reachability analysis for Java programs, in order to find the conditions under which a given target program position can be reached from a start program position. We developed a prototype (using the Soot static analysis framework) that performs local reachability analysis for Java bytecode, given a start program location and a target program location; more precisely, it finds the branches that have to be taken from the start position, to reach the target position.

Our tool can be used as a framework for learning the conditions under which a bug manifests in a Java application, once the bug is detected. The information we learn is useful for debugging Java programs, but also for improving failure immunity techniques like deadlock immunity [5]. A possible usage scenario for our system is the following. Once a bug is detected at runtime, the developer usually knows (from the bug report) the location in the source code where the bug manifested, then he/she feeds it to our system as the target program position. For the start program position, the programmer may input a source code location from which he/she suspects that the bug starts to manifest.

2 Problem Definition

Before diving into details, let us formally define the problem. Let P_{start} represent the start program position and P_{target} represent the target program position. Our goal is to infer the set $C_{start \rightarrow target}$ of conditions that have to hold between P_{start} and P_{target}, for P_{target} to be reachable from P_{start}. A condition is a tuple (s, v) indicating the branch that has to be taken at the conditional statement s for P_{target} to be reachable. For instance, the tuple $(s, true)$ means that, at the conditional statement s, we have to take the *true* branch.

The constraints our approach learns for the following Java code

```
synchronized(x) {//Pstart
  if (cond1) {
    ...
  }
  else {
    ...
  }
  if (cond2) {
    if (cond3) {
      ...
    }
    else {
      synchronized(y) {//Ptarget
        ...
      }
    }
```

```
  }
  else {
    ...
  }
}
```

are $C_{start \to target} = \{(cond_2, true), (cond_3, false)\}$.

Our analysis has two phases. In the first phase, we use static analysis to quickly find an approximation of $C_{start \to target}$. In the second phase, we refine $C_{start \to target}$ by using symbolic execution, in order to get a stronger set of constraints.

3 Static Analysis

We use static analysis to quickly find an approximation of $C_{start \to target}$. Given the CFG (control flow graph) of a program, for each conditional statement s between P_{start} and P_{target}, if P_{target} is not *statically* reachable from the *true/false* branch of s, we add the branch $(s, false/true)$ to $C_{start \to target}$. We say that a statement s_2 is statically reachable from a statement s_1 iff there is a path in the CFG from s_1 to s_2. Let s_{start}/s_{target} denote the statement corresponding to P_{start}/P_{target}, and m_{start}/m_{target} the method containing P_{start}/P_{target}. Let $succ^*(s/m)$ be the transitive closure of the statements/methods reachable from the statement/method s/m (including s/m). For a statement s, $succ^*(s)$ is computed intraprocedurally, within the method containing s.

The static analysis has two phases. First, we detect the set $M_{start \to target}$ of the methods statically reachable between P_{start} and P_{target}. Formally, we have $M_{start \to target} = \{m | m_{target} \in succ^*(m) \wedge m \in \bigcup_{m_0 \in M_{start}} succ^*(m_0)\}$, where $M_{start} = \{m_{start}\} \cup \{m | \exists s \in succ^*(s_{start}) \,.\, s.calls(m)\}$; $s.calls(m)$ means that statement s calls method m (or *may* call m, if we consider inheritence and method overloading). In the second phase, we detect all branching statements s (for simplicity, we refer only to *if* statements) statically reachable from P_{start}, from which P_{target} is reachable through only one branch of s. Let $s_{true/false}$ denote the first statement on the *true/false* branch of s, $method(s)$ the method to which statement s belongs, and $firstStm(m)$ the first statement of method m. Formally, we define $C_{start \to target} = \{(s,v) | method(s) \in M_{start \to target} \wedge Reach^{static}_{start \to target}(s_v) \wedge \neg Reach^{static}_{start \to target}(s_{\neg v})\}$, where $Reach^{static}_{start \to target}(s) \equiv (method(s) = m_{start} \Rightarrow s \in succ^*(s_{start})) \wedge (method(s) \neq m_{start} \Rightarrow s \in succ^*(firstStm(method(s)))) \wedge ((s_{target} \in succ^*(s)) \vee (\exists s' \in succ^*(s), m \in M_{start \to target} \,.\, s'.calls(m)))$.

3.1 Implementation

The static analysis implements the above declarative algorithms. The challenges we encountered were in handling inheritence and in dynamically building the CFG.

We build the CFG incrementally (using a fix point algorithm), while computing $M_{start \to target}$. To handle dynamic method lookups, we consider that a call statement $x.m(...)$ can invoke any method $T.m(...)$, where T is derived from (or equal to) the static type of x.

3.2 Evaluation

We evaluated our implementation on the *hsqldb* library, used in Limewire. We ran our tool to find what branches need to be taken to get to the statement at line 739 in *HsqlTimer* class, from the statement at line 511 from the same class. Our tool accurately reported $M_{start \to target} = \{$ *org.hsqldb.lib.HsqlTimer$TaskQueue.peekTask*(), *org.hsqldb.lib.HsqlTimer.nextTask*(), *org.hsqldb.lib.HsqlTimer.isCancelled*()$\}$, and $C_{start \to target} = \{$ $(super.heap[0] != null \ @ \ 901, true)$, $(task != null \ @ \ 514, true)$, $(wait > 0 \ @ \ 524, true)$, $(!this.isShutdown || Thread.interrupted() \ @ \ 502, true)\}$. The third condition ($(wait > 0 \ @ \ 524, true)$) is not trivial; for some time we thought it is a false positive caused by a bug in our implementation.

The static analysis we use does not have false positives when it returns that P_{target} is unreachable—it reports that P_{target} is unreachable iff there is no path to P_{target}. However, it has false positives when it returns that P_{target} is reachable. We aim at reducing these false positives by symbolically executing the program.

4 Symbolic Execution

We refine the constraints produced by the static analysis, by using symbolic execution, in order to get a stronger set of constraints. We generate a symbolic representation of each execution path, as a set of constraints. The actual symbolic execution consists of solving these sets of constraints; we use the Z3 SMT solver to solve them.

We describe now the symbolic execution in more detail. In the symbolic program, each statement s is labelled with a fresh boolean variable l_s denoting that s has to be executed. We use the prime notation x' to denote that the version of x is incremented. By $next(s)$, we denoted the statement following s. By $\rho(s)$, we denote the constraints accumulated during the symbolic execution of the statement/expression s.

Checking whether P_{target} is reachable from P_{start} is simply checking the satisfiability of $\rho(firstStm(m_{start})) \wedge l_{firstStm(m_{start})} \wedge l_{s_{start}} \wedge l_{s_{target}}$.

Formally, the symbolic execution of an assignment statement $s \equiv x = e$ is $\rho(s) \equiv (l_s \Rightarrow (x' = \rho(e) \wedge l'_{next(s)})) \wedge \rho(next(s))$. Since we use a unique label l_s for each statement s and we increment the version of l_s every time we reencounter s, we do not need a variable for the program counter.

When we meet a conditional statement $s \equiv if\ cond\ goto\ s_{true}\ else\ goto\ s_{false}$, we explore both branches of s, by forking a new symbolic execution engine, that runs in parallel with the current engine. The symbolic execution of s, when the *true* branch is taken, is $\rho(s) \equiv (l_s \Rightarrow (\rho(cond) \wedge l'_{s_{true}})) \wedge \rho(s_{true})$. The symbolic execution of s, when the *false* branch is taken, is $\rho(s) \equiv (l_s \Rightarrow (\neg\rho(cond) \wedge l'_{s_{false}})) \wedge \rho(s_{false})$. When symbolically executing conditional statements, we reuse the results of the static analysis—every time we meet a conditional statement s, for which $(s, false/true) \in C_{start \rightarrow target}$, we know that it is pointless to explore the branch $(s, true/false)$, so we only explore the branch $(s, false/true)$.

We symbolically execute rhs array accesses $a[e]$, using McCarthy's theory of arrays—we have that $\rho(a[e]) \equiv select(a, \rho(e))$. We model objects as arrays. Therefore, rhs accesses to object fields are symbolically executed as array accesses. For each field f, we have a unique id $fieldId(f)$. We have that $\rho(x.f) \equiv select(x, fieldId(f))$. We symbolically execute a lhs array access in an assignment $s \equiv a[e] = v$, as in $\rho(s) \equiv (l_s \Rightarrow (a' = store(a, \rho(e), \rho(v)) \wedge l'_{next(s)})) \wedge \rho(next(s))$. We symbolically execute a lhs object field access in an assignment $s \equiv x.f = v$, as in $\rho(s) \equiv (l_s \Rightarrow (x' = store(x, fieldId(f), \rho(v)) \wedge l'_{next(s)})) \wedge \rho(next(s))$.

Remember that each symbolic execution instance executes only one path. When a symbolic execution instance I reaches P_{target}, we check the satisfiability of the formula $\rho(firstStm(m_{start})) \wedge l_{firstStm(m_{start})} \wedge l_{s_{start}} \wedge l_{s_{target}}$ accumulated in instance I. If the aforementioned formula is satisfiable, then the current execution path ending in P_{target} is feasible, and we say that P_{target} is reachable in instance I. If a symbolic execution instance terminates, and it ends in other position than P_{target}, we say that P_{target} is not reachable in that instance. We say that P_{target} is reachable from a statement s iff there is a symbolic execution instance within $\rho(s)$, in which P_{target} is reachable. We say that P_{target} is unreachable from a statement s iff in every symbolic execution instance within $\rho(s)$, P_{target} is unreachable.

We refine the results returned by the static analysis as follows. For each conditional statement $s \equiv if\ cond\ goto\ s_{true}\ else\ goto\ s_{false}$ encountered during the symbolic execution, for which P_{target} is unreachable (only) from s_{true}/s_{false}, we add $(s, false/true)$ to $C_{start \rightarrow target}$.

4.1 Implementation

One of the main challenges is to efficiently implement the forking. For the moment, we implemented a simple forking mechanism, that clones the current symbolic execution engine (it performs deep copy of the versioning data, and shallow copy for the rest).

An advantage of our design is that the loop/method unrolling is performed automatically by the forking mechanism; we do not have to know a priori how many times a loop/method has to be unrolled.

Another important advantage is that we do not have to perform any merging of the execution paths a priori. However, the path merging is hard to do on-the-fly, during the symbolic execution. This is the price we have to pay for explicitely exploring the execution paths—we may redundantly explore many states. This redundancy may lead to a blowup of the memory consumption. We can mitigate that by using a thread pool—if the thread pool is full when forking a new symbolic execution engine, we run the new engine in the current thread. However, if we use a thread pool, the impact of redundant explorations turns into a longer execution time.

We ensure the consistency of the versioning during method unrolling by (1) maintaining the call stack, (2) associating a fresh id with each method call the we symbolically execute, and (3) maintaining a $(variable, version, callId)$ triple to refer to a local variable.

Keeping track of aliasing was straightforward, since a symbolic execution instance explores only one execution path. We maintain in the *ref* map the aliasing information for each variable x of a reference type (i.e., array/object type). When we encounter an assignment

$x = y$, where x, y are references to arrays/objects, we have that $ref(x) = ref(y)$. For a reference x to a freshly allocated array/object, we have that $ref(x) = x$. This way, $ref(x)$ always points to the original array/object. A field invocation $e \equiv x.f$ is symbolically executed as $\rho(e) \equiv select(ref(x), fieldId(f))$. An assignment $s \equiv x.f = e$ is symbolically executed as $\rho(s) \equiv ref(x)' = store(ref(x), fieldId(f), \rho(e))$.

4.2 Evaluation

We evaluated our symbolic execution approach on simple Java programs; we managed to improve the results obtained by the static analysis. For the following Java code

```
void f(int x) {
    x = 1;//Pstart
    x = this.g(x);
    if (x == 1)
        x--;//Ptarget
}

int g(int x) {
    x++;
    return x;
}
```

the static analysis returns $C_{start \to target} = \{(x == 1, true)\}$; it wrongly assumes that P_{target} is reachable through the branch $(x == 1, true)$. The symbolic execution module easily figures out that x is incremented after the $x = this.g(x)$ assignment, and therefore the branch $(x == 1, true)$ cannot be reached.

5 Related Work

Previous work exists in achieving immunity against exploits like buffer overrun. Systems like Vigilante [4], Bouncer [3], [2] or [1] use static analysis and/or symbolic execution to learn filters from existing exploits. Bouncer [3] also tries to learn new exploits that exercise the same vulnerability as the original exploit. The filters these approaches deduce refer only to the inputs of functions that process messages, and they are applied only at the beginning of such functions. Moreover, these approaches only address exploits like buffer overruns.

We want to develop a technique similar to the above ones, but applicable in a larger context, to help addressing a larger range of problems. We want our approach to be able to infer meaningful predicates at arbitrary program positions; the predicates are expressed in terms of program variables visible at those positions. We will apply our approach to Dimmunix, to learn new deadlock signatures from an existing deadlock signature, and to learn conditions that have to hold (after acquiring a lock) for a deadlock location to be reachable.

In terms of handling loops in symbolic execution, only [1] describes a technique of handling loops, based on computing fix points. They first identify the induction variables in each loop. Then, they infer loop invariants in terms of induction variables and loop conditions. However, it is not clear how they infer the number of iterations starting from loop invariants. We adopt a simple fork-based technique, for which we do not need to estimate the size of the loops/recursive calls a priori.

6 Discussion and Future Work

There are still many challenges that remained untackled.

We need to handle real applications in the symbolic execution engine. We managed to do that for our static analysis module; but it is considerably harder for the symbolic execution, because we actually need to simulate the program's execution.

Another challenge is soundness. Again, for the static analysis it was easy to return sound "P_{target} unreachable" results; but for the symbolic execution that is virtually impossible. There are factors like data races and lack of context information (e.g., aliasing information, program state), that impede the symbolic execution—failing to deal with them makes the symbolic execution unsound. The main handicap of our approach is that we start the symbolic execution from an arbitrary program location P_{start}; we do not have any information about the execution of the application prior to P_{start}, and therefore we do not have any context information at P_{start}. Due to this fact, we will drop any soundness requirement for the symbolic execution. Unfortunately, that is not good for failure immunity systems like Dimmunix [5] (for which we actually want to apply this tool); Dimmunix needs sound unreachability results. However, for debugging purposes, it is not a problem if sometimes the results are unsound.

Some clear advantages of our approach are its flexibility and scalability. A programmer may input any program locations for the start position and the target po-

sition. Moreover, if the two locations are close to each other (e.g., they belong to the same class), our approach scales well—the size of the application matters very little in that case. For the existing reachability analysis techniques, the start position is usually the beginning of the program; this impedes both the flexibility and the scalability.

An important drawback of our approach is that the memory model only handles objects/arrays with fields/elements of primitive types. To handle arrays/objects with elements/fields of reference types, we need to change our memory model. We can model the memory as an array whose elements are either primitive types or arrays. Having nested reference types also changes the aliasing tracking. For instance, for an assignment $x.f = y.f$, where f is a reference type, we have that $ref(x, f) = ref(y, f)$; $ref(x, f)$ represents the object/array to which $x.f$ points.

References

[1] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, 2006.

[2] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *13th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[3] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *ACM Symp. on Operating Systems Principles*, 2007.

[4] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *ACM Symp. on Operating Systems Principles*, 2005.

[5] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *8th Symp. on Operating Systems Design and Implementation*, 2008.