Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks

Jürgen Graf[1], Martin Hecker[1], Martin Mohr[1], and Benedikt Nordhoff[2]

[1] Karlsruhe Institute of Technology {graf|martin.hecker|martin.mohr}@kit.edu
[2] University of Münster b.n@wwu.de

Abstract. We combine the static analysis techniques of Program Dependence Graphs (PDG) and Dynamic Pushdown Networks (DPN) to improve the precision of interference analysis for multithreaded Java programs. PDGs soundly approximate possible dependence between program points in sequential programs through data and control dependence edges. In a concurrent setting a third category of so-called interference edges captures the potential interferences between memory accesses in different threads. DPNs model concurrent programs with recursive procedures, dynamic thread creation and nested locking. We use a lock-sensitive analysis based on DPNs to remove spurious interference edges, and apply the results to information flow control.

1 Information Flow Control for Multithreaded Java Programs

Information flow control (IFC) analyses check whether information about a programs (secret) input can possibly flow to public output, e.g. if a secret value is printed to console. A program is called *non-interferent*, iff it does not leak secret information. Non-interference of a given program can be verified with a sound static analysis that detects possible information flow through dependencies and interference between program statements using PDGs [1]. If the analysis detects no illegal information flow, it is guaranteed that during execution of the program, an attacker observing the public output will learn nothing about the secret input. However, depending on the precision of the analysis, there may be false alarms, because the analysis reports spurious information flow that is impossible during an actual execution of the program. Thus a main goal of static non-interference analyses is to minimize the number of false alarms, by improving analysis precision.

In the following example we show which false alarms may arise when a multithreaded Java program is analyzed for non-interference. The program in figure 1 does not leak information about the secret value `secret` to a public visible output `println` and should be considered non-interferent: It contains two threads, the main thread t_0 and an instance of `MyThread` t_1. Output to a public visible channel only occurs through the two `println` statements in t_0. At a first glance it may seem possible that the value of `secret` is leaked, because t_1

copies its value to shared variable x, but this is not the case. The first println statement cannot leak the secret, because it is executed before t_1 starts. Therefore x cannot contain the secret value at this time. We call an analysis that can detect the absence of this leak *invocation-sensitive*. The second println statement does also not leak the secret, because of the synchronization through lock l. The lock l is acquired in t_0 before t_1 is started. Due to this the write operation in t_1 can only be executed after t_0 releases l again which only happens after the second println. An *invocation-* and *lock-sensitive* analysis is able to detect this.

```java
class MyThread extends Thread {
  private Object l;
  private int secret = 42;
  private int x = 0;

  public static void main() {
    Object l = new Object();
    MyThread t1 = new MyThread(l);
    System.out.println(t1.x);
    synchronized (l) {
      t1.start();
      System.out.println(t1.x);
    }
  }
  public MyThread(Object l) {
    this.l = l;
  }

  public void run() {
    synchronized (l) {
      x = secret;
    }
  }
}
```

Fig. 1: A non-interferent multithreaded Java program.

Therefore in practice even the very precise invocation-sensitive PDG-based IFC analysis [1, 2] can only remove the first false alarm and does raise the second one. We were able to remove this false alarm and proof the program non-interferent by incorporating the results of a DPN-based analysis [3–6].

2 Concurrent Program Dependence Graphs

A PDG is a graph that captures the dependencies between statements in a program. Each node corresponds to a statement and potential dependencies between statements are represented by edges. In a sequential program, two statements s_1 and s_2 may either be *data dependent*, when a s_2 uses a value that s_1 has produced, or *control dependent*, when the outcome of s_1 decides if s_2 will be executed. These dependencies are in general a sound overapproximation of all dependencies that may occur during program execution. So whenever two statements are not connected in the PDG, they will never depend on each other during runtime and there is no information flow between them. Figure 2 shows the PDG of the example program with all data and control dependencies that may occur.

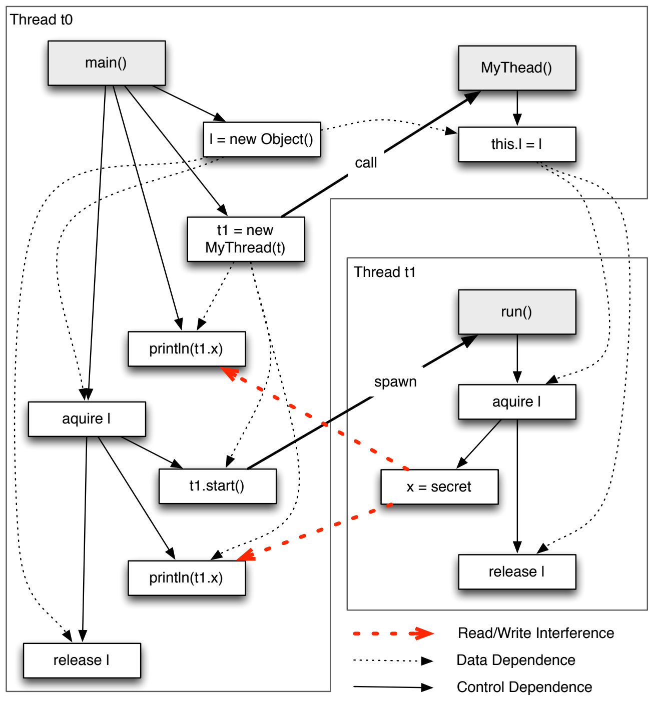

Fig. 2: The concurrent PDG of the program in Figure 1 that contains two spurious interference edges

For concurrent programs, control and data dependencies do not suffice, because they do not capture interference between different threads. Therefore the concurrent PDG contains additional *interference dependence* edges. A write and a read statement from two different threads are connected with an interference dependency, iff the value written may be read by the read statement. The concurrent PDG in Figure 2 shows two interference dependencies between both `println` statements and the statement that writes the value of variable `x`. As previously mentioned, both of these interferences are spurious and can be removed.

Giffhorn [2] proposes an invocation-sensitive but lock-insensitive may-happen-in-parallel (MHP) analysis that keeps track of thread creation and invocation through a dataflow analysis on the control flow graph. This algorithm is able to detect that the first `println` statement may not happen in parallel with the write operation on `x`, because the second thread t_1 has not been started at this time. The second interference dependence however is not removed, because the algorithm does not consider locking.

3 Dynamic Pushdown Networks

In order to achieve lock-sensitivity, we model concurrent Java programs using Dynamic Pushdown Networks (DPN) [3–6]. DPNs can precisely model concurrent programs with dynamic thread creation, unbounded recursion, synchronization via well-nested locks and finite abstractions of thread-local and procedure-local state. *Execution trees* [5] allow us to represent all the DPN's lock-sensitive executions using tree-automata. This allows to check for reachability of configurations with tree-regular properties e.g. calculating MHP information. Note that in the presence of locking MHP is not a sound criteria to remove interference. In fact Giffhorn [2] defines that two statements may-happen-in-parallel iff there exists two executions in which they are executed in opposite order. Recent extensions of DPN-analysis [6] allow to iterate the execution tree based technique and check whether critical configurations can be reached from other configurations while retaining a tree-regular property. In particular, we can check whether there exists an execution that executes the write to `x` first, followed by one of the `println` statements without an intervening killing of `x`. Since this is not the case, the DPN-based analysis will remove the spurious second interference edge.

4 Implementation and Future Work

We have integrated the DPN based interference Analysis in the tool *Joana*[7]. Joana, based on the Wala framework, implements a flow-, object-, context-sensitive IFC Analysis based on PDGs. Within the RS^3 priority program, we plan to integrate further analysis technique in order to further improve the tools precision. In particular, we want to improve on the lock-detection analysis, which in Java is difficult since any object can function as a lock. We plan to use path conditions and linear invariants to detect further spurious information flow. Analyses based on PDGs are typically whole-program analyses. In order to deal with software consisting of several components, we implement modular PDGs and methods for plugin-time analysis.

Acknowledgments. This work was funded by the DFG under the project IFC for Mobile Components in the priority program RS^3 (SPP 1496).

References

1. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security **8**(6) (December 2009) 399–422 Supersedes ISSSE and ISoLA 2006.
2. Giffhorn, D.: Slicing of Concurrent Programs and its Application to Information Flow Control. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)

3. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR 2005. Volume 3653 of LNCS. Springer (2005)
4. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: Proceedings of the 21st International Conference on Computer Aided Verification. CAV '09, Berlin, Heidelberg, Springer-Verlag (2009) 525–539
5. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation. VMCAI'11, Berlin, Heidelberg, Springer-Verlag (2011) 199–213
6. Nordhoff, B., Lammich, P., Müller-Olm, M.: Iterable forward reachability analysis of Monitor-DPNs. Submitted for publication (2012)
7. Programming Paradigms Group, KIT: Joana (Java Object-sensitive ANAlysis) tool. http://joana.ipd.kit.edu