# Implementation Details of PLS Detector

```
1.  void logDestroyNode(NodeContext nc) {
2.      NodeStatus ns = checkDestroyStatus(nc);
3.      if (ns == NodeStatus.WAIT_APP) {                          If condition
4.          LOGGER.debug(String.format(                            Then block
5.                  " timeout:%4ds", getTimeoutInSec(nc)));
6.      }
7.      Iterator<App> it = nc.getApps().iterator();
8.      StringBuilder sb = new StringBuilder();
9.      while (it.hasNext()) {                                    Loop header
10.         App app = it.next();                                   Loop body
11.         sb.append(String.format(
12.                 " appID: %s", app.getId()));
13.     }
14.     LOGGER.debug("Destroy node: " + sb.toString());
15. }
```

Fig. 1: An example of logging call in Hadoop

We implement our algorithm on top of Soot [1], which is a widely-used static analysis framework. The first challenge in our algorithm is how to strike a balance between scalability and performance of side-effect analysis, which further depends on the used pointer analysis. Basically, the existing pointer analysis falls into two categories: context-sensitive and context-insensitive [2]. The former has difficulties being scalable, and the latter is known to be imprecise (i.e., superfluous pointer information). To solve this problem, we build our side-effect analysis based on lazy access path resolution [3]. The side-effects of each method are represented as access paths on formal parameters (including $this$ object) and propagated from the callees to the callers in a bottom-up manner. These side-effects will be lazily resolved to the accessed locations with the help of inclusion-based context-insensitive pointer analysis (e.g., Soot Spark [4]) until they can not be mapped to access paths in the caller. By doing so, we can improve the precision of side-effect computation because access paths in the caller can often be resolved to smaller abstract location sets in inclusion-based context-insensitive pointer analysis.

Despite the adoption of lazy access path resolution, we find that there still are many superfluous side-effects propagated to the caller. It is because there are many superfluous edges in the call graph due to the imprecision of context-insensitive pointer analysis. To avoid this problem, we drop all side-effects that can not be mapped to access paths in the caller during the propagation. By doing so, we may ignore certain side-effects on global logging variables and miss the associated PLSs. However, we observe that it is extremely rare in practice.

The data-dependence information is collected by adapting a standard reaching definitions analysis. We take the side-effect into consideration when retrieving the variable definitions and variable uses for each statement. The control-dependence is computed by traversing the post-dominator tree of each method. To collect the initial set of $nonPLS$, we implement a simplified escape analysis [5] to identify the statements whose variable definitions can escape from the current method.

Next, we describe some key points in the implementations of detectors based on data-dependence and control-dependence. As we mentioned above, data-dependence-based detection is recursive. In our implementation, we maintain a set to store all methods under analysis, and if we are going to visit a method already in the set, we return $True$ immediately to avoid infinite recursion. Generally, control-dependence-based detection is straightforward. As illustrated in Figure 1, the conditional statement in line 3 can be identified as a PLS, because all its guarded statements (i.e., the then block in line 4-5) are PLSs. However, the simple rule fails to handle loop structures because of the cycle dependence. Taking the for loop in Figure 1 as an example, according to the data-dependence-based detection, the statement in line 10 is a PLS if the statements in line 9 (affected by side-effect) and 11 (affected by main-effect) are PLSs. However, according to the control-dependence-based detection, the statement in line 9 is a PLS if its guarded statements (i.e., the loop body in line 10-11) are PLSs. To address the cycle dependence, we identify the loop variables (e.g., $i$ and $it$) in loop headers, and remove the loop-variable modification statements (e.g., $i$++ and $it.hasNext()$) in the loop body.

## References


[1] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, “The soot framework for java program analysis: a retrospective,” in *Cetus Users and Compiler Infastructure Workshop*, vol. 15, 2011, p. 35.

[2] M. Hind, “Pointer analysis: Haven’t we solved this problem yet?” in *Proceedings of 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.

[3] J. Qian, Y. Zhou, and B. Xu, “Improving Side-Effect Analysis with Lazy Access Path Resolving,” in *Proceedings of 2009 IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 35–44.

[4] O. Lhoták and L. Hendren, “Scaling java points-to analysis using s park,” in *International Conference on Compiler Construction*, 2003, pp. 153–169.

[5] D. Gay and B. Steensgaard, “Fast escape analysis and stack allocation for object-based programs,” in *Proceedings of 2000 International Conference on Compiler Construction*, 2000, pp. 82–93.