PCA: Memory Leak Detection using Partial Call-Path Analysis

Wen Li
Washington State University
li.wen@wsu.edu

Haipeng Cai
Washington State University
haipeng.cai@wsu.edu

Yulei Sui
The University of Technology Sydney
yulei.sui@uts.edu.au

David Manz
Pacific Northwest National Laboratory
David.Manz@pnnl.gov

ABSTRACT

Data dependence analysis underlies various applications in software quality assurance, yet existing frameworks/tools for this analysis commonly suffer scalability challenges. We present PCA, a static interprocedural data dependence analyzer for real-world C programs. PCA performs interprocedural points-to and data-flow analyses with a lightweight design. Most of all, it features a partial call-path (PCA) analysis that consists of optimization options to further speed up data dependence computation. As an example application of it, PCA readily supports memory leak detection, for which it helps achieve close or better performance and precision relative to the same application based on a state-of-the-art value flow analysis. In particular, it found four more memory leaks in an industry-scale system which have been fixed by the developers. Through the data dependence it computes, PCA can enable other applications (e.g., impact analysis and taint analysis).

A demo video for PCA can be found here and tool package here.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Theory of computation → Program analysis**;

KEYWORDS

LLVM, static analysis, data dependence, efficiency, scalability

ACM Reference Format:
Wen Li, Haipeng Cai, Yulei Sui, and David Manz. 2020. PCA: Memory Leak Detection using Partial Call-Path Analysis. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3368089.3417923

1 INTRODUCTION

With their growing size and complexity, modern software systems are increasingly challenging to check against desired properties (e.g., correctness and security). One main approach to this task is to reason about program behaviors with respect to how data are computed and accessed (e.g., in terms of data dependence) in the program [5, 6]. This approach, as an underlying technique, supports a range of applications in software quality assurance, such as bug detection [7, 19] and security vulnerability discovery [16, 17].

There are tools that implement this underlying technique or facilitate such implementations. As a recent work, PhASAR [15] provides a generic framework for static interprocedural data flow analysis. Yet rather than a specific tool, it offers building blocks of static analyzers (e.g., call graphs, points-to information) via a set of APIs that tool developers may use to develop their own analyzers (e.g., for computing data dependence). Moreover, based on the IFDS/IDE algorithmic framework [14], PhASAR targets highly precise data flow analyses. For large, complex software systems, efficiency barriers due to its heavyweight nature may not be well paid off by the level of precision it offers—other cost-effectiveness tradeoffs might be more desirable to users [8].

Analyzers of value flow, such as SVF [17] and Pinpoint [16], provide a potential alternative. While a conventional value flow analysis would be built on data dependence analysis hence provide data dependence, these state-of-the-art tools focus on *sparse* value flow analysis. They focuses on precise computation of value flow information necessary for applications that need it, primarily applicable to source-sink problems such as detecting memory defects. As a result, these frameworks/tools may not provide the best cost-effectiveness tradeoffs for particular dependence-based applications that do not need the information. Commercial tools like CodeSonar [1] and Coverity [2] do so. Yet they are closed-source hence may not be sufficient for research purposes.

Therefore, in this paper, we develop PCA, an open-source static interprocedural data dependence analysis tool that scales to industry-scale C software with a practical cost-effectiveness tradeoff that complements to what existing alternatives offer (i..e, primarily focusing on precision or not providing desired balance between analysis cost and effectiveness [15–17]), particularly for solving source-sink problems (e.g., memory leak detection). PCA is built on top of LLVM [13]. It uses the LLVM gold plugin to generate the intermediate representation (IR) code for each module of a given input program and computes interprocedural points-to sets using Andersen's algorithm [4] and LLVM's basic, built-in analysis capabilities. With such information, it then computes data flow facts (e.g., reaching definitions) using a classical fixed-point iterative data-flow analysis algorithm [3]. Finally, it produces the interprocedural data dependence graph. Then in its practical use scenarios, for both practitioners and researchers, PCA enables different (e.g., data dependence based) application/client analyses and tools through its analysis results (e.g., interprocedural control flow and data dependencies).

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3417923

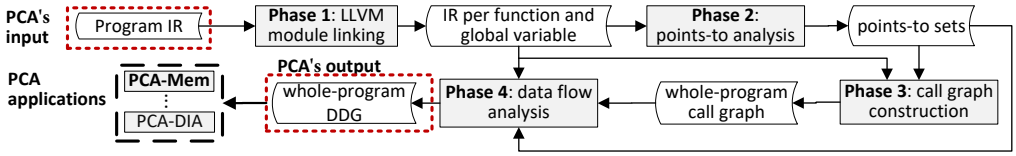

Figure 1: An overview of PCA's architecture, including its input, four phases, output, and applications.

The key merit of PCA is that it offers practical efficiency and scalability for real-world, industry-scale software systems while maintaining a practically useful level of effectiveness in terms of precision. To that end, PCA features a *partial call-path analysis* (hence our tool name) during the interprocedural data dependence computation, as defined by two levels of performance optimization. First, in the data-flow analysis algorithm, PCA adopts a user-customizable partial flow sensitivity at function level (i.e., via the call graph) when computing variable definitions and uses while referring to the points-to sets computed by the (Andersen's) flow-insensitive pointer analysis algorithm. This design substantially reduces the overhead of an otherwise heavy (e.g., flow-sensitive) points-to analysis to save the total time cost of PCA while compensating the precision loss from that underlying analysis later (during the dependence computation). Second, PCA opts to ignore constant strings in the points-to analysis to reduce its time and space costs; it further uses *integer encoding* in storing *definitions* when performing the data flow analysis to reduce memory use.

To demonstrate the usefulness of PCA, we built PCA-Mem as its example application, a static memory leak detector using the partial call-path analysis. We then evaluate PCA through PCA-Mem by applying it to a standard suite of (small) benchmarks (SPEC2000 C programs) and an industry-scale, high-performance computing system Slurm [20]. We assessed its efficiency and effectiveness in terms of memory leaks found in comparison to Saber [18], the same application but of the state-of-the-art value flow analysis SVF [17]. On the small SPEC2000 benchmarks, PCA-Mem was close to Saber in both metrics, while on the large system Slurm, PCA-Mem performed better by finding more memory leaks at lower time and space costs. PCA can support more applications than analyzing memory defects through the dependence information it offers.

2 ARCHITECTURE

PCA's architecture is depicted in Figure 1. As **PCA's input**, the per-module IR files of the given program are generated using the LLVM `gold` plugin [17]. Optionally, users may provide a function blacklist to specify the functions (e.g., those for debugging purposes only) to be skipped by the analysis for better efficiency.

With these user inputs, PCA performs its data flow analysis with optimizations in *four phases*. In the first phase (**LLVM module linking**), PCA links the per-module IR files together and parses the functions and global variables of the program, and returns the separated IR for each function and global variable to be used by other phases. The IR code is the output of the LLVM C frontend (Clang) applied against the program, where the gold plugin allows for generating cross-compilation-unit IR code. By preparing the cross-unit IR, this phase is key for the analyses in PCA to be interprocedural. In the second phase (**points-to analysis**), PCA performs an interprocedural points-to analysis for global, heap, and local variables, using the Andersen's algorithm [4]. During the third phase (**call graph construction**), PCA resolves call targets for direct calls with explicit callees and for indirect calls according to the points-to sets returned by the points-to analysis phase. In the last (fourth) phase (**data flow analysis**), PCA first constructs the interprocedural control flow graph (ICFG) from the given call graph and the LLVM instructions of each function, and then computes data dependence based on the ICFG and the points-to sets. This last phase produces the interprocedural data-dependence graph (DDG) as the final **PCA's output**. Based on this output, various **PCA applications** (e.g., taint checking, testing) can be developed. Currently, our tool package includes two applications: PCA-Mem, a static memory leak detector; and PCA-DIA, a dynamic impact analysis tool. For demo purposes, this paper only elaborates PCA-Mem.

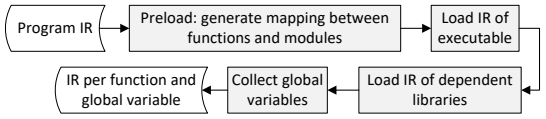

Figure 2: Module linking workflow of PCA.

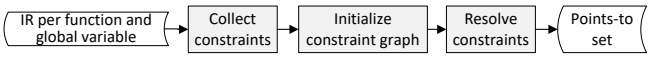

Figure 3: Points-to analysis workflow of PCA.

3 LLVM MODULE LINKING

To enable interprocedural analysis based on LLVM, PCA needs to first link the IR of all the modules of the given input program. Specifically, PCA retrieves all global variables and the definition of all functions in the executable and its dependent libraries, in four major steps as shown in Figure 2 and as elaborated below.

In the first (*preload*) step, PCA traverses all the (per-module) IR files of the input program, and parses each module to create a mapping between each function defined in the module to the module itself (i.e., its IR). Second, PCA loads the IR of the executable which contains the entry function of the program. It then parses each callsite in this function to check whether there is a callee for which merely the declaration is included in the current module—that is, whether the callee is defined in an *external module*. If so, PCA retrieves the module (i.e., its path) that defines the function (according to the mapping created before) and inserts the module path into a on-demand list for later loading. Third, PCA loads modules in the on-demand list and identifies functions defined in these modules the same way as functions are discovered in the previous step. To deal with circular library dependencies, PCA ensures each module is loaded once. Finally, PCA collects all global variables defined in the modules that are loaded in the previous two steps, and produces the set of function definitions and global variables (i.e., the IR of each) in the input program.

4 POINTS-TO ANALYSIS

In this phase, PCA takes the IR of each function and global variable from the previous phase to perform interprocedural points-to analysis of the program in three steps as outlined in Figure 3. The alias analysis implementations in LLVM are *intraprocedural*.

PCA first collects the constraints of stack and heap variables from each function, and also parameters and return values for each call instruction. It models each function in standard C library (libc) as an identity function but allows users to model library function effects differently. Second, from the collected constraints, PCA constructs a constraint graph and initializes a points-to set for each memory object. PCA uses SparseBitVector [10] data structure for efficient set operations. In the next step, PCA adopts the *hybrid cycle detection* technique [11] for efficient constraint resolution. The technique utilizes both an offline pre-analysis and online detection of strongly connected components to achieve high efficiency. In all, we adopted the fast and imprecise Andersen's algorithm [4] for a flow- and context-insensitive points-to analysis to trade precision in the resulting points-to sets for the efficiency of computing them.

Optimization. To further reduce its overheads, PCA has the default option of ignoring constant strings in this phase. Including these constants would in general significantly increase the size of the resulting points-to sets hence slow down the analysis due to their prevalence in C programs. For example, these constants account for over 80% of all global variables in Slurm.

5 CONTROL AND DATA FLOW ANALYSIS

With the results from the previous phases, PCA now constructs the whole-program call graph, identifying all possible callees according to the points-to information. It performs function-level reachability analysis through a (BFS) traversal of the initial call graph and prunes nodes unreachable from the program entry.

From the resulting call graph and per-function IR, PCA then builds the (intraprocedural) control flow graph (CFG) for each function hence the interprocedural CFG (ICFG) for the entire program as in [12]. To construct the interprocedural data dependence graph (DDG), PCA collects both definite and possible (induced by pointer aliasing) definitions/uses of variables at each instruction of every reachable function. Then, it follows the classical fixed-point iterative data flow analysis algorithm [3] to compute the reaching definitions and build intraprocedural DDGs. Finally, based on the ICFG, PCA computes interprocedural data dependence by adding transitive edges (e.g., actual/formal parameters linkage) among intraprocedural DDGs as in the classical interprocedural slicing [12].

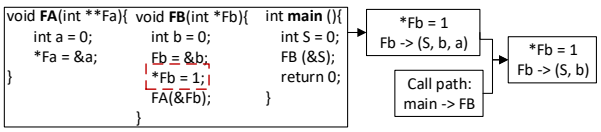

Figure 4: Illustrating part of the *partial call-path analysis*.

Optimizations. PCA improves its efficiency and scalability in the data flow analysis here through two optimizations, as the main part of our *partial call-path analysis*.

The first is to reduce the effects of false positives in the points-to sets by considering function-level control-flow reachability (i.e., on the call graph) to a certain depth. The partial flow sensitivity compensates the precision lost in the flow-insensitive points-to analysis without incurring much cost.

To illustrate the idea, consider the example in Figure 4. When identifying possible definitions at `*Fb = 1`, based on the points-to results, the definitions of {S, b, a} will be identified. However, considering control-flow reachability with respect to call paths of depth 2 (e.g., `main− >FB`), it is clear that the definition of a should **not** be included in the set of possible definitions at `*Fb = 1`. To prune such false definitions, PCA collects the set of definitions (including those in stack, heap, and global data area) along the call paths considered; it then takes the intersection of this set and the original points-to set at the target node (e.g., the node for `Fb` in this example), resulting in the reduced set of possible definitions (e.g., {S, b} in this case). With this reduced set, the following iterative data flow analysis can converge faster hence be accelerated while producing more precise dependence analysis results. PCA allows users to customize this call-path depth threshold as an option based on program complexity. In essence, the optimization safely prunes spurious aliasing-induced data dependence according to control flow up to the given depth on the call graph. Thus, this optimization does not affect the soundness of our analysis.

The second is to use integer encoding in storing data flow facts (variable definitions in particular) to reduce memory usage. As per the algorithm we use, each definition would be conventionally represented as a 2-tuple `(instruction, variable)`, which would take 16 bytes (on a 64bit machine). And four sets of definitions (IN, OUT, GEN, KILL) for each ICFG node need to be maintained during the iterative analysis until the fixed point is reached. For one of the functions in Slurm, for example, the total number of definitions maintained is 17,097,929, thus computing the reaching definitions in this function alone would consume at least 256MB memory. PCA encodes definitions with a simple (short) integer index which only takes 4 bytes per definition. This reduces 75% of the peak memory usage of the data flow analysis—for the same example, only about 64MB memory is needed.

6 EXAMPLE APPLICATION: PCA-MEM

We developed PCA-Mem based on PCA to demonstrate its use (for statically detecting memory leaks). Using the interprocedural DDG from PCA, PCA-Mem addresses this use case as a source-sink problem, with each allocation site considered a source and each free site as a sink. The idea is to check reachability from a source to each corresponding sink against data and control flow conditions.

First, PCA-Mem collects the set (N) of nodes reachable from the source on the DDG through any realizable path by considering different calling contexts leading to each call. Let S be the set of sinks in N—the memory allocated at the source can have more than one free site. If S is empty, then the memory *is never freed*.

If S is not empty, PCA-Mem will employ both control flow and data dependence information for detecting *partial memory leaks* along some program paths. A heap object o allocated at a memory allocation site ℓ is treated as deallocated if (1) there is a free site (sink) $\ell' \in S$ that is control-flow reachable from ℓ along the ICFG, and (2) the object freed at ℓ' must be object o or its alias as determined by our reaching definition analysis. Otherwise, o is treated as a leaked object. PCA-Mem reports both never freed cases and partial leaks.

Table 1: Memory leak detection results of PCA-Mem versus Saber on ten SPEC2000 benchmarks.

Program	Size (KLOC)	#Functions	PCA-Mem				Saber			
			Time (seconds)	Peak Memory (GB)	#Leaks (#GV)	#False Alarms	Time (seconds)	Peak Memory (GB)	#Leaks	#False Alarms
art	1.2	29	0.5	0.04	1 (9)	0	0.2	0.05	2	1
bzip2	4.7	77	1.5	0.07	1 (9)	0	0.5	0.09	1	0
crafty	21.2	112	15	0.35	0 (12)	0	2.3	0.6	0	0
equake	1.5	30	3	0.16	0 (29)	0	0.5	0.07	0	0
gzip	8.6	113	1	0.05	3 (0)	1	0.6	0.1	3	1
mcf	2.5	29	0.3	0.03	0 (3)	0	0.3	0.05	0	0
parser	11.4	327	10.2	0.2	0 (10)	0	3	0.3	0	0
twolf	20.5	194	70	0.8	3 (46)	1	7	0.7	3	1
vpr	17.8	275	6.6	0.16	1 (19)	0	2.6	0.38	1	0
mesa	61.3	1109	5.9	0.3	12 (4)	7	108	4.1	10	5
total	150.7	2295	114	2.16	21 (141)	9	125	6.44	20	8

7 EVALUATION

We evaluate PCA via PCA-Mem against SVF's Saber as the baseline, using 10 *SPEC2000* C programs and Slurm (v15.8.7) [20], a real-world, industry-scale workload manager that includes 21 subsystems (with sizes ranging from 186KLOC to 257KLOC). We then manually checked and compared all the detected memory leaks, including never-free and partial leaks. Our experiments ran on an Ubuntu 16.04 server with 2.40G CPU and 512GB DDR3.

7.1 Efficiency and Effectiveness

Table 1 summarizes the accuracy and time/memory costs of both tools against the SPEC2000 benchmarks. PCA-Mem found 161 leaks with 9 false alarms in about 114 seconds, consuming 2.16 GB memory. In comparison, Saber found 20 leaks with 8 false alarms in about 125 seconds, consuming 6.44 GB memory—the true positives it found were all part of those found by PCA-Mem. As an implementation merit here, PCA-Mem handles global variables and it thus found associated memory leaks (noted as "#GV" in the parentheses). Saber dismisses the GV cases [17]. Excluding such cases, PCA-Mem appeared to be very close to the baseline in effectiveness (in terms of the number of leaks reported and that of false positives)—there were no false alarms among the GV cases.

For almost all of the benchmarks, PCA-Mem took longer time than Saber. This was expected and can be explained by the fact that the underlying analysis of Saber provides information (i.e., sparse value flow) that is lesser and cheaper to compute as in [9], compared to PCA (which underlies PCA-Mem). As mentioned earlier, PCA chooses this relatively heavier analysis to compute information for broader applications. The only exception was with `mesa`, for which Saber was much slower, mainly because its interprocedural analysis is context-sensitive and the value flow computation is fully flow-sensitive. This design incurs especially high costs when there are a large number of functions that form a very deep call graph, which is the case with `mesa`. In contrast, the partial flow sensitivity of PCA-Mem's underlying analysis saved greatly in such cases. As a result, the total costs were close between the two tools.

For the largest benchmark Slurm (v15.8.7), PCA-Mem reported 21 leaks including 10 GV cases. Four of these have been confirmed and fixed by the developers (in v18.8.7): *never freed* memory allocated at hostlist.c:3802 and that at hostlist.c:3803; *partial leaks* of memory allocated at hostlist.c:1328 (leak branch: hostlist.c:2253), and that at hostlist.c:1335 (leak branch: hostlist.c:2253). Saber did not find any of these four—the 10 leaks it reported were all false alarms.

Figure 5 and Figure 6 compare the tools on peak memory usage and time cost (y axes), respectively, for the 21 Slurm subsystems (x axes). PCA-Mem finished analyzing the entire system (4,079KLOC) in 40mins with peak memory usage of 2.8GB, while Saber finished in 58mins with peak memory usage of 11.6GB. The reason was similar to that for the `mesa` case in the SPEC2000 benchmarks. As for those benchmarks, PCA-Mem had higher memory efficiency because of its integer encoding of variable definitions.

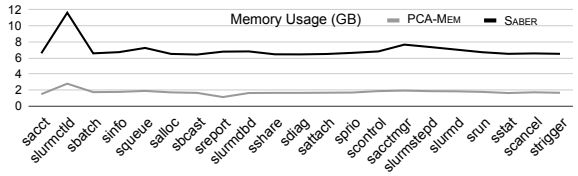

Figure 5: Comparison on peak memory usage against Slurm.

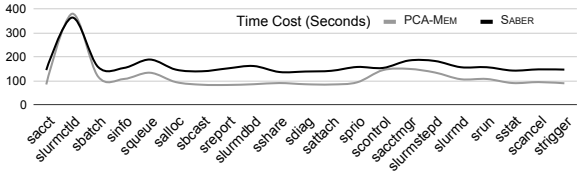

Figure 6: Comparison on time costs against Slurm.

7.2 Limitations

We made design decisions that lead to imprecision (flow-, field-, and context-insensitivity in computing and using points-to sets) to trade for scalability and efficiency. Thus, the data dependence computed by PCA suffers imprecision. Accordingly, PCA-Mem can give false alarms (as our evaluation results confirmed). Like peer tools (e.g., PhASAR [15]), PCA as a static analyzer does not handle dynamically loaded code.

8 CONCLUSION

We presented PCA, a static interprocedural data-flow analyzer for C programs that offers several efficiency optimization options and different cost-precision tradeoffs from peer tools. Based on PCA we further developed PCA-Mem, a static memory leak detector as an example application of our tool. Through PCA-Mem, we empirically demonstrated PCA's merits in efficiency with practical effectiveness against both standard benchmarks and an industry-scale real-world system. PCA and PCA-Mem are open source and publicly available.

ACKNOWLEDGMENTS

This work was supported by DOE through PNNL (No. 379101).

REFERENCES

[1] August 2018. CodeSonar. https://www.grammatech.com/products/codesonar.

[2] August 2018. Coverity. https://scan.coverity.com/.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[4] Lars Ole Andersen. 1994. Program Analysis and Specialization for the C Programming Language. (1994).

[5] Haipeng Cai. 2018. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *IEEE Transactions on Software Engineering (TSE)* 44, 4 (2018), 334–364.

[6] Haipeng Cai and Raul Santelices. 2015. Abstracting Program Dependencies using the Method Dependence Graph. In *International Conference on Software Quality, Reliability, and Security (QRS)*. 49–58.

[7] Haipeng Cai, Raul Santelices, and Siyuan Jiang. 2016. Prioritizing Change Impacts via Semantic Dependence Quantification. *IEEE Transactions on Reliability* 65, 3 (2016), 1114–1132.

[8] Haipeng Cai, Raul Santelices, and Douglas Thain. 2016. DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 18:1–18:50.

[9] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 72–82.

[10] Ben Hardekopf and Calvin Lin. [n.d.]. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*.

[11] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 290–299.

[12] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.

[13] Chris Lattner and Vikram Adve. [n.d.]. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*.

[14] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1-2 (1996), 131–170.

[15] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 393–410.

[16] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 693–706.

[17] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[18] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122.

[19] Yichen Xie and Alex Aiken. 2005. Saturn: A SAT-based tool for bug detection. In *International Conference on Computer Aided Verification*. Springer, 139–143.

[20] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 44–60.

APPENDIX: PCA DEMO WALKTHROUGH

In this section, we walk through how the key ((operational) portion of our demo for PCA is conducted by using illustrative screen-shots to show the installation and usage of the tool.

1. Install PCA

Here we present two ways to get and install PCA. Many of the detailed steps have been included in scripts referred to below, which makes the installation highly automated and easy to do.

1.1 For Using PCA through Virtual Machine.

- Step 1. Download PCA-VM.zip (the virtual disk image) through this link.
- Step 2. Launch the virtual machine in VirtualBox (username/-password: pca/pca, root/pca).
- Step 3. Open a terminal, and go to the directory /home/p-ca/PCA.

1.2 For Using PCA through source code compilation.

- Step 1. Check prerequisites. We need to first make sure that the following prerequisite environments are set up in order for PCA to be installed and run successfully.
  - UNIX (Ubuntu 16.04 LTS or Ubuntu 18.04 LTS)
  - LLVM (v7.0.0) — we provide a script to assist with this (see Step 3 below)
- Step 2. Download the source code of PCA through this link.
- Step 3. Enter directory PCA/llvm7 and run `installLLVM.sh`, which will install LLVM7 and configure environment variables automatically.
- Step 4. Enter directory PCA and build PCA with the script `build.sh`.

2. Use PCA

In this section, we demonstrate a case study of PCA, PCA-Mem, and introduce how to run PCA-Mem against a subject program for memory leak detection.

2.1 Compile the subject program. To enable data-dependence analysis based on LLVM, the subject program needs to be compiled with clang and gold-plugin (refer to here for details).

For the simple subject used in the demo, the command line for this step is:

```
clang -flto leak.c -c -o leak.bc
```

For the Slurm system subject used in the demo, the command line for this step is:

1. Specify environment variables for the compiler:
   ```
   export CC="clang -flto"
   export CXX="clang++ -flto"
   export RANLIB=/bin/true
   ```
2. Compile Slurm: `./configure && make`

2.2 Run PCA-Mem against simple program. In this step, we present how to run memory leak detection with PCA-Mem and generate the data-dependence graph (DDG) (for visual understanding purposes).

A small test case (leak.bc) is shown as Figure 7, where there are two partial free defects obviously. Both are detected by PCA-Mem (with command: PCA-Mem -file leak.bc) as shown (Figure 8).

```
char* Malloc (int Size) { return malloc (Size);}
int main(int argc, char** argv) {
    char *m1 = Malloc (4);
    char *m2 = Malloc (4);
    if (argc) {
        free (m1);}
    else {
        free (m2);
    }
    return 1;
}
```

Figure 7: A simple example program for demo purposes.

```
#=====================================================
#start mempory leak detection....
#=====================================================
        Partial Free : memory allocation at : line: 11 file: source/main.c
        Partial Free : memory allocation at : line: 12 file: source/main.c
Compute MemCheck cost-time: 8.00 (ms), memory-usage: 0 (KB)
Total Memory usage:14436 (K)
root@ubuntu:/home/liwen/PCA/build/bin#
```

Figure 8: Memory leak detection result for the simple program `leak.bc` produced by PCA-Mem.

We then present the use of an option (–dump-DDG) that generates DDG.dot, which can be then opened by GVEdit as shown below (ICFG is shown in black color while DDG in red). In this case, we may use this visualization to help verify the correctness of the DDG manually while debugging.

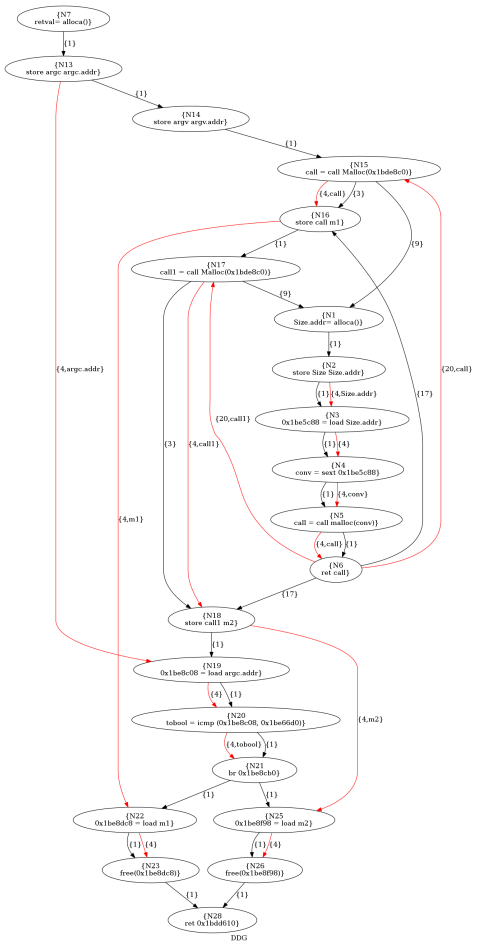

Figure 9: ICFG, DDG of the simple example program `leak.bc` as generated by our tool.

2.3 Run PCA-Mem against Slurm. For a large-scale program like Slurm (Version 15.08.7), which usually contains multiple modules, PCA-Mem performs its analysis in two steps:

- Pre-process: Compute dependencies between modules (e.g., PCA-Mem -dir Slurm -pre=1, as shown in Figure 10).

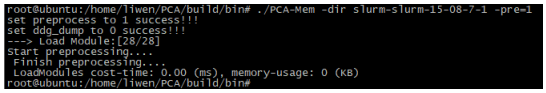

Figure 10: The pre-process step for Slurm by PCA-Mem.

- Program analysis: Link all necessary IR of modules for the target executable and perform data dependence analysis and memory leak detection in sequence (e.g., PCA-Mem -file Slurm/salloc.bc; part of the result is shown in Figure 11).

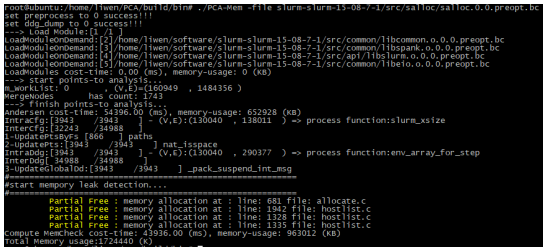

Figure 11: Part of the memory leak detection results for the `scalloc` module of Slurm produced by PCA-Mem.