DG: Analysis and Slicing of LLVM Bitcode*

Marek Chalupa

Masaryk University, Brno, Czech Republic, chalupa@fi.muni.cz

Abstract. DG is a library written in C++ that provides several types of program analysis for LLVM bitcode. The main parts of DG are a parametric points-to analysis, a call graph construction, a data dependence analysis, and a control dependence analysis. The project includes several tools built around the analyses, the main one being a static slicer for LLVM bitcode. This paper describes what analyses are implemented in DG and its use cases with the focus on program slicing.

1 Introduction

DG is a library providing data structures and algorithms for program analysis. The library was created during the re-implementation of the static slicer in the tool Symbiotic [8] and its original purpose was the construction of *dependence graphs* [12] for LLVM bitcode [14]. During the development, we re-designed DG from a single-purpose library for the construction of dependence graphs to a library providing data structures and basic algorithms for program analysis.

The main parts of DG are a parametric *points-to analysis* and a *call graph construction*, a *data dependence analysis* based on the transformation of writes to memory into static single assignment (SSA) form [18], and a *control dependence analysis* providing two different algorithms with different characteristics. The results of these analyses can be used to construct a *dependence graph* [12] of the program that supports forward and backward slicing, among others.

Most of the implemented algorithms are designed to be independent of the programming language. Currently, DG has an LLVM backend that allows using the algorithms with LLVM infrastructure.

Analyses in DG have a public API that is used also in communication between analyses inside DG. As a result, a particular implementation of analysis can be easily replaced by an external analysis. The benefit of being able to integrate an external analysis is that one can use features of DG (e.g., program slicing) along with features of the external analysis (e.g., better speed or precision). At this moment, DG integrates a points-to analysis from the SVF library [19].

LLVM DG works with LLVM [14], which is a strongly typed assembly-like intermediate language for compilers. Instructions in LLVM are arranged into labeled basic blocks to which we can jump using the `br` instruction. Variables on the stack are created by the `alloca` instruction and can be later accessed via

* The work is supported by The Czech Science Foundation grant GA18-02177S.

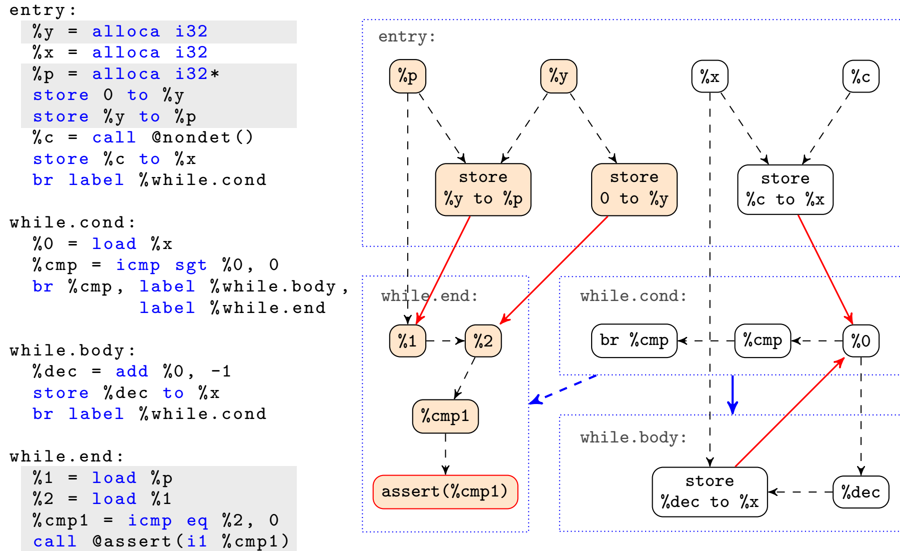

Fig. 1. A simplified LLVM bitcode and its dependence graph. For clarity, we left out nodes with no dependencies. Black dashed edges are use dependencies, red edges are data dependencies, and blue edges are control dependencies. The dashed control dependence is present in the graph only when NTSCD algorithm is used. Highlighted are the nodes that are in the slice with respect to the call of `assert` (using SCD).

the `load` and `store` instructions. The meaning of the rest of the instructions used in this paper should be clear from the text. An example of LLVM bitcode can be found in Figure 1.

In the rest of the paper, we describe the main analyses in DG and its use cases.

2 Points-to Analysis

Points-to analysis is a cornerstone of many other program analyses. It answers the queries: “What is the memory referenced by the pointer?”. For each points-to analysis, we can identify the following basic traits. Points-to analysis is *flow-sensitive* (FS) if it takes into account the flow of control in the program and thus computes information specifically for each program location. Otherwise, it is *flow-insensitive* (FI). It is *field-sensitive* if it differentiates between individual elements of aggregate objects, e.g., arrays or structures.

In DG, we have implemented a parametric points-to analysis framework [6] that supports FS and FI analysis and has dynamically configurable field-sensitivity. Moreover, the analysis can construct a call graph of the program. The FS analysis has also an option to track what memory has been freed [9].

3 Data Dependence Analysis

Data dependence analysis is a crucial part of program slicing. Informally, we say that instruction r is data dependent on instruction w if r reads values from memory that may have been written by w. In Figure 1, for example, instruction `%0 = load %x` is data dependent on instructions `store %c to %x` and `store %dec to %x` as it may read values written by both of these instructions.

In DG, the data dependence analysis constructs SSA representation of writes to memory, the so-called memory SSA form [16]. The input to the data dependence analysis in DG is a program whose instructions are annotated with information about what memory *may/must* be written, and what memory *may* be read by the instructions. These annotations are derived from the results of the points-to analysis.

Our analysis algorithm is based on the algorithm of Braun [5]. We extended the Braun's algorithm, which works only with scalar variables, to handle aggregated data structures, heap-allocated objects and accesses to memory via pointers, and also accesses to unknown memory objects (occurring due to a lack of information about the accessed memory). Also, the algorithm has been modified to handle procedure calls, therefore it yields interprocedural results.

4 Control Dependence Analysis

Informally, control dependence arises between two program instructions if whether one is executed depends on a jump performed at the other. There are several formal notions of control dependence. In DG, we implement analyses that compute two of them. The first one is *standard control dependence* (SCD) as defined by Ferrante et al. [12] and the other is *non-termination sensitive control dependence* (NTSCD) introduced by Ranganath et al. [17]. The difference between these two is that NTSCD takes into account also the possibility that an instruction is not executed because of a non-terminating loop.

For example, in Figure 1, instructions in `while.body` basic block are (standard and non-termination sensitive) control dependent on the `br %cmp` instruction from `while.cond` block as the jump performed by the `br %cmp` instruction may avoid their execution. If the loop in the program does not terminate, we will never get to `while.end` basic block, and therefore NTSCD marks also instructions from this block to be dependent on the `br %cmp` instruction.

The classical algorithms compute control dependencies per instruction. For efficiency, our implementation allows also control dependencies between basic blocks, where if basic block A depends on basic block B than it represents that all instructions from the basic block A depend on the jump instruction at the end of the basic block B. See, for example, the control dependence edge between `while.cond` and `while.body` basic blocks in Figure 1.

We have also an implementation of the computation of interprocedural control dependencies that arise e.g., when calling `abort()` inside procedures. This analysis runs independently of SCD and NTSCD analysis.

5 Dependence Graphs and Program Slicing

The results of hitherto mentioned analyses can be used to construct a dependence graph of the program. A dependence graph is a directed graph that has instructions of the program as nodes and there is an edge from node n_1 to node n_2 if n_2 depends on n_1.

In DG, we distinguish three types of dependencies. The first two types are control and data dependencies as computed by control and data dependence analysis. The last dependence is *use* dependence that is the syntactic relation between instruction and its operands. For instance, in Figure 1, there is use dependence from instruction `%y = alloca i32` to `%0 = load %y` as the later uses `%y`.

Program Slicing *Static backward program slicing* [21] is a method that removes parts of a program that cannot affect a given set of instructions (called *slicing criteria*). Dependence graphs are a suitable representation for program slicing [12] as they capture dependencies between instructions. The set of instructions that comprise a slice is obtained by traversing the graph in the backward direction from nodes corresponding to slicing criteria. In our example in Figure 1, the slice with respect to the call to function `assert` contains all instructions that are backward reachable from the call node in the dependence graph. Depending on whether we use SCD or NTSCD, the slice contains either the highlighted instructions (SCD) or all instructions (NTSCD).

One of the prominent features of our slicer is that we produce executable slices. That is, unlike tools that just output the set of instructions in the slice, we produce a valid sliced bitcode that can be run or further analyzed.

6 Evaluation and Use Cases

We evaluated the effectiveness of our analyses by running our slicer on a set of 8107 reachability benchmarks from Software Verification Competition [1]. These benchmarks range from small artificial programs (tens of instructions) to complex code generated from Linux kernel modules (up to 130000 of instructions). The average size of a benchmark is approximately 5320 instructions. Each benchmark contains calls to an error function which we used as slicing criteria (all together if there were multiple calls). The experiments ran on a machine with *Intel i7-8700 CPU @ 3.2 GHz*. Each benchmark run was constrained to 6 GB of memory and 120 s of CPU time.

Figure 2 on the left shows a quantile plot of the CPU time of slicing with different setups of pointer and control dependence analyses. Slicing is mostly very fast – more than 75 % of benchmarks is sliced in 1 s (more than 80 % for FI setups). However, in each setup are benchmarks on which slicing either timeouts (around 380 benchmarks for FI setups and 280 for FS setups) or crashed e.g., due to hitting the memory limit (around 860 for FI setups and 1400 for FS setups).

[1] https://github.com/sosy-lab/sv-benchmarks, rev. `6c4d8bc`

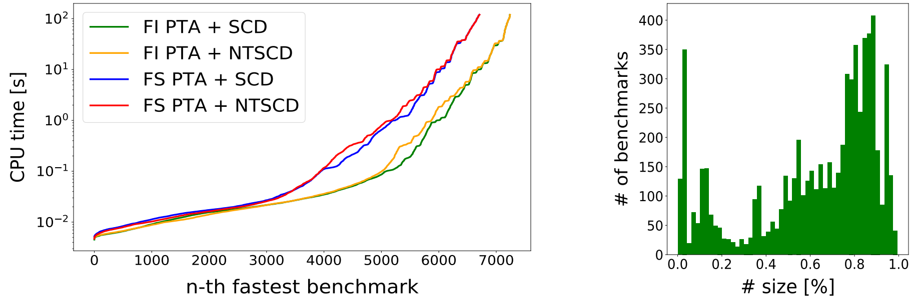

Fig. 2. The plot on the left shows CPU time of slicing using FS or FI points-to analysis (PTA) and NTSCD or SCD control dependence analysis. On the right is depicted the ratio of the number of instructions after and before slicing for FI PTA + SCD setup.

In Figure 2 on the right is depicted the distribution of the ratio of the number of instructions after and before slicing for *FI PTA + SCD* setup (the slicer's default). On average, for this setup, the size of the sliced bitcode was reduced to 67 % of the size before slicing, but there are also numerous cases of reduction to less than 30 %. FS points-to analysis has no big influence on these numbers.

Use cases Since its creation, DG has proved to be useful in many cases, e.g., software verification and bug finding [7, 8, 9, 20, 15] cyber-security [3, 13] cyber-physical systems analysis [10], and network software analysis [11].

Availability DG library and documentation is available under the MIT license at https://github.com/mchalupa/dg.

7 Related Work

Many analyses, including memory SSA construction and various alias analyses, are contained directly in the LLVM project. However, these analyses are usually only intraprocedural and thus too imprecise for sensible program slicing.

Now we survey the projects providing backward program slicing for LLVM. *ParaSlicer* [2] and *llvm-slicing* [22] are projects written in Haskell that make use of procedure summaries to generate more precise slices than the classical slicing algorithms. These slicers only output a list of instructions that should be in the slice. *SemSlice* [4] is a slicer for semantic slicing of LLVM bitcode. The bottle-neck of semantic slicing is the use of SMT solving, which can be inefficient. Finally, there is the obsolete slicer from Symbiotic called *LLVMSlicer* [1] which is no longer maintained.

Acknowledgements The author would like to thank Jan Strejček for his valuable comments on the paper. Further, many thanks go to other contributors to the DG library, mainly Tomáš Jašek, Lukáš Tomovič, and Martina Vitovská.

References

1. LLVMSlicer. URL https://github.com/jslay/LLVMSlicer.
2. ParaSlicer. URL https://archive.codeplex.com/?p=paraslicer.
3. M. Ahmadvand, A. Hayrapetyan, S. Banescu, and A. Pretschner. Practical integrity protection with oblivious hashing. In *ACSAC'18*, pages 40–52. ACM, 2018. https://doi.org/10.1145/3274694.3274732.
4. B. Beckert, T. Bormer, S. Gocht, M. Herda, D. Lentzsch, and M. Ulbrich. SemSlice: Exploiting relational verification for automatic program slicing. In N. Polikarpova and S. Schneider, editors, *IFM'17*, volume 10510 of *Lecture Notes in Computer Science*, pages 312–319. Springer, 2017. https://doi.org/10.1007/978-3-319-66845-1_20.
5. M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In R. Jhala and K. D. Bosschere, editors, *CC'13*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2013. https://doi.org/10.1007/978-3-642-37051-9_6.
6. M. Chalupa. Slicing of LLVM Bitcode. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2016. URL https://is.muni.cz/th/vik1f/.
7. M. Chalupa and J. Strejček. Evaluation of program slicing in software verification. In W. Ahrendt and S. L. T. Tarifa, editors, *IFM'19*, volume 11918 of *Lecture Notes in Computer Science*, pages 101–119. Springer, 2019. https://doi.org/10.1007/978-3-030-34968-4_6.
8. M. Chalupa, T. Jašek, L. Tomovič, M. Hruška, V. Šoková, P. Ayaziová, J. Strejček, and T. Vojnar. Symbiotic 7: Integration of predator and more (competition contribution). In A. Biere and D. Parker, editors, *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'20)*, volume 12079 of *Lecture Notes in Computer Science*, pages 413–417. Springer, 2020. https://doi.org/10.1007/978-3-030-45237-7_31.
9. M. Chalupa, J. Strejček, and M. Vitovská. Joint forces for memory safety checking revisited. *Int. J. Softw. Tools Technol. Transf.*, 22(2):115–133, 2020. https://doi.org/10.1007/s10009-019-00526-2.
10. L. Cheng, K. Tian, D. Yao, L. Sha, and R. A. Beyah. Checking is believing: Event-aware program anomaly detection in cyber-physical systems. *CoRR*, abs/1805.00074, 2018.
11. B. Deng, W. Wu, and L. Song. Redundant logic elimination in network functions. In A. Wang, E. Rozner, and H. Zeng, editors, *SOSR'20*, pages 34–40. ACM, 2020. https://doi.org/10.1145/3373360.3380832.
12. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
13. H. Hamadeh, A. Almomani, and A. Tyagi. Probabilistic verification of outsourced computation based on novel reversible pufs. In A. Brogi, W. Zimmermann, and K. Kritikos, editors, *ESOCC'20*, volume 12054 of *Lecture Notes in Computer Science*, pages 30–37. Springer, 2020. https://doi.org/10.1007/978-3-030-44769-4_3.
14. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.
15. Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin. VulDeeLocator: A deep learning-based fine-grained vulnerability detector. *CoRR*, abs/2001.02350, 2020. URL http://arxiv.org/abs/2001.02350.

16. D. Novillo. Memory SSA - A Unified Approach for Sparsely Representing Memory Operations, 2006.
17. V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *Proceedings of the 14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2005.
18. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In J. Ferrante and P. Mager, editors, *POPL'88*, pages 12–27. ACM Press, 1988. https://doi.org/10.1145/73560.73562.
19. Y. Sui and J. Xue. SVF: interprocedural static value-flow analysis in LLVM. In A. Zaks and M. V. Hermenegildo, editors, *CC'16*, pages 265–266. ACM, 2016. https://doi.org/10.1145/2892208.2892235.
20. D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *ICSE'18*, pages 350–360. ACM, 2018. https://doi.org/10.1145/3180155.3180251.
21. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
22. Y. Zhang. Sympas: Symbolic program slicing. *CoRR*, abs/1903.05333, 2019. URL http://arxiv.org/abs/1903.05333.