Software Impacts 6 (2020) 100038

Contents lists available at ScienceDirect

Software Impacts

journal homepage: www.journals.elsevier.com/software-impacts

Original software publication

DG: A program analysis library

Marek Chalupa

Masaryk University, Brno, Czech Republic

ARTICLE INFO

Keywords:
Program analysis
Points-to analysis
Dependence analysis
Dependence graphs
Program slicing

ABSTRACT

DG is a C++ library providing elements for building program analysis tools. Its main components are a points-to analysis, a data dependence analysis, a control dependence analysis and an analysis of relations between variables. DG contains also a set of tools for displaying and exporting the results of the analyses and a program slicer for LLVM bitcode. It has been successfully used in several research projects.

Code metadata

Current code version	v0.9.1
Permanent link to code/repository used for this code version	https://github.com/SoftwareImpacts/SIMPAC-2020-44
Permanent link to Reproducible Capsule	
Legal Code License	MIT
Code versioning system used	git
Software code languages, tools, and services used	C++, python
Compilation requirements, operating environments & dependencies	python, C++ compiler, LLVM 3.4 or newer, cmake
If available Link to developer documentation/manual	https://github.com/mchalupa/dg/blob/master/doc/README.md
Support email for questions	chalupa@fi.muni.cz

1. About the DG library

Program analysis is a wide area of research concerned with automatically deriving properties of computer programs, like the possible runtime values of variables or whether an assertion in the program may fail. *Static* program analysis strives to derive such properties without actually executing the program, which is very useful e.g., during early development phases when none executable form of the program is available.

Our contribution to the static program analysis field is the library DG that contains an implementation of several basic program analyses and data structures that can be further used by more complex tools.

The analyses are internally programmed independently of the input programming language, but we provide an LLVM[1] frontend for all the analyses so that they can be easily used by external tools.

Also, the library itself contains several tools that use the analyses to analyze LLVM bitcode. One of these tools is a static program slicer [3], that is, a tool that can remove instructions from programs that cannot have any effect on a particular (user-specified) behavior of the program.

2. Key features

The main analyses and data structures implemented in DG are depicted in Fig. 1. In the following, we briefly describe them and their interactions.

Points-to analysis is one of the cornerstones of program analysis [4]. Points-to analysis answers the question *"What memory regions can the given pointer refer to?"* The results of a points-to analysis can be used in a wide range of areas, like program optimization, error detection, or software security to mention some [5–10]. Inside DG, we use the points-to analysis for creating the input for *data dependence analysis* and we can use it optionally in the *control dependence analysis* and the *call-graph construction*.

Call-graph captures the information about what functions may be called by the given function [11]. If the analyzed program contains calls via function pointers, points-to analysis must be used for resolving the targets of function pointer calls.

E-mail address: chalupa@fi.muni.cz.

[1] LLVM [1,2] is a popular assembly-like program representation designed for compiler construction. It has a strong developer community support.

https://doi.org/10.1016/j.simpa.2020.100038
Received 9 October 2020; Received in revised form 27 October 2020; Accepted 30 October 2020

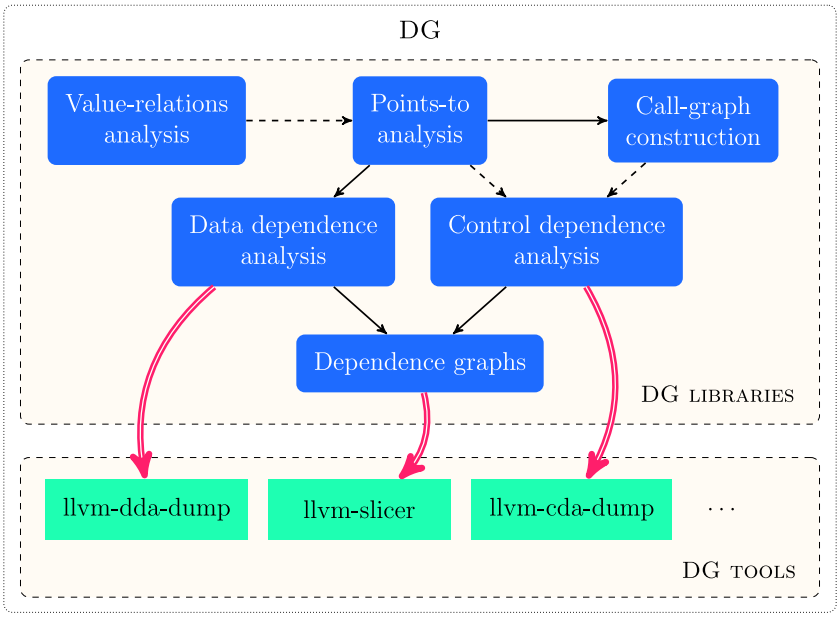

Fig. 1. The structure of the DG library. A black arrow between library modules means that the source module is used by the target module. If the arrow is dashed, then the usage is optional. Violet arrows depict that the module is used by the tool.

Data dependence analysis, also called *value-flow analysis* [12], answers the question *"Given an instruction N that reads from the memory, what instructions may have written the value that N is reading?"*. Data dependence analysis may be used to optimize and parallelize code or speed-up other program analyses [13–18]. For programs that use pointers, points-to analysis is required to resolve what memory is read and written by instructions.

Control dependence analysis [14,19,20] computes for each conditional jump in the program a set of instructions that may not be executed due to that jump. More precisely, it answers the question *"Given an instruction N, what is the closest conditional jump that may result in not executing N?"*. Points-to analysis or call-graph are used for resolving function pointer calls if the user wants to compute control dependencies also between instructions from different functions.

Dependence graphs [14,21] are directed graphs that have instructions as nodes and there is an edge between two nodes if one instruction is (control or data) dependent on the other. Dependence graphs have a variety of uses in e.g., program optimization and parallelization, or test generation [14,22,23]. However, one of the main uses is program slicing, that is, removing instructions from the program that do not affect a specified behavior. In DG, program slicing is implemented in the tool `llvm-slicer`.

Value-relations analysis in DG is an analysis that computes relations (e.g., $x > 0$, $y = z$, etc.) between variables (variables and constants, respectively) that must hold in any execution of the program. This analysis can be used e.g., to perform simple out-of-bound checks for pointers or to perform constant propagation. In DG, we can use this analysis to refine the results of the points-to analysis.

The analyses in DG are all *interprocedural*,[2] that is, the analyses compute the result for the whole program instead of for separate functions. Data and control dependence analyses work also *on-demand* – they do not compute the whole information at once, but compute it by parts per user requests.

Every analysis in DG is a self-standing module which comprises of a language-independent implementation of the analysis itself, and an adaptor that creates the input for the analysis from the given program in a specific programming language and that maps the results of the analysis back to the input program. At this moment, we have adaptors for LLVM bitcode.

Every module provides a public API for specifying the inputs and handling the results. This allows DG to easily integrate external analyses by writing an adaptor that converts inputs and outputs of the external analysis to the DG API. At this moment, we have integrated a points-to analysis from the SVF [12] project.

3. Limitations

DG targets programs in LLVM bitcode that was generated from C programming language. Other LLVM bitcode, e.g., generated from C++ or Rust, can be problematic as it may contain features like exceptions handling, which is currently unsupported (however, if such features are not present in the program, DG should be able to analyze it).

Also, we have only limited support for the analysis of parallel programs.

Another current limitation is the scalability of our points-to analysis. However, until we develop better points-to analysis, this can be overcome by using the bindings to the SVF framework (see the end of the previous section).

4. Impact overview

DG was originally created to replace the program slicer in the tool Symbiotic [24,25] which is a bug-finding tool for sequential C programs that uses program slicing to reduce the input program. At the time of the creation of DG, there was no other suitable slicer for LLVM bitcode and the one in Symbiotic was insufficient. The program slicer based on DG enabled Symbiotic to be used in research [26–28], but also in the industry [29] and competitions [30–33]. In all these scenarios, Symbiotic uses DG to perform program slicing to obtain a possibly smaller input program that preserves the specified kind of errors that are being sought for.

Apart from Symbiotic, DG has been used also in Chopper [34], another bug-finding tool. Similarly to Symbiotic, Chopper uses DG to slice away parts of the program that may only hinder it from achieving its goal but have no effect on the result. In this case, program slicing is

[2] The control dependence analysis is by default *intraprocedural*. However, there is a switch that changes this behavior.

used to reduce the number of paths inside a function that need to be explored in order to find one that is compatible with a given context.

Finally, DG has been used to determining dependencies between instructions of LLVM bitcode in several other areas different from program verification and bug finding. These areas include cyber-security [35–38], analysis of cyber-physical systems [39,40], and network software analysis [41].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Many thanks go to other contributors to the DG library, mainly Tomáš Jašek, Anna Řechtáčková, Lukáš Tomovič, Martina Vitovská, and Lukáš Zaoral.

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

References

[1] C. Lattner, 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), IEEE Computer Society, 2004, pp. 75–88.

[2] LLVM project webpages. URL https://llvm.org.

[3] M. Weiser, Program slicing, IEEE Trans. Softw. Eng. 10 (4) (1984) 352–357.

[4] O. Lhotak, Y. Smaragdakis, M. Sridharan, Pointer analysis (dagstuhl seminar 13162), Dagstuhl Rep. 3 (4) (2013) 91–113, http://dx.doi.org/10.4230/DagRep.3.4.91, URL http://drops.dagstuhl.de/opus/volltexte/2013/4169.

[5] M. Hind, A. Pioli, Which pointer analysis should I use? in: D.J. Richardson, M.J. Harold (Eds.), Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000, ACM, 2000, pp. 113–123, http://dx.doi.org/10.1145/347324.348916.

[6] D. Avots, M. Dalton, V.B. Livshits, M.S. Lam, Improving software security with a C pointer analysis, in: G. Roman, W.G. Griswold, B. Nuseibeh (Eds.), 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, ACM, 2005, pp. 332–341, http://dx.doi.org/10.1145/1062455.1062520.

[7] S.Z. Guyer, C. Lin, Client-driven pointer analysis, in: R. Cousot (Ed.), Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings, in: Lecture Notes in Computer Science, vol. 2694, Springer, 2003, pp. 214–236, http://dx.doi.org/10.1007/3-540-44898-5_12.

[8] S.Z. Guyer, C. Lin, Error checking with client-driven pointer analysis, Sci. Comput. Program. 58 (1–2) (2005) 83–114, http://dx.doi.org/10.1016/j.scico.2005.02.005.

[9] C.L. Conway, D. Dams, K.S. Namjoshi, C.W. Barrett, Pointer analysis, conditional soundness, and proving the absence of errors, in: M. Alpuente, G. Vidal (Eds.), Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings, in: Lecture Notes in Computer Science, vol. 5079, Springer, 2008, pp. 62–77, http://dx.doi.org/10.1007/978-3-540-69166-2_5.

[10] C. Lattner, V.S. Adve, Automatic pool allocation: improving performance by controlling data structure layout in the heap, in: V. Sarkar, M.W. Hall (Eds.), Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, ACM, 2005, pp. 129–142, http://dx.doi.org/10.1145/1065010.1065027.

[11] A. Milanova, A. Rountev, B.G. Ryder, Precise call graph construction in the presence of function pointers, in: 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), 1 October 2002, Montreal, Canada, IEEE Computer Society, 2002, pp. 155–162, http://dx.doi.org/10.1109/SCAM.2002.1134115.

[12] Y. Sui, J. Xue, SVF: interprocedural static value-flow analysis in LLVM, in: A. Zaks, M.V. Hermenegildo (Eds.), Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016, ACM, 2016, pp. 265–266, http://dx.doi.org/10.1145/2892208.2892235.

[13] M.G. Burke, R. Cytron, Interprocedural dependence analysis and parallelization (with retrospective), in: K.S. McKinley (Ed.), 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection, ACM, 1986, pp. 139–154, http://dx.doi.org/10.1145/989393.989411.

[14] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (3) (1987) 319–349.

[15] P. Petersen, D.A. Padua, Static and dynamic evaluation of data dependence analysis techniques, IEEE Trans. Parallel Distrib. Syst. 7 (11) (1996) 1121–1132, http://dx.doi.org/10.1109/71.544354.

[16] T.B. Tok, S.Z. Guyer, C. Lin, Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers, in: A. Mycroft, A. Zeller (Eds.), Compiler Construction, 15th International Conference, CC 2006, Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings, in: Lecture Notes in Computer Science, vol. 3923, Springer, 2006, pp. 17–31, http://dx.doi.org/10.1007/11688839_3.

[17] R. Castillo, F. Corbera, A.G. Navarro, R. Asenjo, E.L. Zapata, Complete def-use analysis in recursive programs with dynamic data structures, in: E. César, M. Alexander, A. Streit, J.L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, S. Jha (Eds.), Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 5415, Springer, 2008, pp. 273–282, http://dx.doi.org/10.1007/978-3-642-00955-6_32.

[18] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, Z. Yang, Dependence guided symbolic execution, IEEE Trans. Softw. Eng. 43 (3) (2017) 252–271, http://dx.doi.org/10.1109/TSE.2016.2584063.

[19] V.P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, M.B. Dwyer, A new foundation for control dependence and slicing for modern program structures, ACM Trans. Program. Lang. Syst. 29 (5) (2007) 27, http://dx.doi.org/10.1145/1275497.1275502.

[20] S. Danicic, R.W. Barraclough, M. Harman, J. Howroyd, Á. Kiss, M.R. Laurence, A unifying theory of control dependence and its application to arbitrary program structures, Theoret. Comput. Sci. 412 (49) (2011) 6809–6842, http://dx.doi.org/10.1016/j.tcs.2011.08.033.

[21] S. Horwitz, T.W. Reps, D.W. Binkley, Interprocedural slicing using dependence graphs, ACM Trans. Program. Lang. Syst. 12 (1) (1990) 26–60.

[22] E.F. Najumudheen, R. Mall, D. Samanta, A dependence graph-based representation for test coverage analysis of object-oriented programs, ACM SIGSOFT Softw. Eng. Notes 34 (2) (2009) 1–8, http://dx.doi.org/10.1145/1507195.1507208.

[23] K. Hotta, Y. Higo, S. Kusumoto, Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph, in: T. Mens, A. Cleve, R. Ferenc (Eds.), 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012, IEEE Computer Society, 2012, pp. 53–62, http://dx.doi.org/10.1109/CSMR.2012.16.

[24] M. Chalupa, M. Jonás, J. Slaby, J. Strejcek, M. Vitovská, Symbiotic 3: New slicer and error-witness generation - (competition contribution), in: M. Chechik, J. Raskin (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, the Netherlands, April 2-8, 2016, Proceedings, in: Lecture Notes in Computer Science, vol. 9636, Springer, 2016, pp. 946–949, http://dx.doi.org/10.1007/978-3-662-49674-9_67.

[25] M. Chalupa, Slicing of LLVM Bitcode (Master's thesis), Masaryk University, Faculty of Informatics, Brno, 2016, URL https://is.muni.cz/th/vik1f/.

[26] M. Chalupa, J. Strejcek, M. Vitovská, Joint forces for memory safety checking, in: M. Gallardo, P. Merino (Eds.), Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings, in: Lecture Notes in Computer Science, vol. 10869, Springer, 2018, pp. 115–132, http://dx.doi.org/10.1007/978-3-319-94111-0_7.

[27] M. Chalupa, J. Strejček, Evaluation of program slicing in software verification, in: W. Ahrendt, S.L.T. Tarifa (Eds.), IFM'19, in: Lecture Notes in Computer Science, vol. 11918, Springer, 2019, pp. 101–119, http://dx.doi.org/10.1007/978-3-030-34968-4_6.

[28] M. Chalupa, J. Strejček, M. Vitovská, Joint forces for memory safety checking revisited, Int. J. Softw. Tools Technol. Transf. 22 (2) (2020) 115–133, http://dx.doi.org/10.1007/s10009-019-00526-2.

[29] Red hat research - AUFOVER project, 2020, URL https://research.redhat.com/blog/research_project/aufover-2/.

[30] M. Chalupa, M. Vitovská, M. Jonás, J. Slaby, J. Strejcek, Symbiotic 4: Beyond reachability - (competition contribution), in: A. Legay, T. Margaria (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II, in: Lecture Notes in Computer Science, vol. 10206, 2017, pp. 385–389, http://dx.doi.org/10.1007/978-3-662-54580-5_28.

[31] M. Chalupa, M. Vitovská, J. Strejček, Symbiotic 5: Boosted instrumentation (competition contribution), in: D. Beyer, M. Huisman (Eds.), Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), in: Lecture Notes in Computer Science, vol. 10806, Springer, 2018, pp. 442–446.

[32] M. Chalupa, M. Vitovská, T. Jašek, M. Šimácek, J. Strejcek, SYMBIOTIC 6: generating test cases by slicing and symbolic execution, Int. J. Softw. Tools Technol. Transf. (2020) http://dx.doi.org/10.1007/s10009-020-00573-0.

[33] M. Chalupa, T. Jašek, L. Tomovič, M. Hruška, V. Šoková, P. Ayaziová, J. Strejek, T. Vojnar, Symbiotic 7: Integration of predator and more (competition contribution), in: A. Biere, D. Parker (Eds.), Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'20), in: Lecture Notes in Computer Science, vol. 12079, Springer, 2020, pp. 413–417, http://dx.doi.org/10.1007/978-3-030-45237-7_31.

[34] D. Trabish, A. Mattavelli, N. Rinetzky, C. Cadar, Chopped symbolic execution, in: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (Eds.), ICSE'18, ACM, 2018, pp. 350–360, http://dx.doi.org/10.1145/3180155.3180251.

[35] M. Ahmadvand, A. Hayrapetyan, S. Banescu, A. Pretschner, Practical integrity protection with oblivious hashing, in: ACSAC'18, ACM, 2018, pp. 40–52, http://dx.doi.org/10.1145/3274694.3274732.

[36] L. Cheng, Program Anomaly Detection Against Data-Oriented Attacks (Ph.d. thesis), Virginia Tech, 2018, URL https://vtechworks.lib.vt.edu/handle/10919/84937.

[37] H. Hamadeh, A. Almomani, A. Tyagi, Probabilistic verification of outsourced computation based on novel reversible PUFs, in: A. Brogi, W. Zimmermann, K. Kritikos (Eds.), ESOCC'20, in: Lecture Notes in Computer Science, vol. 12054, Springer, 2020, pp. 30–37, http://dx.doi.org/10.1007/978-3-030-44769-4_3.

[38] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, H. Jin, VulDeeLocator: A deep learning-based fine-grained vulnerability detector, 2020, CoRR abs/2001.02350. arXiv:2001.02350. URL http://arxiv.org/abs/2001.02350.

[39] L. Cheng, K. Tian, D.D. Yao, Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks, in: ACSAC'17, ACM, 2017, pp. 315–326, http://dx.doi.org/10.1145/3134600.3134640.

[40] L. Cheng, K. Tian, D. Yao, L. Sha, R.A. Beyah, Checking is believing: Event-aware program anomaly detection in cyber-physical systems, 2018, CoRR abs/1805.00074. arXiv:1805.00074.

[41] B. Deng, W. Wu, L. Song, Redundant logic elimination in network functions, in: A. Wang, E. Rozner, H. Zeng (Eds.), SOSR'20, ACM, 2020, pp. 34–40, http://dx.doi.org/10.1145/3373360.3380832.