Who's Debugging the Debuggers?
Exposing Debug Information Bugs in Optimized Binaries
Extended Abstract

Giuseppe Antonio Di Luna[1], Davide Italiano [2], Luca Massarelli[1], Sebastian Österlund[3], Cristiano Giuffrida[3], Leonardo Querzoni[1]

1: Sapienza, University; 2: Apple; 3: Vrije Universiteit Amsterdam.

1. Motivation

Production software is often optimized by compilers, which use sophisticated optimization techniques to improve several aspects of the produced artifacts (e.g., speed, size, energy consumption [19], etc.). While these optimizations are fundamental to ensure the final binary performance, it is well known that they may expose bugs that are observable only in the optimized case (e.g., race conditions [9], use-after-free [5], and other classes of heisenbugs [25]). Hence, there is often a need for debugging the exact same version of the binary running in production to triage issues that are otherwise not reproducible. A complete and reliable debugging experience for optimized binaries [6] is thus crucial for post-deployment triaging efforts.

Note that a reliable debugging experience not only requires the debugger to behave correctly, but it also depends on the compiler's ability to produce precise debug information. In other words, the *entire toolchain* has to be bug-free. However, preserving debug information for optimized production binaries is a daunting task. There is no obvious mapping between source and assembly statements [6] and there is potential to introduce bugs at each of the several layers of modern toolchains. As we will show, even efforts to provide debug-friendly optimization levels (such as *-Og* in modern compilers) fall short of providing a bug-free debugging experience.

Therefore, it is crucial to analyze the entire debug information lifecycle and look for bugs that could negatively impact the debugging experience.

A motivating example — Snippet 1 shows a bug we exposed with our framework in LLVM when compiling with *-Og*.

```
1
2  int a, b, c;
3  int main()
4  {
5    {int ui1 = 5, ui2 = b;
6      c =
7        ui2 == 0 ?
8        ui1 :
9        (ui1 / ui2);
10   }
11 }
```

Snippet 1: Clang bug, wrong step at line 8 (-*Og*)

The bug is observable as an incorrect line stepping in the debugger, but it is a compiler bug. More precisely, while stepping, the debugger points to a source line that is dead code: in this case, it points to line 8, which does not get executed, instead of line 7 (note that variables `a,b,c` are static and initialized to 0). This and similar bugs significantly degrade the debugging experience by showing misleading execution flows. Since source line stepping is a staple of debugging and other tools (e.g., profilers), the example highlights the need for more rigorous testing of debug information for optimized code.

As a matter of fact, such a bug is an instance of a more general class of bugs in which transformation passes move instructions across the control flow graph blocks without a proper update of the associated debug information. In this particular example, the branch folding pass (which is responsible for optimizing the control flow) does not preserve the correct line information. In this work, we found several other problems in passes that perform similar transformations (e.g., passes that perform simplifications of the control flow graph and passes moving invariants instructions outside of a loop).

2. Limitations of the State of the Art

Surprisingly, testing the debug lifecycle of optimized binaries has been mostly neglected (despite much prior work on compiler testing, for a survey see [3]). The majority of the efforts [6, 23, 7, 8, 20] studying the debugging of optimized binaries focused on generating debug information or creating debugging strategies without looking at the correctness of real-world production toolchains.

Some efforts have focused on debugger testing [21, 14], but these solutions cannot find bugs in other parts of the toolchain (e.g., the compiler).

Most related to our work is a recent work by Yuanbo Li et al. [15]. The paper focuses on assessing the correctness of the variables' values shown by a debugger for optimized code of statically compiled languages, but does not consider other classes of bugs (e.g., line information bugs). Furthermore, the authors assume that source-line stepping is correct, which, as shown in our motivating example, is not always true.

3. Key Insights

This paper introduces DEBUG^2, a framework to expose debug information bugs in production toolchains. We tested DEBUG^2 on various toolchains, and found several new bugs like the one presented in the motivating example.

Key insights: our analysis shows that after decades of development, mature compiler (e.g., clang and GCC) and debugger implementations still have a conspicuous amount of bugs that can be found automatically. Surprisingly, many of these bugs plague optimization levels specifically created for a smooth debugging experience (i.e., *-Og*).

Beyond the state of the art: our framework is much more general than previous efforts [15, 21, 14], being able to find sev-

eral classes of bugs in the entire debug lifecycle of optimized binaries. Specifically, DEBUG^2 is not limited to discovering specific bugs in the debugger (as [21, 14]). It relies on generic cross-toolchain and cross-language invariants to find a rich family of bugs (differently from [15]) in the entire toolchain, including wrong source-line stepping and wrong backtraces.

4. Main Artifacts

- We introduce DEBUG^2 (see Full Paper §2). The framework feeds random source programs to a target toolchain and performs a differential analysis on the debugging behavior of their optimized/unoptimized binaries to expose bugs. To automatically check for (un)expected behavioral differences, DEBUG^2 relies on *trace invariants* based on the (in)consistency of common debug elements, such as source lines, stack frames, and function arguments.
- We introduce four different trace invariants that check different debug information (see Full Paper §3):
  The Line Invariant (LI) checks for misstepped lines;
  The Backtrace Invariant (BI) checks for spurious frames in the backtrace of a line;
  The Scope Invariant (SI) checks if there are out-of-scope variables;
  The Parameters Invariants (PI) checks the consistency of the values assumed by function parameters.
- The extensive evaluation presented in the paper positively answers the following questions (see Full Paper §3):
  – Does DEBUG^2 find real bugs in our reference toolchain (LLVM) across different optimization levels and components (e.g., compiler and debugger)?
  – Does DEBUG^2 generalize to different toolchains and programming languages?
- We answered our research questions by evaluating a python prototype of DEBUG^2. In roughly three months, we found 23 bugs on the trunk versions of the LLVM toolchain (from April to mid-July). Our methodology was able to find several violations of our invariants, which we automatically triaged and manually analyzed to identify bugs (see Full Paper §5.2). We studied the relationship between invariant violations and bugs. Our results showed that more than 99% of violations were caused by bugs (see Full Paper §5.1). We also performed additional experiments to identify the optimization passes that are more prone to bugs. We ran similar tests on the GNU toolchain and Rust toolchain aimed at investigating the generality of DEBUG^2 across different toolchains and programming languages (see Full Paper §5.3).

5. Key Results and Contributions

We make the following key contributions:

- DEBUG^2, a framework to scrutinize the debug information lifecycle for optimized binaries using trace invariants.
- Using DEBUG^2 on the tested toolchains, we exposed and reported 23 bugs for the LLVM toolchain (clang and lldb), 8 bugs for the GNU toolchain (GCC and gdb) and 3 bugs on the Rust toolchain (rustc and lldb). The developers confirmed 22 bugs, 14 of which have been fixed.

The bugs we reported sparked much discussion among developers, leading to interesting lessons learned:

- There are no formal guidelines to define how debug information should be preserved during optimization passes of modern toolchains. This led developers to adopt a case-by-case approach to decide how fix our bugs. Such strategy, in the long term, is likely to lead to inconsistent behavior. After reviewing our findings, the LLVM developers published a set of basic guidelines to update debug information during certain optimization passes [12]. This is a concrete outcome of our effort and a first important step towards defining shared guidelines.
- We discovered some shortcomings of DWARF, the de-facto standard used to encode debug information for UNIX platforms. Notably, DWARF does not provide a way to map a single address to multiple source locations, making it impossible to represent the result of some optimization passes correctly (e.g., common subexpression elimination).

6. Why ASPLOS

Debugging, reproducing, and triaging bugs are all well-established and long-studied problems in the systems community [22, 16, 1, 11, 10, 2]. In previous efforts, the main focus has been on improving the reproducibility of production bugs, assuming correct debug information. With DEBUG^2, we show how bugs in real-world toolchains might provide the developer with invalid debug information, making it impossible to triage or reproduce a particular production bug. In other words, DEBUG^2 complements prior work in the systems community on improving debuggability. At the same time, DEBUG^2 makes important contributions to the field of compiler/toolchain testing, which is on the critical path of the PL community [15, 17, 24, 13]. Finally, DEBUG^2's generic bug detection strategy based on differential invariants draws inspiration from much prior work on invariants-based bug detection published at ASPLOS (e.g., [18, 4]).

7. Citation for Most Influential Paper Award

This paper introduced DEBUG^2, the first framework to scrutinize the entire debug information lifecycle for optimized programs. DEBUG^2 uses trace invariants to perform differential analysis on the debug traces produced by optimized and unoptimized binaries. Invariant violations expose bugs that affect the debug lifecycle and that may reside in different parts of the toolchain. DEBUG^2 helped discover and fix many bugs affecting widely used toolchains for several years, severely impacting the debugging experience on production software.

References

[1] Joy Arulraj, Guoliang Jin, and Shan Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *ASPLOS '14: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 207–222, 2014.

[2] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 1073–1086, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[4] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 348–362, New York, NY, USA, 2018. Association for Computing Machinery.

[5] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *2015 IEEE Security and Privacy Workshops*, pages 73–87, 2015.

[6] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):323–344, 1982.

[7] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43, 1992.

[8] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Fulldoc: A full reporting debugger for optimized code. In *International Static Analysis Symposium*, pages 240–259. Springer, 2000.

[9] C. Jia and W. K. Chan. Which compiler optimization options should i use for detecting data races in multithreaded programs? In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 53–56, 2013.

[10] Mark Scott Johnson. Some requirements for architectural support of software debugging. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, ASPLOS I, page 140–148, New York, NY, USA, 1982. Association for Computing Machinery.

[11] Omer Katz, Noam Rinetzky, and Eran Yahav. Statistical reconstruction of class hierarchies in binaries. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 363–376, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Vedant Kumar. How to update debug info: A guide for llvm pass authors. https://github.com/llvm/llvm-project/blob/master/llvm/docs/HowToUpdateDebugInfo.rst, 2020. [Online; accessed 27-July-2020].

[13] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

[14] Daniel Lehmann and Michael Pradel. Feedback-directed differential testing of interactive debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 610–620, 2018.

[15] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. Debug information validation for optimized code. In *PLDI*, pages 1052–1065, 2020.

[16] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *ASPLOS '17: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[17] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA, 2012. Association for Computing Machinery.

[18] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. Association for Computing Machinery.

[19] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. *ACM SIGARCH Computer Architecture News*, 42(1):639–652, 2014.

[20] Caroline Tice and Susan L Graham. Optview: a new approach for examining optimized code. *ACM SIGPLAN Notices*, 33(7):19–26, 1998.

[21] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283, 2019.

[22] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 15–26, New York, NY, USA, 2011. Association for Computing Machinery.

[23] Roland Wismüller. Debugging of globally optimized programs using data flow analysis. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 278–289, 1994.

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[25] Jie Yin, Gang Tan, Hao Li, Xiaolong Bai, Yu-Ping Wang, and Shi-Min Hu. Debugopt: Debugging fully optimized natively compiled programs using multistage instrumentation. *Science of Computer Programming*, 169:18 – 32, 2019.