LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation*
(Competition Contribution)

Carsten Sinz, Florian Merz, and Stephan Falke

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{carsten.sinz, florian.merz, stephan.falke}@kit.edu

Abstract. We present LLBMC, a bounded model checker for C programs. LLBMC uses the LLVM compiler framework in order to translate C programs into LLVM's intermediate representation (IR). The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. In contrast to many other tools, LLBMC uses a flat, bit-precise memory model. It can thus precisely model, e.g., memory-based re-interpret casts.

1 Verification Approach

Bounded model checking (BMC) has proven to be a very successful technique in hardware verification. More recently, it has also been applied for verifying software written in C [1, 4]. Applying BMC for verifying C programs, however, comes with many obstacles that have to be tackled. One of the most important differences is that the syntax and semantics of a programming language like C is much more complicated than a hardware description. One has to deal, e.g., with memory allocation and de-allocation, (function) pointers, complex data structures, and function calls.

LLBMC uses an approach which, instead of exploring the source code directly, makes use of existing compiler technology and performs the analysis on a compiler intermediate representation. Such an intermediate representation offers a much simpler syntax and semantics than a programming language like C, and thus eases a logical encoding of the verification problem considerably.

We have chosen the LLVM [5] compiler infrastructure and its assembler-like intermediate representation as the starting point for our approach, but the idea can also be applied to other low-level languages. LLVM is both a (GCC-compatible) C/C++/Objective-C compiler and a library of compiler technologies, providing, e.g., source- and target-independent optimizations.

Our primary goal is to detect memory errors in C code [7, 2, 6]. Memory errors include invalid memory accesses, heap and stack buffer overflows, and invalid frees (e.g., double frees).

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

2 Software Architecture

While LLBMC is designed for C programs, its input format is LLVM-IR, the intermediate representation of the LLVM compiler framework. LLVM-IR is an abstract assembler language that is programming-language-independent. This makes it easier to extend LLBMC to other languages supported by LLVM (like C++ or Objective-C). Furthermore, the challenges in parsing complex high-level language syntax, such as C++, are eliminated. Instead, only a limited instruction set needs to be supported. LLVM-IR is architecture-dependent in the sense that the compiler frontend selects, e.g., the bitwidth of pointers and integer data types.

After reading in the LLVM-IR code, LLBMC applies a number of transformations to it. In particular, loops are unrolled, functions are inlined, and the control flow graph is simplified. The transformed code is then converted to ILR, which is a representation of a program in the logic of bit-vectors and arrays plus some extensions, related to memory allocation. ILR provides an explicit state object for the memory content as well as for the state of the memory allocation system. These state objects encode the dependencies between memory accessing instructions in the ILR formula. Because of this, dependencies between instructions in LLVM, which were implicitly given by the ordering of the read and write operations are made explicit in the ILR formula. This change makes the expressions in an ILR formula ordering-independent. The ILR formula is then simplified using rewrite rules, and memory access correctness expressions are reduced to bit-vector formulas (see [2, 7] for details). If no more rewrite rules can be applied, the formula is passed to the SMT solver STP [3].

3 Strengths and Weaknesses of the Approach

LLBMC is tailored towards finding bugs in C programs, especially memory-related ones (not so much towards proving their absence). Detectable errors include:

- arithmetic overflow and underflow, including shift overflow,
- invalid memory access operations,
- invalid memory allocation, including invalid `free`s, and
- overlapping memory regions in `memcpy`.

Furthermore, LLBMC supports checking of user assertions and reachability of labels named "ERROR" in the C-code. It can also detect whether the loop unrolling and function inlining bound was sufficient or has to be increased in order to achieve full coverage.

In the competition, LLBMC was used with a fixed unwinding bound of 7 and an automatically determined function inlining bound. It was not checked whether the unwinding bound is sufficient, but only whether the "ERROR" label was reachable within these bounds (as other comparable tools have chosen similar settings). If no error was found, the instance was considered safe. LLBMC was able to successfully handle 146 out of 269 benchmark instances (not participating in category "Concurrency", as this is not supported by LLBMC), resulting in a

first place in category "Device Drivers" and a second place in category "Heap Manipulation". Among the unsolved instances, 65 were due to time-outs, and 48 due to current restrictions of LLBMC (e.g., related to memcpy or inline assembly). LLBMC produced 7 false positives and 3 false negatives. The false negatives (i.e. where an error was missed) were due to an insufficient loop unrolling bound. Among the 7 false positives, one was due to an error in LLBMC related to detecting a malloc function. The other 6 were due to uninitialized pointer variables, by which other (e.g., global) variables could be overwritten and thus be modified, resulting in the "ERROR" label becoming reachable. We do not consider these errors as "false positives", but see here a special strength of LLBMC and its precise memory model, as such errors are very hard to detect and, in practice, result in non-deterministic program behavior.

4 Tool Setup and Configuration

The version of LLBMC (0.9) submitted to TACAS can be downloaded from

http://llbmc.org/llbmc-tacas12.zip.

LLBMC requires llvm-gcc (version 2.9) in order to convert C input files to LLVM's intermediate representation. For instructions on how to use LLBMC, just enter llbmc --help. The ZIP archive also contains two wrapper shell scripts to run LLBMC on individual C files. The first, llbmcc, iteratively increases the loop unwind bound and also checks whether the unwind bound is sufficient. The second, llbmcc2, which was used in the competition, also increases the unwind bound, but only up to a maximal value of 7, and does not perform unwind bound checks. Both shell scripts compile the C program, run LLBMC, and perform only a reachability check for a basic block labelled "ERROR", but no other checks, such as for invalid memory accesses. They output either SAFE, if the error label is unreachable (within the given bound for llbmcc2), or UNSAFE otherwise.

Further information on LLBMC is available on the web at http://llbmc.org.

References

1. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS 2004. LNCS, vol. 2988, pp. 168–176 (2004)
2. Falke, S., Merz, F., Sinz, C.: A theory of C-style memory allocation. In: Proc. SMT 2011. pp. 71–80 (2011)
3. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proc. CAV 2007. LNCS, vol. 4590, pp. 519–531 (2007)
4. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. TCS 404(3), 256–274 (2008)
5. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO 2004. pp. 75–88 (2004)
6. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Proc. VSTTE 2012. LNCS (2012)
7. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: Proc. SSV 2010 (2010)