Map2Check Using LLVM and KLEE
(Competition Contribution)

Rafael Menezes[1], Herbert Rocha[1(✉)], Lucas Cordeiro[2], and Raimundo Barreto[3]

[1] Department of Computer Science, Federal University of Roraima, Boa Vista, Brazil
herberthb12@gmail.com
[2] Department of Computer Science, University of Oxford, Oxford, UK
[3] Institute of Computing, Federal University of Amazonas, Manaus, Brazil

Abstract. Map2Check is a bug hunting tool that automatically checks safety properties in C programs. It tracks memory pointers and variable assignments to check user-specified assertions, overflow, and pointer safety. Here, we extend Map2Check to: (i) simplify the program using Clang/LLVM; (ii) perform a path-based symbolic execution using the KLEE tool; and (iii) transform and instrument the code using the LLVM dynamic information flow. The SVCOMP'18 results show that Map2Check can be effective in generating and checking test cases related to memory management of C programs.

1 Overview

Map2Check v7.1 uses source code instrumentation based on dynamic information flow, to monitor data from different program executions. Map2Check automatically produces concrete inputs to the program via symbolic execution, in order to execute different program paths and to detect failures related to arithmetic overflow, invalid deallocation, invalid pointers, and memory leaks. Map2Check uses Clang [5] as a front-end, which supports the main C standard, e.g., C99 according to the standard ISO/IEC 9899:1990. In its previous version [7], Map2Check was able to automatically generate test cases to check memory management using bounded model checkers (e.g., ESBMC [4]). The main original contributions of Map2Check v7.1 are: (i) added Clang [5] as a front-end to improve the symbolic execution of C programs; (ii) adopted the LLVM [6] framework as a code transformation engine; and (iii) integrated the KLEE [1] tool as a symbolic execution engine to automatically explore different program paths.

2 Verification Approach

The Map2Check tool is inspired by LEAKPOINT [3] and Symbiotic 4 [2], which use compiler techniques to analyze C programs using code instrumentation. The

H. Rocha—Jury member.

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10806, pp. 437–441, 2018.
https://doi.org/10.1007/978-3-319-89963-3_28

main novelty of Map2Check v7.1 is the integration of the LLVM Intermediate Representation (IR) to analyze and verify C programs. This LLVM IR is based on the static single assignment representation and provides type safety, low-level operations, and the capability of representing high-level languages. If we compare Map2Check to other related tools, e.g., Symbiotic 4, it does not perform static program slicing and does not use the symbolic execution of KLEE to directly explore the program state space. Map2Check applies source code instrumentation to monitor and gather areas of data memory from different concrete program executions; this code instrumentation focuses on exploring dynamic information flow to avoid the need for an approximate static analysis. Similarly to LEAKPOINT, Map2Check taints program data (e.g., variables or memory locations) with a taint mark metadata and then propagates the taint marks over the concrete program executions. Fig. 1 shows an overview of the Map2Check verification flow. The tool input is a C program and a safety property (e.g., overflow and pointer safety); it returns *TRUE* (if there is no path that violates the safety property), *FALSE* (if there exists a path that violates the safety property), or *UNKNOWN* otherwise.

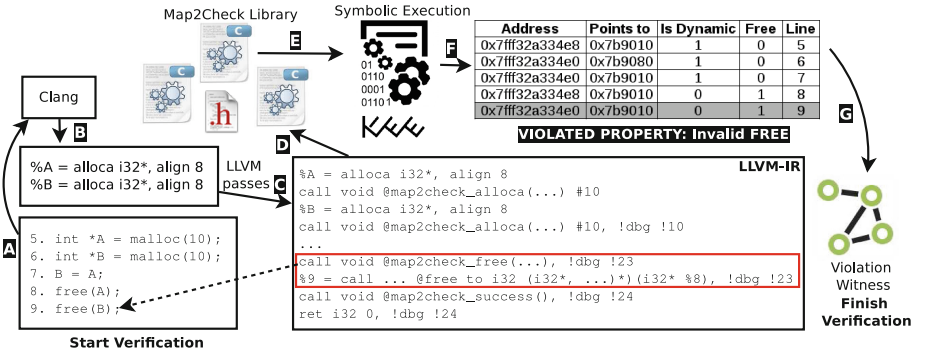

Fig. 1. Map2Check verification flow.

The Map2Check verification flow has the following main steps: (A) convert the C code into the LLVM IR using Clang [5]; (B) apply specific code optimizations, e.g., dead code elimination and constant propagation; (C) add Map2Check library functions to track pointers, and add assertions into the LLVM bitcode; (D) connect the code instrumented by Map2Check to support the execution of its functions; (E) apply further Clang optimizations to improve the symbolic execution (e.g., canonicalize natural loops and promote memory to register); (F) generate concrete inputs for the Map2Check instrumented functions by performing symbolic execution of the analyzed code in LLVM IR using KLEE; and (G) generate witnesses: if a safety property is violated, then a "violation witness" is produced using the KLEE output to trace the error location; if there is no path that violates the safety property, then a "correctness witness" is produced,

which identifies each basic block executed in the control flow graph of the LLVM IR using the concrete inputs produced by KLEE (LLVM syntactically enforces some of those basic blocks as invariants from its assignments).

Map2Check v7.1 tracks important data of the analyzed C code to identify functions and operations over pointers. Then, it checks the respective assertions via symbolic execution, which produces inputs to concretely execute the program. In particular, Map2Check tracks the heap memory used by the analyzed code using the following data log lists: **Heap log** tracks the allocated memory address (i.e., arguments of functions, functions, and variables) and its memory size in the heap memory; **Malloc log** tracks the addresses that are dynamically allocated/deallocated, their size and pointer actions (allocation and deallocation), executed at the current program location; and **List log** stores data about operations over pointers, e.g., the code line number for each operation, program scope, variable name, memory addresses, and addresses pointed to by program variables.

Map2Check v7.1 implements a function `map2check_non_det_x` with `x` in the supported C data types (e.g., char, int, and float), which is interpreted by KLEE to model non-deterministic values. In this respect, Map2Check v7.1 differs from its previous version, which implements for non-deterministic values, a function that returns a random number based on a probabilistic distribution. To check the unreachability of an error location, Map2Check identifies a given target function (e.g., `__VERIFIER_error`) and then replaces that by an error assertion, where the target function is called. To check overflow, Map2Check adds an assertion before all arithmetic instructions over integers to analyze the results over the signed operations and the maximum and minimum integer values. To check pointer safety, Map2Check checks whether a given address to be deallocated is tracked in the Malloc log list and then identifies whether the deallocation of memory was already performed for that program location (invalid deallocation); Map2Check also identifies whether allocated memory was not released at the end of the program execution (memory leak); Additionally, Map2Check analyzes the memory addresses in the Malloc log and Heap log lists to identify if those addresses point to a valid address (invalid pointer). Map2Check does not distinguish between the usual "valid-memtrack" and "valid-memclean" properties in SV-COMP.

3 Proposed Architecture

Map2Check v7.1 is implemented as a source-to-source transformation tool in C/C++ using LLVM (v3.8.1). It uses Clang (v3.8.1) as a front-end to parse a C program and to generate the respective LLVM bitcode to be used in the code transformation to track pointers to areas of memory and variable assignments (Fig. 2). It uses KLEE (v1.2.0) as a path-based symbolic execution engine; STP[1] (v2.1.2) is used as the SMT solver by KLEE to check constraints over bit-vectors and arrays. The Boost[2] C++ library is used as a helper library,

[1] http://stp.github.io.
[2] http://www.boost.org.

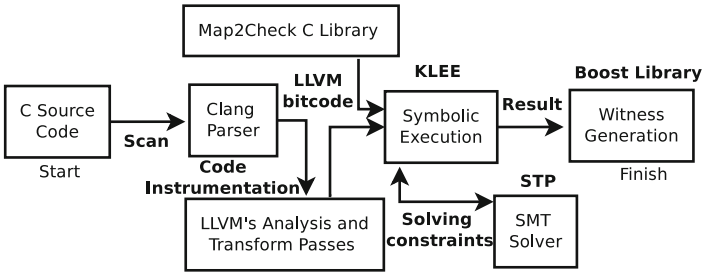

Fig. 2. Map2Check architecture flow.

e.g., to generate the witness in the GraphML format. Map2Check participates in SVCOMP'18 (as in the map2check.xml benchmark definition) in the following categories: ReachSafety-Arrays, ReachSafety-BitVectors, ReachSafety-Heap, ReachSafety-Loops, ReachSafety-Recursive, MemSafety, and NoOverflows.

3.1 Availability and Installation

Map2Check v7.1 (for 64-bit Linux) is available[3] under the GPL license. The Clang, LLVM, KLEE, and STP tools are included in the Map2Check distribution. Map2Check is invoked via a command-line (as in the map2check.py module for BenchExec) as:

```
./map2check-wrapper.py -p propertyFile.prp file.i
```

Map2Check accepts the property file and the verification task and provides as result: *TRUE + Witness, FALSE + Witness, or UNKNOWN*. For each error-path or correctness witness, a file (called `witness.graphml`) with the witness proof is generated in the Map2Check root-path folder.

4 Strengths and Weaknesses of the Approach

Map2Check exploits dynamic information flow by tainting program data. It uses Clang/LLVM as an industrial-strength compiler to simplify and instrument the code; and also employs KLEE to produce concrete inputs for different program executions. The integration between LLVM and KLEE opens up several possibilities to implement new testing and verification techniques in Map2Check. Particularly, we intend to improve our symbolic execution by synthesizing inductive invariants to prove properties of loops and recursive programs and also to prune the search-space, given that Map2Check bounds the loops and recursion up to a given depth k. The SVCOMP'18 results show that Map2Check can be effective in generating and checking test cases of memory management for C programs. Map2Check achieved a score of 228 in the MemSafety category with no single

[3] https://github.com/hbgit/Map2Check/archive/map2check_v7.1_svcomp18d.zip.

incorrect result; in particular, Map2Check produced the highest score (i.e., 106) in the MemSafety-Arrays subcategory. In the NoOverflows category, Map2Check achieved a score of −263; some incorrect results are due to our imprecise overflow check. In the ReachSafety category, we noted that Map2Check claims 312 correct results; however, it reported 16 incorrect true and 1 incorrect false. Some of these incorrect results are related to Map2Check limitation to handle loops and recursion.

Acknowledgments. We thank C. Cadar, D. Poetzl, and the anonymous reviewers for their comments, which helped us to improve the draft version of this paper.

References

1. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224. USENIX (2008)
2. Chalupa, M., Vitovská, M., Jonáš, M., Slaby, J., Strejček, J.: Symbiotic 4: beyond reachability. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 385–389. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_28
3. Clause, J., Orso, A.: LEAKPOINT: pinpointing the causes of memory leaks. In: ICSE, pp. 515–524. ACM (2010)
4. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: TSE, pp. 957–974. IEEE (2012)
5. Fandrey, D.: Clang/LLVM maturity report. In: Computer Science Department, University of Applied Sciences Karlsruhe (2010). http://www.iwi.hs-karlsruhe.de
6. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88. IEEE (2004)
7. Rocha, H.O., Barreto, R.S., Cordeiro, L.C.: Hunting memory bugs in C programs with Map2Check. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 934–937. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_64