Hunting Memory Bugs in C Programs with Map2Check
(Competition Contribution)

Herbert O. Rocha[1(✉)], Raimundo S. Barreto[2], and Lucas C. Cordeiro[2]

[1] Federal University of Roraima, Boa Vista, Brazil
herberthb12@gmail.com
[2] Federal University of Amazonas, Manaus, Brazil
rbarreto@icomp.ufam.edu.br, lucasccordeiro@gmail.com

Abstract. Map2Check is a tool for automatically generating and checking unit tests for C programs. The generation of unit tests is based on assertions extracted from (memory) safety properties, which are generated by the ESBMC tool. In particular, Map2Check checks for SV-COMP invalid-free, invalid-dereference, and memory-leak properties in C programs.

1 Overview

Map2Check automatically generates and checks unit tests for C programs [1]. The unit test generation is based on assertions, which are extracted from the memory safety properties generated by ESBMC tool [2]. In particular, Map2Check checks for SV-COMP properties "invalid-free", "invalid-dereference", and "memory-leak". Map2Check adopts source code instrumentation to create test cases from those properties, and monitors data from program's executions, in order to detect failures originating from the execution of those (generated) test cases. Map2Check supports full C99, according to the standard ISO/IEC 9899:1990, and checks programs that make use of arrays, pointers, structs, unions, and dynamic memory allocation. ESBMC is adopted as a verification condition (VC) generator, which translates a program fragment and its correctness property into a logical formula that is automatically translated into a unit test. Map2Check does not require the user to annotate C programs with pre/post-conditions to generate that VCs.

2 Verification Approach

Map2Check executes seven steps to generate and check test cases related to memory safety in C programs as shown in Fig. 1. In step 1, Map2Check uses ESBMC to identify memory safety properties via the option `--show-claims`, which shows all safety properties that ESBMC automatically generates from the original C program. In ESBMC, a `claim` represents a safety property; examples

© Springer-Verlag Berlin Heidelberg 2016
M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 934–937, 2016.
DOI: 10.1007/978-3-662-49674-9_64

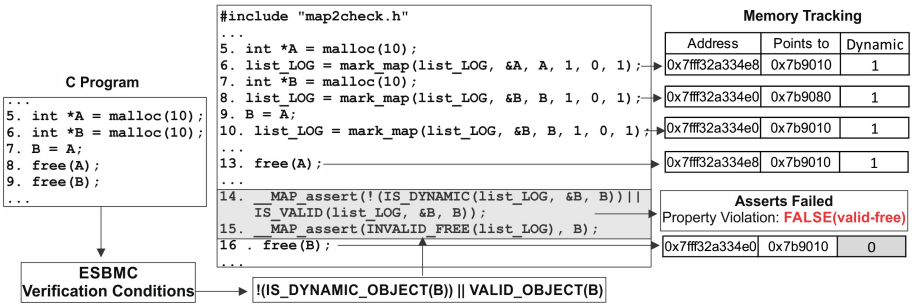

Fig. 1. Example of the Map2Check steps.

of claims include invalid-free, invalid-dereference, and memory-leaks; a particular claim can be violated by Map2Check if there is an execution that leads to the assertion failure.

In step 2, Map2Check analyzes the results produced in step 1 to collect several important pieces of information needed in the following steps, e.g., identification of the claim, comments about the claim, line number of the code where the claim occurred, and the property identified by that claim. For example, the particular claim !(IS_DYNAMIC_OBJECT(B)) || VALID_OBJECT(B) states a potential invalid dynamic object of "B", where an object can be represented by a pointer to a scalar variable or to a (more complex) data structure [2]. In particular, IS_DYNAMIC_OBJECT function checks whether the argument to any dereferencing operation is a dynamic object; and VALID_OBJECT(B) checks if the argument for any free or dereferencing operation is still a valid object. In Map2Check, we adopt regular expressions to find all claims information related to invalid-free, invalid-dereference, and memory-leak.

In step 3, Map2Check translates the claims provided by ESBMC into assertions written in C code, which are supported by a C library of Map2Check; this strategy is similar to that performed by Delahaye *et al.* [3], whose pre/post-conditions based on formal program specification are translated into C code via assertions.

In step 4, Map2Check performs a memory tracking, which consists of two phases:

1. Track program variable operations and assignments in the analyzed source code. Map2Check performs this tracking by means of the abstract syntax tree (AST), which is generated from the analyzed C program;
2. Instrument the source code with functions that monitor the memory addresses and the addresses pointed by these variables (identified in step 1) according to the program execution. The assertions generated in step 3 are checked over the data, which are generated by the functions that monitor the memory addresses.

In step 5, test cases are inserted into the program by adding assertions (generated from step 3) into the new copy of the source code (of the analyzed program), with their respective properties related to memory safety. In step 6, Map2Check applies a template over the analyzed program to allow the validation of the test cases and to insert directives of the Map2Check library into the new copy of the analyzed program. Map2Check also provides a template for the CUnit framework [5].

Finally, in step 7, Map2Check executes that new copy of the analyzed C program, together with the functions to monitor the memory addresses (added from step 4) and the test cases, in order to check each assertion. Instead of calling a theorem-prover, Map2Check executes the code to check whether the assertions fail. Map2Check provides a program execution trace log in case of the assertion violation (i.e., if the test case fails), with data such as: the line number, memory addresses, pointer actions (e.g., allocation and deadlocation) already executed at the current point of the program.

3 Strengths and Weaknesses of the Approach

Map2Check participates in the Heap Data Structures category only. The strength of the tool lies in the precision of its answers based on the concrete execution of the analyzed program over the VCs generated by ESBMC, i.e., ESBMC is adopted only as a VC generator and it is not used to formally verify the properties. In preliminary experiments, Map2Check outperforms ESBMC due to time-outs or memory model limitations. Map2Check is in the initial development and there are still restrictions on the structure of the programs (*e.g.*, the C *alloca* function is not supported) that can be analyzed by our memory tracking. Most incorrect answers produced by our tool are due to bugs in the implementation. Additionally, our strategy based on random data to unwind loops and their respective loop exit condition do not allow the correct execution of the program. In particular, we implement a specific function to simulate the non-deterministic values, which are generated from the function call `__nondet__int()`.

4 Architecture, Implementation and Availability

Architecture. Map2Check is implemented as a source-to-source transformation tool in Python (V2.7). It uses the pycparser[1] to parse a C program into an AST, and then identifies variables for tracking memory. The pyparsing[2] is used to create a parse of the ESBMC claims. It adopts uncrustify[3] as a source code beautifier. Map2Check also uses networkx[4] to generate the witness format[5] in GraphML format, and GCC compiler.

[1] https://github.com/eliben/pycparser.
[2] https://pyparsing.wikispaces.com.
[3] http://uncrustify.sourceforge.net.
[4] https://networkx.github.io.
[5] http://www.sosy-lab.org/~dbeyer/cpa-witnesses.

Availability and Installation. Map2Check source code version 6 for 64-bit Linux environment for the competition is available to freely download at https://github.com/hbgit/Map2Check under GPL license. It must be installed as a Python script and it also requires installation of pycparser, pyparsing, networkx, uncrustify, and GCC.

User Interface. Map2Check is invoked via a command-line interface to SV-COMP as follows: `./map2check-wrapper.sh -c propertyFile.prpfile.i`. Map2Check accepts the property file and the verification task and provides as verification result: *FALSE + Witness* or *UNKNOWN*. For each error-path, a file that contains the violation path is generated in Map2Check root-path *graphml* folder; this file has the same name of the verification task with the extension `graphml`.

References

1. Rocha, H., Barreto, R., Cordeiro, L.: Memory management test-case generation of C programs using bounded model checking. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 251–267. Springer, Heidelberg (2015)
2. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans. Soft. Eng. **38**(4), 957–974 (2012)
3. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: SAC, pp. 1230–1235 (2013)
4. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: POPL, pp. 193–205 (2001)
5. Rocha, H., Cordeiro, L., Barreto, R., Netto, J.: Exploiting safety properties in bounded model checking for test cases generation of C programs. In: SAST, pp. 121–130 (2010)