Predator Hunting Party (Competition Contribution)

Petr Muller, Petr Peringer, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. This paper introduces PredatorHP (Predator Hunting Party), a program verifier built on top of the Predator shape analyser, and discusses its participation in the SV-COMP'15 software verification competition. Predator is a sound shape analyser dealing with C programs with lists implemented via low-level pointer operations. PredatorHP uses Predator to prove programs safe while at the same time using several bounded versions of Predator for bug hunting.

1 The Underlying Verification Approach

At the heart of PredatorHP there is the Predator shape analyser [2]. The main aim of Predator is *sound* shape analysis of sequential, non-recursive C programs that use low-level pointer operations for working efficiently with various kinds of linked lists. Predator supports many advanced uses of pointer arithmetics, address alignment, and block operations common in highly optimized system code, such as operating system kernels, drivers, memory allocators, and the like.

Predator is based on abstract interpretation with the abstract domain of *symbolic memory graphs* (SMGs) [2]. In a nutshell, SMGs consist of two kinds of nodes—namely, individual memory regions and uninterrupted list segments—and two kinds of edges, in particular, the so-called has-value and points-to edges. SMGs were inspired by separation logic with higher-order list predicates but with an added support for low-level memory operations. Moreover, all the needed algorithms for dealing with SMGs (symbolic execution of program statements, the join operator, widening in the form of abstraction, entailment checking) were newly designed to be as efficient as possible by leveraging the graph structure of SMGs. The most essential role is played by the join operator: both abstraction and entailment checking are built on top of it. Predator supports inter-procedural analysis by means of function summaries.

Recently, a new extension of Predator was implemented [1]. It uses the Predator kernel for transforming programs with list containers implemented by low-level pointer operations into equivalent programs with high-level container operations, which can be useful, e.g., for code understanding, easier verification, parallelisation, optimisation, etc.

2 From Predator to Predator Hunting Party

Predator is implemented as a GCC plug-in, which provides it with an industrial-strength compiler front-end. In particular, GCC is used to pre-process the input programs and to compile them into an intermediate representation (known as GIMPLE), which is further transformed into a bit more concise representation of the Code Listener framework [3] over which Predator runs. Predator is written in C++ with a use of the Boost libraries, mainly to enable using legacy compilers for building it.

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 443–446, 2015.
DOI: 10.1007/978-3-662-46681-0_40

Predator requires all external functions used in an analysed program to be properly modelled wrt. memory safety in order to exclude any side effects that could possibly break soundness of the analysis. The distribution of Predator includes models of some memory manipulating functions (like `malloc`, `free`, `memset`, `memcpy`, etc.).

PredatorHP is implemented as a Python script which runs several instances of Predator in parallel and composes the results they produce into the final verification verdict. In particular, PredatorHP first starts four Predators: One of them is the original Predator that soundly over-approximates the behaviour of the input program—we denote it as the *Predator verifier* below. Apart from that, three further Predators are started which are modified as follows: Their join operator is reduced to joining SMGs equal up to isomorphism, they use no list abstraction, and they use a bounded depth-first search to traverse the state space. They use bounds of 400, 700, and 1000 GIMPLE instructions, and so we call them as *Predator DFS hunters* 400, 700, and 1000, respectively.

If the Predator verifier claims a program correct, so does PredatorHP, and it kills all other Predators. If the Predator verifier claims a program incorrect, its verdict is ignored since it can be a false alarm (and, moreover, it is highly non-trivial to check whether it is false or not due to the involved use of list abstractions and joins). If one of the Predator DFS hunters finds an error, PredatorHP kills all other Predators and claims the program incorrect, using the trace provided by the DFS hunter who found the error.[1] If a DFS hunter claims a program correct, its verdict is ignored since it may be unsound.

In case the Predator verifier claims a program incorrect and no Predator DFS hunter finds an error within the appropriate bound, then PredatorHP starts one more Predator—a *Predator BFS hunter*. The BFS hunter does not use list abstraction and its join is reduced to equivalence up to isomorphism, but it performs an unlimited breadth-first search. If it manages to find an error within the SV-COMP'15 time budget, PredatorHP claims the program incorrect (note that without a time limit, the BFS hunter is guaranteed to find every error). If the BFS hunter finishes and does not find an error, the program is claimed correct. Otherwise, the verdict "unknown" is obtained.

3 Strengths and Weaknesses

The main strength of Predator lies in its sound treatment of heap manipulation. Unlike for various bounded model checkers, when Predator claims a program safe, all its possible behaviours are indeed safe. At the same time, Predator is also quite efficient. On the other hand, due to using over-approximation, it can easily generate false alarms. This danger was greatly reduced in PredatorHP by combining the sound Predator verifier with Predator hunters. This way, false alarms caused by abstraction are often suppressed, and a program claimed possibly unsafe by Predator can even be proved correct if its behaviour is bounded. Unfortunately, true error warnings can sometimes be also suppressed, resulting in a neutral "unknown" answer. However, overall, the balance is positive: about twice more false than true alarms were prevented on the SV-COMP benchmarks in the two categories where PredatorHP competes. The benefit is further

[1] The obtained trace can still be spurious due to the harsh abstraction of non-pointer data by Predator: All such data, apart from integers up to some fixed bound, are abstracted away.

amplified by the SV-COMP scoring scheme, which rewards preventing a wrong answer over keeping a correct one.

The improvement manifested mainly in the MemorySafety category, containing test-cases causing list abstractions in Predator to produce false alarms. By preventing all but a single one, while keeping all correct answers, PredatorHP is much more reliable than Predator alone. PredatorHP reduced false positives even for remaining SV-COMP categories, but unfortunately not enough to allow us to successfully participate in these.

The main weakness of PredatorHP is inherited from Predator, and it is the same as in previous years of SV-COMP. Namely, it is a rather weak support of non-pointer data and missing models of some library functions, which has not changed since SV-COMP'14. That is why, PredatorHP is participating in the MemorySafety and HeapManipulation categories only. Even within these categories, PredatorHP loses some points due to imprecise treatment of non-pointer data, leading to false alarms. The only other reason for Predator losing points in the MemorySafety and HeapManipulation categories is the fact that it cannot handle tree-like data structures and skip lists. In fact, it can handle them in a bounded way (i.e., in the same way as bounded model checkers)[2], but we have decided not to "harvest" easy points by sacrificing soundness of the verifier.

4 Tool Setup and Configuration

The source code of the PredatorHP release used in the competition can be downloaded from the project web page[3]. The file `README-SVCOMP-2015` included in the archive describes how to build PredatorHP from source code and how to apply the tool on the competition benchmarks. After successfully building the tool from sources, a script named `predatorHP.py` can be invoked, once for each input program. The script takes a verification task file as a single positional argument. Paths to both the property file and the desired witness file are accepted via long options. The verification outcome is printed to the standard output. The script does not impose any resource limits other than terminating its child processes when they are no longer needed.

5 Software Project and Contributors

Predator is an open source software project developed at Brno University of Technology (BUT) and distributed under the GNU General Public License version 3. The main author of Predator is Kamil Dudka. Besides Kamil and the PredatorHP team, numerous external contributors are listed in the `docs/THANKS` file in the distribution of Predator. Collaboration on further development of Predator (e.g., better support of non-pointer data, handling of incomplete code, support of tree data structures, etc.) is welcome.

[2] According to our experiments, if we interpreted the fact that no error was found by any Predator DFS hunter such that the program is correct, we could successfully handle all programs manipulating trees and skip lists present in the SV-COMP'15 benchmark.

[3] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator-hp`

Acknowledgement. The work was supported by the Czech Science Foundation project 14-11384S, the internal BUT project FIT-S-14-2486, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

References

1. Dudka, K., Holík, L., Peringer, P., Trtík, M., Vojnar, T.: From Pointers to Containers. Under submission (2015)
2. Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 215–237. Springer, Heidelberg (2013)
3. Dudka, K., Peringer, P., Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2011, Part I. LNCS, vol. 6927, pp. 527–534. Springer, Heidelberg (2012)