Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark (Competition Contribution)*

Michal Kotoun, Petr Peringer, Veronika Šoková, and Tomáš Vojnar

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

Abstract. This paper describes shortly the PredatorHP (Predator Hunting Party) analyzer and its participation in the SV-COMP'16 software verification competition. The paper starts by a brief sketch of the Predator shape analyzer on which PredatorHP is built, using multiple, concurrently running, specialised instances of Predator. The paper explains why the concrete mix of the different Predators was used, based on some characteristics of the SV-COMP benchmark.

1 Verification Approach

Predator Hunting Party (PredatorHP) uses the Predator shape analyzer, and so we first give a brief overview of Predator. Next, we discuss how Predator is used in the concurrent setting of PredatorHP, stressing changes from PredatorHP used in SV-COMP'15 together with a short analysis of the SV-COMP benchmark that motivated these changes.

1.1 The Predator Shape Analyzer

Predator aims at *sound* shape analysis of sequential, non-recursive C programs that use various kinds of lists (singly- or doubly-linked, possibly circular, nested, and/or shared) implemented using low-level C pointer statements. Predator can soundly deal with various forms of pointer arithmetics, address alignment, block operations, memory contents reinterpretation, etc.

The shape analysis implemented in Predator is a form of *abstract interpretation* which uses a domain of the so-called *symbolic memory graphs* (SMGs) [1]. SMGs are oriented graphs with two main kinds of nodes and two main kinds of edges. Nodes can be divided into *objects* and *values*. Objects are further divided into *regions* (representing concrete blocks of memory allocated on the stack, on the heap, or statically) and singly- or doubly-linked *list segments*, which represent in an abstract way uninterrupted sequences of singly- or doubly-linked regions. Edges can be divided into *has-value* and *points-to* edges. The former represent values stored in allocated memory (which are either pointers or other kinds of data), the latter represent targets of pointer values.

Both nodes and edges are annotated by a number of *labels* that carry information such as the size of objects, offsets at which values are stored in objects, offsets with which pointers point to target objects, the type of values, offsets at which linking fields of lists are stored, the nesting level of objects (to be able to represent nested lists), or

* The work was supported by the Czech Science Foundation project 14-11384S.

a constraint on the number of linked regions that a list segment represents. In particular, a list segment can either represent n or more regions for $n \geq 0$, or 0 or 1 regions. Further, SMGs can also contain *optional regions* where a pointer to such a region either points to some allocated memory or to `NULL`. Sizes of blocks and offsets can have the form of *intervals* with constant bounds which allows Predator to deal with operations such as address alignment. A special kind of edges are then *disequality edges* allowing one to express that two values are for sure different (while equality of objects is expressed by representing these objects by a single node of an SMG).

Symbolic execution of C statements on SMGs uses a concept of *reinterpretation* that is able to synthesize values of previously not explicitly written fields from the known values of other fields. Currently, this concept is instantiated for dealing with blocks of nullified memory, which is quite needed for analyzing low-level programs. Another key operation on SMGs is the *join operation* that is implemented via a synchronous graph traversal of the two SMGs to be joint. The join operation is used not only to reduce the number of SMGs to deal with but also as a basis of abstraction and entailment checking. Predator uses *function summaries* to facilitate inter-procedural analysis. The support of *arithmetic* in Predator is such that Predator deals with integers exactly up to some bound (32 in SV-COMP'16) and then replaces them by an unknown value.

Compared with SV-COMP'15, not many changes were done in the Predator analyzer itself. We have just resolved several minor issues by, e.g., correcting arithmetic in the 32-bit mode or replacing error messages produced when performing so-far unsupported operations over interval-based values by producing the "unknown" verdict.

1.2 Predator Hunting Party

In SV-COMP'15, we started to run several variants of Predator in parallel. Among them there was one Predator *verifier* implementing the above sketched sound shape analysis. Due to its use of abstraction, the verifier could produce false alarms, and so its result was accepted only when it proved a program correct. In parallel with the verifier, three Predator *DFS hunters* without any list abstraction (though still with limited precision of the arithmetic) and with different bounds on the depth of the state space search (in particular, 400, 700, and 1000 GIMPLE instructions) were used. The verdict of these hunters was considered only when they reported an error. If neither the verifier nor the DFS hunters produced an acceptable answer, a *BFS hunter* was started to perform a breadth first search without any list abstraction and with no bound on the length of its run (other than the timeout used by SV-COMP). The BFS hunter was allowed to report errors as well as to prove a program correct in case it exhausted its state space.

For SV-COMP'16, we have decided to preserve the above concept but to revisit suitability of the concrete numbers of hunters used, their limits on the state space search, as well as the order in which they are run. First, the number of concurrently running Predators stayed at four given by the four available cores. We have, however, decided to use only two DFS hunters, with the depth of the state space search limited to 200 and 900 GIMPLE instructions, respectively. In general, this move is motivated by having one hunter that quickly searches for bugs with very short witnesses and one than searches for longer but still not very long witnesses. Moreover, we have decided to start the BFS hunter right away in place of one of the cancelled DFS hunters. Its role is to

either prove correct finite-state programs (not proved correct by the verifier due to the abstraction used) or to find bugs that are not quickly found by the DFS hunters.

The above mentioned concrete DFS bounds are based on an analysis of those SV-COMP'16 programs in the heap data structures category that contain an error. In particular, it appears that: (1) In over 80 % of the cases, the error can be found in the limit of 200 instructions. (2) In about 96 % of the cases (meaning all but four of the considered programs with errors), the error can be found within 900 instructions. (3) In the remaining cases, the witness may be much longer (going up to over 50,000 instructions), which is too much for being used over all programs. Fortunately, in some of the cases, the witness may be quite long, but the search space is relatively narrow, so an error can still be found by the BFS hunter. In the end, we have programs proved correct by the verifier (but not the BFS hunter), programs proved correct by the BFS hunter (but not the verifier), programs with errors found by the DFS hunters (but not the BFS hunter), as well as programs with errors found by the BFS hunter (but not the DFS hunters).

The above change alone allowed us to prove one more program correct in the given time limit while at the same time saving around 38 % of the wall time. While the concrete numbers and bounds of hunters are tuned for the SV-COMP benchmark, the general set up of the prover and the hunters is applicable more broadly. The concrete numbers may be adjusted in a similar way for other sets of programs to be verified (especially when the programs are being repeatedly modified and verified) as common, e.g., in the world of search-based testing.

2 Strengths and Weaknesses

The main strength of PredatorHP is that—unlike various bounded model checkers—it treats unbounded heap manipulation in a *sound* way. At the same time, it is also quite *efficient*, and the use of various concurrently running Predator hunters greatly decreases chances of producing *false alarms* (there do not arise any due to heap manipulation, the remaining ones are due to imprecise treatment of other data types).

The main weakness of PredatorHP and also of Predator itself is its weak treatment of non-pointer data. Due to this, Predator participates in the heap data structures category only. Within this category, a weakness of Predator is that it is specialized in dealing with lists, and hence it does not handle structures such as trees or skip-lists (that is, it handles them very well in a bounded way, but our aim is to stick with sound verification).

3 Tool Setup and Configuration

The source code of PredatorHP used in SV-COMP'16 is freely available on the Internet[1]. The file `README-SVCOMP-2016` shipped with the source code describes how to build the tool. To run it, the script `predatorHP.py` can be invoked. 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

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

than terminating its child processes when they are no longer needed. To run PredatorHP in the BenchExec environment, the `predatorhp.py` wrapper can be used. More information about the setting of PredatorHP used in the competition can be found here: `http://sv-comp.sosy-lab.org/2016/systems.php`.

4 Software Architecture, Project, and Contributors

Predator is implemented in C++ with a use of Boost libraries as a GCC plug-in based on the Code Listener framework [2]. PredatorHP is implemented as a Python script. Predator is an open source software project distributed under the GNU General Public License version 3. The main author of Predator is Kamil Dudka. Besides him and the PredatorHP team, Petr Muller and numerous other people listed in the `docs/THANKS` file in the distribution of Predator have contributed to the distribution of Predator.

References

1. K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *Proc. of SAS'13*, *LNCS* 7935, pages 214–237, Springer, 2013.
2. K. Dudka, P. Peringer, and T. Vojnar. An Easy to Use Infrastructure for Building Static Analysis Tools. In *Proc. of EUROCAST'11*, *LNCS* 6927, pages 527–534, 2012.