Predator: A Tool for Verification of Low-level List Manipulation (Competition Contribution)*

Kamil Dudka, Petr Muller, Petr Peringer, and Tomáš Vojnar

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

Abstract. Predator is a tool for automated formal verification of sequential C programs operating with pointers and linked lists. The core algorithms of Predator were originally inspired by works on separation logic with higher-order list predicates, but they are now purely graph-based and significantly extended to support various forms of low-level memory manipulation used in system-level code. This paper briefly introduces Predator and describes its participation in the Software Verification Competition SV-COMP'13 held at TACAS'13.

1 Predator Introduction

Predator is a tool for fully automated verification of sequential C programs with pointers and dynamic linked data structures, such as complex kinds of singly- and doubly-linked lists that can be circular, shared, and/or hierarchically nested in an arbitrary way. The long term goal of the Predator project is handling real system code, such as the Linux kernel. To achieve this, the tool strives to cope with implementation tricks and techniques used frequently by system programmers to obtain highly efficient code. Such techniques include pointer arithmetic, valid usage of pointers with invalid targets, operations with memory blocks, or reinterpretation of the memory contents. The degree to which Predator can deal with such techniques is currently to a large degree unique among fully automated shape analysis tools. Although Predator supports checking for error label reachability, it concentrates on an implicit detection of memory-related bugs. Hence, our main focus in SV-COMP'13 is the newly introduced *MemorySafety* competition category.

Predator is available in the form of a GCC plug-in, which brings several advantages. First, it is possible to re-use the existing build systems of GCC-based projects for running the verification without a need to manually process the source code. Predator, as a GCC plugin, can take advantage of the powerful parsing capabilities of GCC. Error messages are presented in a format compatible with GCC, hence Predator can be used with any IDE that can use GCC. Predator uses the low-level GIMPLE representation of the GCC intermediate code as an input for its analysis. By default, Predator disallows external function calls in order to exclude any side effects that could potentially

* This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (project MSM 0021630528), the BUT projects FIT-S-11-1 and FIT-S-12-1, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

break memory safety. The only allowed external functions are those which are properly modelled by Predator wrt. proving memory safety. Besides `malloc` and `free`, Predator supports selected memory manipulating functions like `memset`, `memcpy`, or `memmove`.

Predator is implemented in C++ and runs on Linux. The dependencies needed for building Predator are Boost, CMake, and the GCC plug-in development files. Predator is publicly available under the GPLv3 license.

2 Verification Approach

Predator was inspired by works on fully automated shape analysis using separation logic with higher-order inductive predicates [1]. However, Predator represents sets of heap configurations using a graph-based representation instead of separation logic formulae, which allows one to easily apply various efficient graph-based algorithms for dealing with the representation. Since SV-COMP'12, the graph-based representation has been redesigned into the form of the so-called *symbolic memory graphs* (SMGs) and made much more fine-grained (byte-precise) to allow for successfully verifying programs that use the above mentioned low-level memory manipulation techniques [3].

Predator iteratively computes sets of SMGs for each basic block of the CFG of the given program, covering all its reachable configurations. Termination of the analysis is aided by join and abstraction algorithms operating on SMGs. The join algorithm is based on simultaneously traversing two SMGs and merging their corresponding nodes. The abstraction uses the join algorithm to merge pairs of neighbouring nodes of the same SMG, together with their sub-SMGs, into a single list segment. Predator does not use any off-the-shelf decision procedure since an expensive conversion from our representation would be needed. Instead, entailment between SMGs is checked rather efficiently using the join algorithm, which is extended to compare on-the-fly the generality of the SMGs being joined. To allow for multiple views of a single block of memory, Predator implements *read* and *write reinterpretation* algorithms (needed, e.g., for dealing with unions and type-casts). For more details, see [3].

Predator can prove absence of common memory safety bugs, such as invalid dereferences or memory leaks. Apart from that, Predator uses the fact that SMGs make it possible to easily check whether a given pair of memory areas overlaps in order to check for bugs caused by memory overlapping in a way prohibited by the C language (as in the parameters of `memcpy`). Predator can provide diagnostic information accompanying errors or warnings, which due to the use of abstraction and join has a form of acyclic graphs covering multiple program paths possibly leading to the error.

Predator supports pointers with both positive and negative offsets from the beginning of allocated objects. Moreover, it even supports pointers with offsets given by integer intervals, which is needed to cope with some low-level code using, e.g., address alignment. Predator provides a simple support for integer data by tracking integers precisely up to some bound and then abstracting them to unknown values. Further details can be found in the tool paper [2] and in the technical report [3].

3 Benchmark Results

The latest release of Predator can be downloaded from its web page[1]. Specific instructions for building and running Predator within the SV-COMP'13 competition are located in the file `README-sv-comp-TACAS-2013` in the distribution of Predator.

Since the main focus of Predator is on memory- and pointer-related bugs, where it can utilize its precise analysis of reachable heap configurations, we concentrate on the *MemorySafety* and *HeapManipulation* categories. Compared to the SV-COMP'12 version of Predator, we successfully analysed many more test cases in the *MemorySafety* category. The new version of Predator managed all but one test case in this category. In particular, it did not scale well-enough to verify a program working with a 32KB array. On the other hand, even the new version of Predator still timed out on several tests in the *HeapManipulation* category. These test cases store integral data in list nodes in a way that prevents the list segment abstraction of Predator from applying. As Predator aims at verification of system software (including device drivers), we were interested in the *FeatureChecks* category as well. Predator successfully verified all test cases in the *ldv-regression* directory and a few test cases from the *ddv-machzwd* directory. Further, Predator achieved good results in the *ProductLines* category, where it successfully verified 585 of 597 test cases.

Results in the *SystemC*, *Loops*, and *ControlFlowInteger* categories had a higher ratio of false positives than in the above mentioned categories, but still with a majority of judgements being correct. The false positives are again caused mostly by a too coarse analysis of integers. For many cases in these categories, Predator was unable to provide an answer. The *BitVectors* category is problematic for Predator: safe test cases were often judged as unsafe because the byte-precise memory model used by Predator was too coarse for the bit-level operations. Due to undefined external functions, Predator was not able to analyze any test case from the *DeviceDrivers64* and *Concurrency* categories.

Over all categories, there was not a single case where Predator would issue an incorrect `TRUE` answer. This is a design goal of Predator, and it strengthens our claims that implementation of our verification techniques is sound.

Our future work includes handling of low-level tree data structures, a support for code fragment analysis, and better handling of integer data. Especially the last item would be beneficial for Predator's performance in some SV-COMP categories since we observed a high number of false positives caused by a too coarse analysis of integer data.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. CAV'07*, *LNCS* 4590, Springer, 2007.
2. K. Dudka, P. Peringer, and T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Proc. of CAV'11*, *LNCS* 6806, 2011.
3. K. Dudka, P. Peringer, and T. Vojnar. Byte-Precise Verification of Low-Level List Manipulation. Technical Report No. FIT-TR-2012-04, FIT BUT, 2012.

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