2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation

A Value Analysis for C programs

Géraud Canet, Pascal Cuoq, Benjamin Monate*
Software Reliability Labs
CEA LIST
Boite 65, 91191 Gif-sur-Yvette Cedex, France
First.Last@cea.fr

Abstract

We demonstrate the value analysis of Frama-C. Frama-C is an Open Source static analysis framework for the C language. In Frama-C, each static analysis technique, approach or idea can be implemented as a new plug-in, with the opportunity to obtain information from other plug-ins, and to leave the verification of difficult properties to yet other plug-ins. The new analysis may in turn provide access to the data it has computed.

The value analysis of Frama-C is a plug-in based on abstract interpretation. It computes and stores supersets of possible values for all the variables at each statement of the analyzed program. It handles pointers, arrays, structs, and heterogeneous pointer casts. Besides producing supersets of possible values for the variables at each point of the execution, the value analysis produces run-time-error alarms. An alarm is emitted for each operation in the analyzed program where the value analysis cannot guarantee that there will not be a run-time error.

1. Frama-C's value analysis

Frama-C's value analysis[1] is a mostly automatic source code analyzer that relies on the principles of abstract interpretation. Its defining quality is that it is correct: it never remains silent for an operation in the source code for which there is a risk of run-time error. This guarantee only holds in the conditions defined in the corresponding manual[1]. The complete source code must be provided and the user takes responsibility for any necessary modelization of system calls. This includes system calls for dynamically allocating memory and creating concurrency. We show what results can be expected from Frama-C's value analysis on two examples. In the first example, the value analysis helps identify a bug in the analyzed code.

*This work has been supported by the French Agence Nationale de la Recherche (ANR) project U3CAT/2008SEGI02101

For a given precision, being allowed false negatives gives a static analyzer a chance to display less false positives (the analyzer can discard the alarms that seem most likely to be false positives). Frama-C's value analysis renounces the privilege to make this (difficult in itself) selection and displays all the alarms it finds. It aims to be usable for proving the absence of bugs, for instance in the verification of critical embedded software where the restrictions the value analysis places on the analyzed source code can be worked with. Despite the extreme position adopted by the value analysis on the compromise between false positives and false negatives, the number of false positives can remain contained, when the code lends itself well to analysis. The second included example, an analysis of the cryptographic hash function Skein-256, shows that sometimes the number of false positives can be zero. When, as in this example, there are no alarms, the software is guaranteed not to cause any run-time error—with other interesting properties that can be deduced along the way. One reason for having the value analysis as a plug-in in a more general verification framework (by opposition to existing standalone sound analyzers such as Astrée or PolySpace) is in fact to provide means to recover from false positives when the objective is not to find bugs but to prove their absence.

2. Example from the Verisec files

Verisec[3] is a C analysis benchmark. It is composed of numerous separate files, each containing code from various real-life software sources (Apache, sendmail, etc.) where bugs such as buffer overflows have previously been detected and fixed. We show how using the value analysis, we found a bug in a C file extracted from the OpenSER source code (case CVE-2006-6876). Though a mistake in a declaration can lead to the bug that is the subject of the benchmark, we found a more subtle error in this file.

When it is launched with default settings on this example, the value analysis warns about two statements. For each, the analyzer is unable to exclude the possibility that

978-0-7695-3793-1/09 $26.00 © 2009 IEEE
DOI 10.1109/SCAM.2009.22

some executions of the program might perform out of bounds accesses, but it cannot guarantee that there is a problem for either.

We instruct the analyzer to try for increased precision at the cost of using more resources—specifically, unrolling loops. This second analysis eliminates the second warning. This means that it was a false alarm after all: the analyzer now guarantees that there is no out-of-bound access at that statement. The first alarm remains, located a few statements before the bug mentioned in the original CVE report. By interactive examination of the supersets of values computed by the analysis, we uncover the cause of the alarm. The value analysis detects an execution path with a possible buffer overflow when an option "`+C`" expecting a numerical argument is parsed right at the end of the 8-char null-terminated input buffer.

We then design an input vector to expose the problem. A key step is to use the concrete value "`ABCDE+C`" in the incriminated buffer. Launched on the now deterministic piece of code, the analysis positively detects an out-of-bound access at the incriminated statement, validating our suspicions. Note that the buffer overflow would likely not be detected in a test of the uninstrumented binary, even if the problematic input vector was used.

The code in the Verisec benchmark is a much simplified version of a function that is still present in OpenSIPS and Kamailio, two projects derived from OpenSER. Thanks to the knowledge gathered on the simplified version, and again with the help of the interactive observation of the supersets propagated by the value analysis, we were able to reproduce the bad sequence in the original code.

3. A verification of Skein-256

Skein[2], a NIST SHA-3 contestant, is a set of cryptographic hash functions. Of the many formal properties that would be desirable to establish for it, we only consider the absence of run-time error and similar safety properties, not cryptographic properties. For this demo, we only establish these in the context of computing the 64-bit hash of an arbitrary 80-character message in a single call to the function `Skein_256_Update` from the reference implementation. Also, the results only hold for the platform for which Frama-C is configured (by default, IA-32 and gcc).

The functions `memcpy` and `memset`, used by Skein-256, must be provided in source form. We also define, in ten lines of C, an analysis context. The context initializes a 80-char input buffer `msg` to unknown values. This guarantees that all further results (values, absence of RTE) will encompass all possible messages. The buffer is then passed to the functions `Skein_256_Init`, `Skein_256_Update` and `Skein_256_Final`. The value analysis takes about 10 seconds and does not warn about any possible run-time error in this program. Because the analysis is correct, the absence of alarms formally establishes that there cannot be any run-time error when calling the Skein-256 functions in a pattern that follows that of our analysis context. In addition, further automatic Frama-C analyses show that the output buffer and the state explicitly passed from each Skein function to the next are the only variables that are written to, and `msg`, the state, and a static variable named `ONE`[1] the only variables that are read from, during these function calls. Because in the analysis context, the state is declared as an uninitialized local variable, the value analysis would emit an alarm if any single field of this struct was read before having been written to[2]. In fact, thanks to all the preceding results, and because any possibility of an unspecified operation at run-time would cause an alarm, we can guarantee that on IA-32/gcc, with our implementations of `memcpy` and `memset`, two successive sequences of calls to the functions `Init`, `Update` and `Final` on the same 80-byte message, in the same or different programs, produce the same hash as long as `ONE` has not been modified, regardless of compiler decisions in the memory layout, of the contents of the state before it is passed to `Init`, and of the contents of other variables[3].

4. Conclusion

We have shown that the value analysis of Frama-C can be useful in itself to detect bugs or prove their absence. Any Frama-C plug-in can also make use of the supersets of possible values computed by the value analysis to simplify its own treatment of C programs (e.g. regarding aliasing). The workability of the concept is corroborated by the list of existing plug-ins: value analysis, synthetic functional dependencies, program dependency graph, slicing (each of which builds upon the results of the previous ones), weakest precondition...

References

[1] Frama-C: http://frama-c.cea.fr.

[2] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family. Submission to NIST, 2008.

[3] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 389–392, New York, NY, USA, 2007. ACM.

[1] The variable named ONE is used to dynamically detect endianness

[2] By contrast, if a local variable is passed by address to several functions, a C compiler does not detect when it is used without being initialized

[3] We have not proved that the functions always terminate, but if they fail to terminate for one message, then they fail to terminate every time they are applied to this message