Detection of Security Vulnerabilities using Guided Model Checking

Aliaksei Tsitovich
University of Lugano, Switzerland
aliaksei.tsitovich@lu.unisi.ch

1 Introduction

Software security problems are good candidates for application of verification techniques. Usually it is not a complex task to represent certain security-related property in a particular verification framework. For instance in any software model checking environment (MC)[1] it is possible to state buffer overflow detection as a reachability problem. The approach works in theory and in practice, but has a major scalability drawback: the state-space, which represents all possible behaviors of the system, might grow exponentially in the size of the product of a model and a property. From the other side MC has an important advantage - a counter-example is produced automatically when the bug is found.

In contrast, several static analysis techniques [2,3] use abstract interpretation [4] to address security problems. They attempt to represent the nature of the vulnerability in the values from some abstract domain and to calculate such an abstract value for each location of the program. Carefully selected abstract domains allow both scalable computation and fairly precise results [2]. The algorithm is sound (no bugs are missed) but, 1) abstraction leads to detection of false bugs (so called false positives) and 2) no counter-example can be produced. Reported comparisons of tools, based on abstract interpretation, state that they are inapplicable in a wide industrial practice because of the unacceptably high number of false positives [5].

Dealing with program's loops is Achilles heel of the most existing static analysis techniques. In order to reason about programs with (possibly infinite) loops one has to unwind all loop iterations or to build an approximation of a program. The first variant is a direct way to countless refinements and/or the state explosion, the second one leads to false positives or even to the loss of soundness (if under-approximation of the loop is used). In this research I particularly tackle loops as a main source of both scalability and precision problems. I want to explore how the existing techniques can be combined in a way that minimizes the effect of their drawbacks in analysis of the program loops.

2 Goals and Achieved Results

The goal of my research is the development of automated methods to detect security vulnerabilities in a large-scale software. I would like to come up with a problem-driven algorithm, which combines model checking and abstract interpretation in application to the reachability analysis. I see a following possible way to achieve the goal:

1. Develop an algorithm, which creates an over-approximated model of a program by summarization of the code fragments with a possibly infinite behaviors, i.e., loops.
2. Build reachability analysis using MC algorithms to verify "loop-less" models. In particular bounded model checking (BMC) [6] is a promising candidate because: 1) it targets bug detection but not a bug-absence proof; 2) it removes loops, the main limitation of BMC.
3. Develop a strategy to refine the summarized program.

The first part of the work, dedicated to loop summarization, has been accomplished and presented in [7]. The summarization algorithm was implemented in a tool called LoopFrog[1]. It targets verification of ANSI-C programs for string-related properties, e.g. buffer overflows. In [7] each loop is summarized with a help of localized abstract domain tailored to the verified property. Abstract domain suggests invariant candidates which are checked to be inductive invariants of a given loop. Repeating this procedure in a bottom-up manner gives an algorithm to over-approximate every loop instance by its *summary*. A summary is a combination of loop's variants (i.e. nondeterministically assigned variables) and discovered invariants. At the end of this *summarization* one obtains an over-approximated *loop-less* model of the program. The important property of this model is that any path in it is finite and, thus, is easily analyzable by BMC.

There are still a lot of ideas to explore such as abstract domains incremental strengthening, abstract counter-examples analysis, effective abstract transformers computation or incremental BMC. Finally, I want to obtain "a guided model-checker" - algorithm that delivers property-tailored and incremental abstraction/refinement scheme, which is applicable to a large-scale software.

References

1. Edmund M. Clarke, J., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge, MA, USA (1999)
2. Ganapathy, V., Jha, S., Chandler, D., Melski, D., Vitek, D.: Buffer overrun detection using linear programming and static analysis. In: Proceedings of CCS '03, New York, NY, USA, ACM (2003) 345–354
3. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software **19** (2002) 42–51
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
5. Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. In: SIGSOFT FSE. (2004) 97–106
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 118–149
7. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Proceedings of ATVA 2008, Springer (2008) To appear.

[1]Loopfrog binaries, benchmarks results and examples are available at http://www.verify.inf.unisi.ch/loopfrog