JAIST Repository
https://dspace.jaist.ac.jp/

Title	Static Program Analysis for Software Validation
Author(s)	Li, Xin
Citation	
Issue Date	2008-03-04
Type	Conference Paper
Text version	publisher
URL	http://hdl.handle.net/10119/8232
Rights	
Description	JAIST 21世紀COEシンポジウム2008「検証進化可能電子社会」= JAIST 21st Century COE Symposium 2008 Verifiable and Evolvable e-Society, 開催：2008年3月3日～4日, 開催場所：北陸先端科学技術大学院大学, GRP研究員発表会　セッションB-1発表資料

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Static Program Analysis for Software Validation

Li Xin

Japan Advanced Institute of Science and Technology
li-xin@jaist.ac.jp

1 Objectives

The increasing complexity of software nowadays makes their validation more challenging, especially for safety-critical e-commerce applications. For example, the economic cost of security vulnerability on web applications alone is currently estimated to be $180 billion [5]. At present, practiced methods for software validation are mostly based on simulation and testing. The fundamental problem for these methods is that they cannot cover all possible scenarios of system runs. Thus, they are incapable of exposing subtle defects. A promising alternative to software validation is formal methods of mathematics, of which popular approaches are *theorem proving*, *model checking*, etc. Theorem proving is a deductive approach, and the use of it usually demands expertise and enough experience. In contrast, model checking, the so-called "*push-button technique*", is a fully automatic algorithmic technique for verification on temporal safety of reactive and concurrent systems. In particular, if model checking once fails, counterexamples are provided as evidences for the failure and clues for fixing the problem. The purpose of this research is to present static program analysis based on model checking techniques for software validation.

2 Approaches

Our approach is motivated by the insight [6] that program analysis can be regarded as model checking of abstract interpretation. Following this methodology, program analysis naturally enjoys soundness guarantee from abstract interpretation and "push-button" facilities from model checking. Popular model checkers, such as SPIN, NuSMV/SMV, are model checkers on finite state space. Recently, practical model checking algorithms on pushdown systems [2] are proposed, which enables the design of *context-sensitive* program analysis. More specifically, procedure calls and recursions can be modeled as pushdown systems in the analysis without placing a restriction on the calling depth. The static analysis thus produces results for which procedure calls and returns are guaranteed to correctly match one another, called *valid paths.* Model checking on pushdown systems enables us to perform a real *interprocedural* program analysis.

In recent years, Java has emerged as the popular language for building large-scale web applications, because of not only Java's nice built-in security model but also the proposal of J2EE(Java 2 Enterprise Edition) tailored for implementing

e-commerce applications. Our work in general is an interprocedural extension of BANDERA-like approach. As shown in Figure 1, our analysis framework provides

- The design and prototype implementation of static program analysis with model checkers as the analysis engine; and
- The support of automatic extraction for conservative and infinite models to be analyzed, i.e., model checked, from Java source codes.

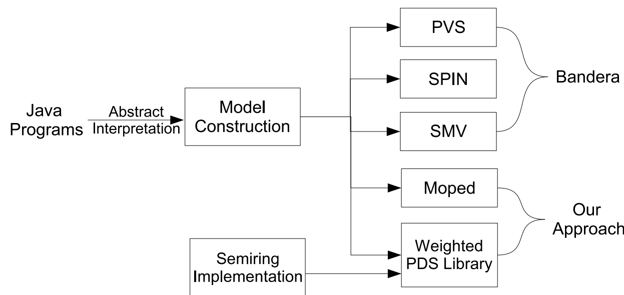

Fig. 1. Our Methodology

Points-to analysis detects the set of heap objects possibly referred to by references at run-time. In Java, call graph generation and points-to analysis are mutually dependent. Since they are the starting point of most other program analysis on Java applications. We firstly proposed context-sensitive, field-sensitive, and flow-insensitive points-to analysis algorithms which are implemented as a tool in our analysis framework.

3 Progress

Provided with a ready-made points-to analysis, we propose a precise context-sensitive, field-sensitive and flow-insensitive *relevancy analysis*(equivalently, irrelevancy analysis). Our relevancy analysis is to compute the set of program variables of concerned type that are *relevant* to the set of designated variables, e.g., typically program inputs. The analysis is leveraged from an interprocedural irrelevant code elimination [3] based on weighted pushdown model checking. The idea is, if the change of a value does not affect the value of an output, we regard it as *irrelevant* and relevant otherwise. By *relevancy*, we mean there could exist dataflow from designated variables to some program variables. Our relevancy analysis can be applied to target many application scenarios. For instance, it could be utilized to pin-point the program parts where symbolic inputs can flow into in symbolic execution [1]. The results would guide the program instrumentation and thus improve the performance of symbolic execution. It could be also utilized to find security vulnerabilities in Java applications [4].

The analysis infrastructure is shown in Figure 2, which consists of five modules necessary to our context-sensitive relevancy analysis. The (0)th module is for the underlying points-to analysis that relevancy analysis is based on. The choice of points-to analysis is independent of relevancy analysis, but is sensitive to the analysis precision. Since points-to analysis is mutually dependent to call graph generation for Java programs, the interprocedural control flow graph is meanwhile prepared in this phase. In the analysis, we take Jimple [7] as the target language, which is a three-address intermediate representation of Java and syntactically much simpler.

The relevancy analysis consists of two phases. The first phase (1) models the program as an interprocedural dependence graph. A dependence graph is built with a sound approximation on Java programs, which is later encoded as the underlying pushdown system for modeling checking. Various choices of modeling on Java programs are presented tradingoff the precision and efficiency. The second phase (2) performs relevancy analysis on concerned program variables by calling model checkers against the dependence graph. Pushdown and weighted pushdown model checkers are explored as the back-end model checking engines. Currently the relevancy analysis is implemented based on the Weighted PDS library (3), which calls a weight package (4) that is designed and prepared in advance. The modules in the model checking engine is implemented in C, and other modules of the analysis is implemented in Java.

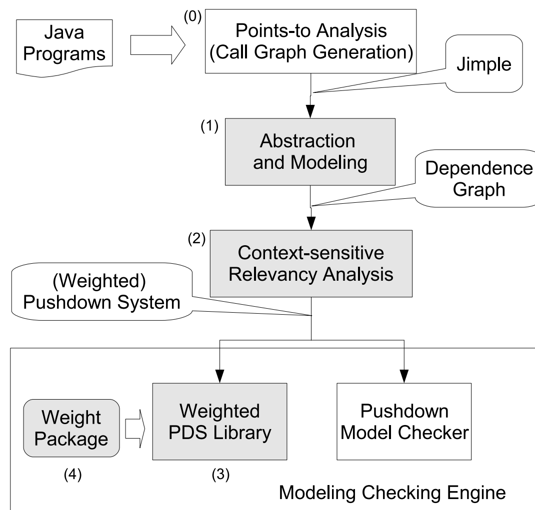

Fig. 2. The Analysis Infrastructure.

4 Future Work

Our future work will be:

- Evaluate our (ir)relevancy analysis with applying it to various application scenarios in practice, e.g., to help improve the efficiency of symbolic execution and to find security vulnerabilities in Java applications.
- Trading off precision and efficiency, a demanded-driven or a compositional way of points-to analysis will be explored to make our points-to analyzer scalable on large-scale applications.
- Our analysis framework arises industrial interests. More collaboration will be further carried out later on.

References

1. S. Anand, A. Orso, and M. J. Harrold. Type-dependence analysis and program transformation for symbolic execution. In *TACAS*, pages 117–133, 2007.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150, London, UK, 1997. Springer-Verlag.
3. X. Li and M. Ogawa. Interprocedural program analysis for java based on weighted pushdown model checking. In *The 5th International Workshop on Automated Verification of Infinite-State Systems*. AVIS, April 2006.
4. V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.
5. D. Rice. *Geekonomics: The Real Cost of Insecure Software*. Addison Wesley, New York, 2007.
6. D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48, New York, NY, USA, 1998. ACM.
7. R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON*, page 13, 1999.

A System Development

Develop the Jaist Static Analysis Framework on Java for points-to analysis and (ir)relevancy analysis.