Science of Computer Programming 163 (2018) 93–114

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Projected control graph for computing relevant program behaviors

Ahmed Tamrawi [a,*], Suresh Kothari [b]

[a] EnSoft Corp., Ames, IA 50010, USA
[b] Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA

ARTICLE INFO

Article history:
Received 17 August 2017
Received in revised form 17 April 2018
Accepted 26 April 2018
Available online 30 April 2018

Keywords:
Program behaviors
Software analysis
Software verification
Software security
Software safety

ABSTRACT

Many software engineering tasks require analysis and verification of all behaviors relevant to the task. For example, all relevant behaviors must be analyzed to verify a safety or security property. An efficient algorithm must compute the relevant behaviors directly without computing all the behaviors. This is crucial in practice because it is computationally intractable if one were to compute all behaviors to find the subset of relevant behaviors. We present a mathematical foundation to define relevant behaviors and introduce the Projected Control Graph (PCG) as an abstraction to directly compute the relevant behaviors. We developed a PCG toolbox to facilitate the use of the PCG for program comprehension, analysis, and verification. The toolbox provides: (1) an interactive visual analysis mechanism, and (2) APIs to construct and use PCGs in automated analyses. The toolbox is designed to support multiple programming languages.
Using the toolbox APIs, we conducted a verification case study of the Linux kernel to assess the practical benefits of using the PCG. The study shows that the PCG-based verification is faster and can verify 99% of 66,609 instances compared to the 66% instances verified by the formal verification tool used by the Linux Driver Verification (LDV) organization. This study has revealed bugs missed by the formal verification tool. The second case study is an interactive use of the PCG Smart View to detect side-channel vulnerabilities in Java bytecode.

1. Introduction

Accounting precisely for the execution behavior along each path of a Control Flow Graph (CFG) blows up the computational complexity: (1) the number of CFG paths grows exponentially with the number of non-nested branch nodes [1,2], and (2) path feasibility checks can incur an exponential computation [3–8]. Moreover, the number of paths blows up because of loop iterations.

In practice, the number of behaviors relevant to a task is often significantly smaller than the totality of behaviors. We present a mathematical foundation for computing *relevant program behaviors* as *relevant base behaviors* and *relevant iterative behaviors*. The goal is to compute the relevant behaviors directly without computing all possible behaviors. Based on the mathematical foundation, we introduce the Projected Control Graph (PCG) as an abstraction to directly compute the relevant behaviors.

* Corresponding author.
E-mail addresses: ahmedtamrawi@ensoftcorp.com (A. Tamrawi), kothari@iastate.edu (S. Kothari).

https://doi.org/10.1016/j.scico.2018.04.003
0167-6423/© 2018 Elsevier B.V. All rights reserved.

Along with the mathematical foundation, we present insightful examples to illustrate possibilities of the drastic reduction in the number of behaviors from all behaviors to the relevant behaviors. Next, we summarize results of applying our PCG-based verification to the Linux kernel to verify the pairing of `Lock` instances with corresponding `Unlock` instances on all feasible control flow paths [9]. A control flow path is feasible if the path can be taken during an actual execution, i.e., variable values can be attained to satisfy the conditions governing the path. The study includes three versions of the Linux kernel which altogether have 37 million lines of code and 66,609 `Lock` instances. We present a comparison with the formal verification tool that uses BLAST [10]. This BLAST tool has been a top performer in the annual software verification competition (SV-COMP) [11] and it is used by the Linux Driver Verification (LDV) organization [12]. The BLAST tool verifies 43,766 (65.7%) of `Lock` instances as correctly paired (safe), and it is inconclusive (crashes or times out) on 22,843 instances. The BLAST tool does not find any unsafe instances and requires 172 hours and 56 minutes for its verification. Our PCG-based automated verification tool verifies 66,151 (99.3%) of `Lock` instances as safe, and it is inconclusive on 451 instances. Seven unsafe instances are found through our study, including an instance that was incorrectly verified as safe by the BLAST tool. The PCG-based tool required 3 hours and 24 minutes.

Our second study is to use the PCG interactively to analyze Java bytecode to detect side-channel vulnerabilities. A compact PCG not only improves efficiency of an automated analysis, it also facilitates an interactive visual analysis and program comprehension. Using the Atlas Platform [13,14], we have designed a visual analysis mechanism, called the *PCG Smart View*, to use the PCG interactively.

The key research contributions are:

- A mathematical foundation to define and compute relevant behaviors as *relevant base behaviors* and *relevant iterative behaviors*.
- The PCG as a graph abstraction to directly compute the relevant behaviors and an efficient algorithm to compute the PCG.
- An assessment of the practical impact of using the PCG interactively and programmatically for analyzing or verifying large software.

The remainder of the paper is organized as follows. We first describe the class of software safety and security problems to which the mathematical foundation for computing relevant behaviors applies in Section 2. Next, Section 3 describes the mathematical foundation for computing relevant behaviors. Section 4 presents the linear-time algorithm for constructing the PCG from its corresponding CFG. The developed PCG toolbox is presented in Section 5. Section 6 presents our Linux verification study that assesses the practical benefits of using PCGs in automated analysis and for interactive analysis. Section 7 discusses the use of PCGs in detecting side-channel vulnerabilities. Section 8 presents the related work. Finally, we conclude in Section 9.

2. Software safety and security properties

This section describes a fairly broad class of software safety and security properties which can be verified efficiently using the PCG. In general, the PCG can be of significant value for program comprehension, analysis, and verification.

Definition 1 *(2-event matching).* Verify that an event $e_1(O)$ is succeeded by an event $e_2(O)$ on every feasible execution path, where the two events are operations on the same object O.

Besides the lock/unlock pairing verification described in this paper, the 2-event matching covers several problems such as memory allocation/deallocation pairing, or file open/close pairing. A number of vulnerabilities listed by the MITRE Corporation [15] can be viewed as 2-event problems.

Definition 2 *(2-event anti-matching).* Verify that an event $e_1(O)$ is not succeeded by an event $e_2(O)$ on any feasible execution path, where the two events are operations on the same object O.

Anti-matching covers software security verification defined according to *Confidentiality*, *Integrity*, and *Availability* (CIA) triad [16]. A confidentiality verification problem could be defined as: a *sensitive source* must *not* be followed by a *malicious sink* on any feasible execution path. Similarly, an integrity verification problem could be defined as: an *access to sensitive data* must *not* be followed by a *malicious modification to sensitive data* on any feasible execution path.

The following defines the general class of verification tasks for applying the PCG.

Definition 3 *(n-event verification).* Verify on every feasible execution path, that the occurrence of events on the path follow the acceptability test defined by a Finite State Machine (FSM) $\phi(\mathcal{E})$, where $\mathcal{E}$ is a set of n events that operate on the same object O.