A Case Study on Verification of a Cloud Hypervisor by Proof and Structural Testing*

Nikolai Kosmatov¹, Matthieu Lemerre¹, and Céline Alec²

¹ CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette, France
{firstname.lastname}@cea.fr
² LRI, CNRS UMR 8623, Université Paris-Sud, France
lastname@lri.fr

Abstract. Complete formal verification of software remains extremely expensive and often reserved in practice for the most critical products. Test generation techniques are much less costly and can be used in combination with theorem proving tools to provide high confidence in the software correctness at an acceptable cost when an automatic prover does not succeed alone. This short paper presents a case study on verification of a cloud hypervisor with the Frama-C toolset, in which deductive verification has been advantageously combined with structural all-path testing. We describe our combined verification approach, present the adopted methodology and emphasize its benefits and limitations.

Keywords: deductive verification, test generation, specification, Frama-C.

1 Introduction

Deductive verification can provide a rigorous mathematical proof that a given annotated program respects its specification, but remains relatively expensive, whereas testing can find counter-examples or increase confidence in the program correctness at a much lower cost. This short paper describes how both techniques have been combined during the verification of a critical module of a cloud hypervisor using the Frama-C toolset [1]. This case study has focused on combining automatic theorem proving and automatic structural testing in order to provide a high confidence in the system within limited time and costs. In particular, we address the question of how to share the roles between formal proof and testing in order to take the best of each technique and to increase the final level of confidence. The contributions of this paper include the presentation of the combined verification approach, the proposed methodology, its evaluation and results.

2 The Anaxagoros Hypervisor and Its Virtual Memory Module

Since the usage of cloud becomes pervasive in our lives, it is necessary to ensure the reliability, safety and security of cloud environments [2]. Anaxagoros [3,4] is a secure

* This research work has received funding from the FUI-AAP14 SYSTEM@TIC Paris-Région project “PISCO” partially funded by bpifrance.
microkernel and hypervisor developed at CEA LIST, that can virtualize preexisting operating systems, for example, Linux virtual machines. It enables execution of hard real-time tasks or operating systems, for instance the PharOS real-time system [5], securely along with non-real-time tasks, on a single chip. This goal has required to put a strong emphasis on security in the design of the system.

A critical component to ensure security in Anaxagoros is its virtual memory system [4]. The x86 processor (as many other high-end hardware architectures) provides a mechanism for virtual memory translation, that translates an address manipulated by a program into a real physical address. One of the goals of this mechanism is to help to organize the program address space, for instance, to allow a program to access big contiguous memory regions. The other goal is to control the memory that a program can access. The physical memory is split into same-sized regions, called frames or physical pages, that we will simply call pages in this paper. Pages can be of several types: data, pagetable, pagedirectory. Basically, page directories contain mappings (i.e. references) to page tables, that in turn contain mappings to data pages. The page size is 4kB on standard x86 configurations.

Anaxagoros does not decide what is written to pages; rather, it allows tasks to perform any operations on pages, provided that this does not affect the security of the kernel itself, and of the other tasks in the system. To do that, it has to ensure only two simple properties. The first one ensures that a program can only access a page that it “owns”. The second property states that pages are used according to their types.

Indeed, the hardware does not prevent a page table or a page directory from being also used as a data page. Thus, if no protection mechanism is present, a task can change the mappings and, after realizing a certain sequence of modifications, it can finally access (and write to) any page, including those that it does not own.

The virtual memory module should prevent such unauthorized modifications. It relies on recording the type of each page and maintaining counters of mappings to each page (i.e. the number of times the page is referred as a data page, page table, or page directory). The module ensures that pages can be used only according to their role. In addition, to allow dynamic reuse of memory, the module should make it possible to change the type of a page. To avoid possible attacks, changing the page type requires that we ensure even more complex additional properties. (Simplified) examples of properties include: page contents should be cleaned before any type change; still referred pages cannot be cleaned; the cleaning should be correctly resumed after an interruption; the counters of mappings (references) should be correctly maintained; cleaned pages are never referred to; etc.

3 The Verification Approach and Methodology

3.1 Context and Objectives

The verification target of this case study was a simplified sequential version of the Anaxagoros virtual memory system containing a significant subset of its features (pages of all three types, read-only and writable mappings, page cleaning with possible interruptions, page type changes, counters of mappings, etc.). Our objective was to study how such different verification techniques as automatic theorem proving and structural