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

Nikolai Kosmatov, Matthieu Lemerre, Céline Alec

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

Nikolai Kosmatov[1], Matthieu Lemerre[1], and Céline Alec[2]

[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

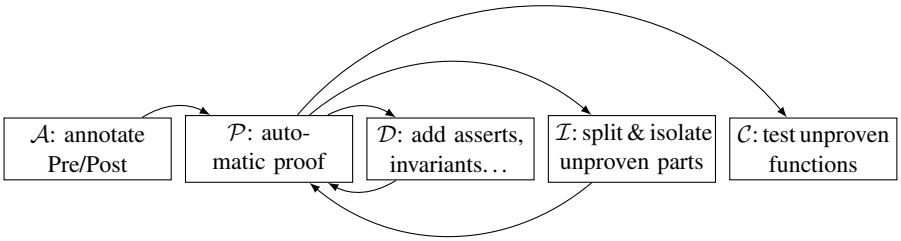

Fig. 1. Methodology of combined verification

how such different verification techniques as automatic theorem proving and structural testing could be combined together in order to provide the best trade-off between correctness guarantees obtained by rigorous formal proof and low cost of automatic structural testing.

We used verification tools offered by the FRAMA-C framework for verification of C programs [1], in particular, the JESSIE plugin [6] for Hoare logic based deductive verification with the automatic provers Alt-Ergo and Simplify, and the concolic test generator PATHCRAWLER [7, 8]. FRAMA-C also offers an expressive specification language for C programs called ACSL [1]. Therefore we needed to annotate the C code in ACSL, apply whenever possible automatic theorem proving using JESSIE and complete verification using PATHCRAWLER. The ACSL specification was derived from informal specification of the code and an earlier formalization and (paper-and-pencil) proof of properties for the Anaxagoros virtual memory system detailed in [9].

3.2 The Methodology

Very soon after the beginning of the project, when the first proof failures occurred (along with the difficulties of their analysis) and the first naive attempts to complete verification with testing appeared to be inconclusive, it became clear that we needed to elaborate a structured methodology that would allow to advantageously combine proof and testing. The adopted methodology is outlined in Fig. 1.

Step ($\mathcal{A}$) consists of writing initial **A**nnotations including e.g. function contracts with pre-/postconditions and auxiliary predicates with global invariants necessary to express function contracts. Step ($\mathcal{P}$) applies automatic **P**roof. When proof failures occur, the specification **D**etailing step ($\mathcal{D}$) consists in analysis of failures and adding further annotations (assertions, loop invariants, revised function contracts). Several iterations between Steps ($\mathcal{P}$) and ($\mathcal{D}$) can be necessary to help an automatic prover to prove as many properties as possible, and to identify the origin of each remaining proof failure, for instance, by surrounding relevant statements by appropriate assertions. The first steps ($\mathcal{A}$), ($\mathcal{P}$), ($\mathcal{D}$) are commonly used in deductive verification practice.

When the origin of each proof failure is identified (in terms of particular statements that cannot be traversed by the proof, and particular parts of the global property whose proof fails), we apply the **I**solation step ($\mathcal{I}$). It consists of splitting an unproven function

into simpler ones in order to isolate an unproven part in a smaller annotated function. So, if a function f is not proven and a (block of) statement(s) s is identified as the origin of a proof failure, we isolate s in a separate annotated function g, and a call to g will now replace s in f. Since modular deductive verification of f relies on the contract of g, it allows us to prove f (under hypothesis that g is correct). The original function f is now proven, and we isolated proof failures in simpler functions.

Finally, each remaining unproven function g is verified using Step ($\mathcal{C}$) that applies all-path testing w.r.t. a specification, sometimes also called ***C**ross-checking*. Assume the specification of a function g is translated into a C function u. In cross-checking, the user runs all-path testing on a new function h that calls the function under test g and its specification u, in order to check whether it is possible to cover a path which conjoins a path in g and a path in u which fails to satisfy the specification (see [10, "Bypassing the limits..." section] for more detail). When cross-checking is used with an all-path testing tool ensuring completeness like PATHCRAWLER [8, Sec. 3.1] and when the tool manages to explore all paths (hence, the resulting function h has a finite number of paths), the absence of a counter-example provides a guarantee of correctness.

In general, the number of paths in h can be however too big to be explored. In this case, testing cannot provide any guarantee of correctness. Hence, we propose to *limit the program input domain so that the number of paths becomes finite but remains representative* of the behavior of the function under test (cf discussion below). Absence of counter-examples for the remaining unproven functions established by automatic cross-checking on a reduced input domain provides the verification engineer with additional confidence in correctness of these unproven parts when an automatic prover fails to complete the proof.

3.3 Benefits and Limitations of the Approach

We applied and evaluated the proposed methodology on the present case study. The results are very encouraging. First of all, relying only on automatic verification techniques, our approach could be acceptable for most software developers and validation engineers. Indeed, the present case study was performed within 2 months, and was mainly conducted by a junior software engineer who did not have any experience in software verification before this project. The complete annotated C code contains 2400 lines with 37 functions and the total number of 3915 proof obligations was generated by JESSIE.

Second, Steps ($\mathcal{D}$) and ($\mathcal{I}$) helped to prove as much as possible (98.8% of generated proof obligations were proven by JESSIE), and to identify and isolate actions and properties for which automatic proof failed. Starting from a situation where automatic proof failed for most functions without any clear reason, we were finally able to precisely identify and isolate the real proof issues. Unproven code has been reduced to one-line functions (e.g. changing one element of a page) that impact the counters of mappings.

Maintaining the counters of mappings appeared to be the most difficult issue for automatic proof in this case study so we present it here in more detail. Fig. 2 shows a (simplified) definition of two inductive predicates used to count mappings, where `pData[p*PageSize + i]` represents the element of index `i` in page of index `p`. The predicate `CountOne` states that `N` is the number of occurrences of target page `targ` at index

range `0..last` of the page of index `p`. The predicate `CountAll` states that `N` is the number of occurrences of target page `targ` in the pages of index range `0..lastP`. A frequent origin of proof failures in this case study is related to the (simplified) global invariant `Inv` that says that `Mappings[targ]` is indeed the number of occurrences of `targ` in all pages.

Finally, thanks to Steps ($\mathcal{D}$) and ($\mathcal{I}$), the isolated unproven functions were quite appropriate for cross-checking. For instance, consider a simple function writing a new element into a page: `pData[p*PageSize + i]=new`. If true in the precondition, the invariant `Inv` would remain true in the postcondition for all elements except for the new and the old element (if they are different) for which the real number of occurrences becomes greater (resp., less) by 1 than the unmodified counter of mappings. Such elementary unproven functions do not contain any loops, and the C version of its specification contains only fixed-size loops over all page entries to compute the real number of occurrences (specified by `CountAll` in Fig. 2). Therefore, a simple way to limit path explosion for such functions would be to limit the page size and number of pages to smaller constants, say, 5. This limit does not modify the function logic, and is very unlikely to eliminate a counter-example in this case (even if it cannot be excluded). It shows that cross-checking of Step ($\mathcal{C}$) can be run on reduced, but representative input domain, and provide a higher confidence in program correctness at a relatively low cost.

One limitation of the methodology is the need to restructure the code and to move some parts of a function into a separate function at Step ($\mathcal{I}$) that might be not desirable at the verification step. Notice however, that it could often be acceptable because high-level function interfaces (such as microkernel hypercalls) are not modified, since the code restructuring is performed on the sub-function level. Moreover, the proposed methodology can be hopefully adopted by the developers that might find it useful to structure their code in a way that facilitates verification.

The second limitation is related to the need of reducing the input domain for cross-checking at Step ($\mathcal{C}$). When there is no way to reduce the program input domain to a representative smaller subset, it can still be interesting to obtain some confidence after a partial cross-checking.

4 Related Work and Conclusion

Klein et al. [11] presented formal verification for seL4, a microkernel allowing devices running seL4 to achieve the EAL7 level of the Common Criteria. Another formal verification of a microkernel was described in [12]. In both projects, the verification used interactive, machine-assisted and machine-checked proof with the theorem prover Isabelle/HOL. The formal verification of a simple hypervisor [13] used VCC, an automatic first-order logic based verifier for C. The underlying architecture was precisely modeled and represented in VCC, where the mixed-language system software was then proved correct. Unlike [11] and [12], this technique was based on automated methods.

[14] reports on verification of the translation lookaside buffer (TLB) virtualization, a core component of modern hypervisors. As devices run in parallel with software, they require concurrent program reasoning even for single-threaded software. This work gives a general methodology for verifying virtual device implementations, and demonstrate the verification of TLB virtualization code in VCC.

```
#define NumPages 10000    // number of memory pages
#define PageSize 1024    // page size (in words)
unsigned int pData[NumPages * PageSize]; // page entries
unsigned int Mappings[NumPages]; // counters of references (mappings) to pages
/*@
inductive countOne{L}(integer p, integer last, integer targ, integer N){
  case oneEq: \forall integer p, targ;
    0<=p<NumPages && pData[p*PageSize] == targ ==> countOne(p, 0, targ, 1);
  case oneNotEq: \forall integer p, targ;
    0<=p<NumPages && pData[p*PageSize] != targ ==> countOne(p, 0, targ, 0);
  case severalLastNotEq: \forall integer p, last, targ, N;
    (0<=p<NumPages && 0<last<PageSize && pData[p*PageSize + last] != targ &&
    countOne(p, last-1, targ, N) ==> countOne(p, last, targ, N) );
  case severalLastEq: \forall integer p, last, targ, N;
    (0<=p<NumPages && 0<last<PageSize && pData[p*PageSize + last] == targ &&
    countOne(p, last-1, targ, N) ==> countOne(p, last, targ, N+1) );
}
inductive countAll{L}(integer lastP, integer targ, integer N){
  case onePage: \forall integer targ, N;
    ( countOne(0, PageSize-1, targ, N) ) ==> countAll(0, targ, N);
  case severalPages: \forall integer lastP, targ, N1, N2;
    ( 0 < lastP < NumPages && countAll(lastP-1, targ, N1) &&
    countOne(lastP, PageSize-1, targ, N2) ==> countAll(lastP, targ, N1+N2) );
}
*/
/*@
predicate Inv{L} = \forall integer targ; 0<=targ<NumPages ==>
    countAll(NumPages-1, targ, Mappings[targ]);
*/
```

Fig. 2. Simplified ACSL predicates for counting mappings (occurrences) in memory pages

Formal verification nowadays remains very expensive. [15] estimates that the verification of the seL4 microkernel took around 25 person-years, and required highly qualified experts. seL4 contains only about 10,000 lines of C code, and verification cost is about $700 per line of code.

Our present work continues these efforts, but in addition fixes a quite different objective: to perform a real-life case study using a combination of automatic theorem proving and automatic all-path testing, and to explore how to find a reasonable trade-off between rigorous proof and cross-checking of the program on a reduced program domain. We described our methodology and evaluated it during this project. In particular, our results suggest that all-path testing of the code w.r.t. a specification on a reduced program domain can be a precious complement to deductive verification allowing any verification engineer (without being a highly qualified expert) to achieve a higher level of confidence within a very limited time and cost and without using more expensive interactive proof.

An ongoing work is aimed at a complete formal verification of the virtual memory module of Anaxagoros by combining automatic and interactive proof tools. The first observations confirm the conclusions of this case study: properties we validated by cross-checking appear so far to be correct and provable in the interactive proof tool Coq, while their interactive proof takes (at least 10x) more time and requires a higher level of qualification of the verification engineer.

Future work includes further evaluation of the proposed combined methodology, as well as verification of the complete code of the Anaxagoros virtual memory module taking into account parallel execution in several threads.

References

1. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - a software analysis perspective. In: SEFM 2012
2. Loulergue, F., Gava, F., Kosmatov, N., Lemerre, M.: Towards Verified Cloud Computing Environments. In: HPCS 2012
3. Lemerre, M., David, V., Vidal-Naquet, G.: A communication mechanism for resource isolation. In: IIES 2009
4. Lemerre, M., David, V., Vidal-Naquet, G.: A dependable kernel design for resource isolation and protection. In: IIDS 2010
5. Lemerre, M., Ohayon, E., Chabrol, D., Jan, M., Jacques, M.B.: Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In: AMICS 2011
6. Moy, Y.: Automatic Modular Static Safety Checking for C Programs. PhD thesis, Univ. Paris 11 (2009)
7. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: EDCC 2005
8. Botella, B., Delahaye, M., Hong Tuan Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST 2009
9. Lemerre, M.: Intégration de systèmes hétérogènes en termes de niveaux de sécurité. PhD thesis, Université Paris Sud XI — Orsay (2009) In French.
10. Williams, N., Kosmatov, N.: Structural testing with PathCrawler. Tutorial synopsis. In: QSIC 2012
11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: SIGOPS 2009
12. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel. In: VSTTE 2010
13. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: VSTTE 2010
14. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.J.: Verification of TLB virtualization implemented in C. In: VSTTE 2012
15. Klein, G.: From a verified kernel towards verified systems. In: APLAS 2010