Journal of Universal Computer Science, vol. 16, no. 4 (2010), 543-553
submitted: 10/10/09, accepted: 21/10/09, appeared: 28/4/10 © J.UCS

Static Analysis of the XEN Kernel using Frama-C

Armand Puccetti
(CEA LIST
Centre d'Etudes Nucléaires
F-91191 Gif Sur Yvette Cedex, France
armand.puccetti@cea.fr)

Abstract: In this paper, we describe the static analysis of the XEN 3.0.3 hypervisor using the Frama-C static analysis tool.

Keywords: Abstract interpretation, static analysis, virtualisation, Linux
Categories: D.2.4

1 Introduction

Operating systems are present in our everyday lives, through desktop or embedded computing systems. Given the applications running on them as well as the nature of the corporate and personal data stored on these systems, their criticality is increasing. Therefore, we posed the following questions: how reliable are today's operating systems? Is it possible to evaluate statically such large pieces of code within reasonable amounts of time and effort. Are there any particular obstacles particular with OS?

Currently, very few tools are dedicated to the static analysis of OS code written in C and which are used in practice. For open-source Linux code we may use the commercial tool Coverity Prevent [Coverity, 09] that implements numerous heuristics and uses model checking. For Microsoft Windows the SDV toolkit [Microsoft, 09a] is used for checking the correct use of kernel API by Windows drivers, as well as the research tools VCC [Microsoft, 09b] and HAVOC [Microsoft, 09c]. However, several other tools can analyse ANSI C programs and its compiler-dependent variants. They differ mainly in terms of analysis precision and degree of automation: MathWorks Polyspace [MathWorks, 09] and Absint Astree [Absint, 09] are very automated tools based on abstract interpretation (AI) [Cousot, 77] capable of analysing large code bases; Caduceus [LRI, 09] and HAVOC make use of Hoare Logic, requiring manually specifying the C code in detail by predicates and assertions and using several semi-automated or automated theorem provers to discharge the verification conditions; VCC and BLAST [EPFL, 09] allow one to check concurrency properties (mainly safety) using model checking; many other tools [SAT, 09] are only reasoning at the syntax level instead of the semantic level (as was done in previous tools), such as Splint [Splint, 09] or QA C [PR, 09].

We decided to provide a partial answer to these previous questions by means of some application experiments on typical OS codes (see below), namely analysing them using a novel static analysis tool, Frama-C [CEA, 09]. The latter has been developed during experiments done to meet the needs of an automated analysis tool

based on AI complemented by a very expressive specification language with proof support (see below).

2 Context of the Experiments

2.1 Project OPENTC

The context of the experiments is the European research project OPENTC [OPENTC, 09] dedicated to the development of a secure Linux Operating System (OS) for desktop and server machines based on x86 family of processors. The XEN and L4 hypervisors are some of the ground stones for security, together with the TPM as well as all the other software layers that are necessary between them and their applications (management, encryption, etc.). A sub-project of OPENTC is devoted to the verification and validation of the new OS code as well as the preparation of the Common Criteria [CC, 06] Certification at level EAL5. It was decided to verify and validate (V&V) some of the most critical OS components. Given the sometimes limited availability of the documentation and support from code authors, it was decided to focus on components developed by OPENTC partners. We selected the OSLO boot loader, some hardware drivers (the Infineon TPM TSS driver [Infineon, 09] and the vGallium graphics driver [TG, 09]) and the hypervisor kernels XEN [Chrisnall, 09] and L4/Fiasco [TUD, 09]. The static analysis team was in charge of verifying successively the XEN kernel, OSLO and vGallium.
This was a challenging experiment as 1) the verification tool Frama-C and the targets were under development at the same time, and 2) the targets belong to a class of applications that has not been handled before, namely OS code. Indeed, most applications subject to formal analysis are usually embedded software.
This paper reports on the analysis of XEN as this it is based on pre-existing solid Linux kernel code, and whose results might be generalized to Linux. We will report on the complete set of targets in a forthcoming paper.

2.2 XEN

During the last few years virtualisation tools have taken an important role for server and desktop OS as they provide means to isolate several OS running concurrently on the same platform from each other. They can also provide an additional level of security and safety to the entire platform. One of the most widely distributed and used virtualisation tools is XEN [Chrisnall, 09], whose code is based on Linux kernel code. XEN has initially been developed by Cambridge University Computing Lab. (CUCL) as part of their research and has now made its way to becoming a solid and commercial product [Citrix, 09].
For the XEN experiment, we considered version 3.0.3, that was considered to be sufficiently stable at this time by the CUCL team. The source code of the kernel concerning the x86 architecture has approximately 135K lines of C. In order to remain within the project budget, it was necessary for us to consider a representative sub-part of this code. CUCL decided that we shall consider the most critical part, namely the XEN kernel API functions, also called hypercalls [Chrisnall, 09]. From a total of 21 hypercalls, the 5 most critical hypercalls were selected (see below).

The verification team's task consisted in finding as many bugs as possible within these functions and reporting the findings to CUCL for corrections.

2.3 Methodology

Experimenting with such a large application as the XEN kernel requires that a software engineering approach be taken and that details are considered only when necessary.
At first, we had to understand the architecture of the code and the structure of the different source files. XEN is similarly organized as the Linux kernel, where drivers and machine specific files are grouped in well-defined directories and the main code is functionally structured.
Documentation was very limited, especially concerning the design and the internal structure of the functions. We had to rely on the XEN developers and users mailing lists.
Analyzing the code requires determining some entry points where the analysis starts. As we decided to analyze specific hypercalls (kinds of systems calls), these functions were clearly defined in the code. Five hypercalls have been considered, whose signature is as follows:

```
int do_mmu_update(
   XEN_GUEST_HANDLE(mmu_update_t) ureqs, unsigned int
   count,
   XEN_GUEST_HANDLE(uint) pdone,
   unsigned int foreigndom)
long do_grant_table_op(
    unsigned int cmd,
   XEN_GUEST_HANDLE(void) uop, unsigned int
   count)
long do_memory_op(
   unsigned long cmd,
   XEN_GUEST_HANDLE(void) arg)
long do_domctl(
   XEN_GUEST_HANDLE(xen_domctl_t)
   u_domctl)
int do_page_fault(
   struct cpu_user_regs *regs)
void __init __start_xen(
   multiboot_info_t *mbi)
```

We included the main initialization function in the list above, namely `__start_xen`, as this appeared to be essential for building up a context used by the other hypercalls of XEN. Numerous initializations are done in this function, allowing us to understand *what* is really going on in XEN, especially about global variables.

As the reader might see above, each hypercall has several parameters whose value is *a priori* unknown. In order to constrain these parameters, we build a main function for each one, from where the analysis starts and setting some parameters

to pre-determined known values and leaves the other parameters to be any value and then calls the hypercall. In particular, the `mbi` parameter of function `start_xen` was set to

```
multiboot_info_t MULTI_BOOT_INFO_CEA = {
  flags = 107,
  mem_lower = 640,
  mem_upper = 3667483,
  boot_device = -2147287041,
  cmdline = __pa(&CEA_CMDLINE),
  mods_count = 1,
  mods_addr = __pa(&CEA_MODS_ADDR),
  u.elf_sec = {   num = 3,
                  size = 40,
                  addr = 1740800,
                  shndx = 2 },
  mmap_length = 8*sizeof(memory_map_t),
  mmap_addr = (unsigned long) (&(CEA_MMAP))
};
```

where the constant values were taken from a real machine (Pentium D, 4 Gb RAM) running XEN. Most hypercalls' parameters were kept as general as possible.

The XEN source code contains some assembly code located in specific files or in-lined in the C code. As Frama-C does not analyze assembly code, we manually transformed this code into equivalent but simpler C code, in order to keep an acceptable level of precision during the analysis.

Next, we entered an iterative process where we could launch the analyzer on the source code and examine the results. Frama-C churns out thousands of lines of messages that are then manually filtered in order to understand what happened during the analysis and see what alarms rang. The messages also contain traces of what functions were traversed during the analysis (it is in fact an abstract execution of the code using abstract variables). One can follow the interpreter's work to determine the location and causes of the problems. These were mainly due to what Frama-C calls *divergences,* meaning that the value of some variables became meaningless at some point (i.e. equal to any value of their domain, also called the *supremum).* If there is only one execution path Frama-C stops analyzing, otherwise it continues with another path. Each divergence had to be understood at the code level and adjustments were made such that 1) either Frama-C understands what the code does precisely, or 2) some pragmas, formatted as comments, are added to the code to guide the interpreter, or 3) some missing feature was added to Frama-C, such as understanding the predefined C library functions.

Iterations stopped when the code of each hypercall was entirely analyzed.

Finally, the results of the analyses were exploited to extract the alarms and interpret them in light of the source code's real execution. This requires investigating the code and also sometimes asking developers for further explanation of the code's internals when it became too obscure. Many

alarms were discarded at this stage as they were false alarms, leaving us with a much lower number of alarms (sec below).

2.4 The Frama-C Toolkit

Frama-C is an open-source static analysis tool that targets ANSI C programs as well as a subset of the C++ language. Frama-C is an open toolkit with a plug-in architecture, that allows one to connect different kinds of analysis tools together such that they can cooperate and provide precise results. Currently, the following plug-ins are provided with Frama-C:

- **Valviewer**: this is the core module thtat computes for a given function, let main, an abstract interpretation of the code and returns a set of alarms. Each alarm is a potential error and relates to a given location within the code and a set of local and global variables. For instance, in the following function

```
1   int i, t[10];
2
3   void main(void) {
4   for (i=0; i<=8+2; i++)
5      t[i]=i;
6   }
```

ValViewer produces the following warning about an out-of-bound access at line 5:

```
rte.c:5: Warning: accessing out of bounds index.
assert ( (0 <= i) && (i < 10));
```

The interpreter also provides an assertion that can be inserted into the code, but that must be discharged by other plug-ins (see below). For each variable and each location, Valviewer provides an over-approximation of the set of values taken by this variable at the indicated location. The domain of values computed is guaranteed to be correct, i.e. contains the real set of values taken by the variable during any execution. Over-approximations might therefore lead to false alarms.

- **Slicer**: this module slices the source code and produces a copy of the program that keeps only parts relevant to a given variable or location. The code obtained is generally much shorter and therefore easier to analyze by other plug-ins.
- **Jessie**: this module implements a deductive verification tool, based on Home Logic [Hoare, 69]. Each C function must be annotated by extra predicates (pre-, post-conditions, loop and data-types invariants, assertions, etc.) written in the ACSL (standing for ANSI C Specification Language [CEA, 09]) and that builds up its specifications. The Jessie module proves that the code is correct w.r.t this specification. To reach this goal, some verification conditions (VC) are computed using the WP calculus [Dijkstra, 76] and are handed over to some automatic or semi-automatic theorem provers. The code is correct iff all VC are satisfied.
- **Code browsing**: some modules help to examine and to understand the code by navigating through the data flow representation of the program, or by highlighting the locations impacted by a given code change (impact analysis) or by highlighting the locations where a given variable is used.

- **Semantic constants unfolding**: this module allows replacing expressions that Valviewer has determined to be always constant by the constant value itself.
- **Aoraï**: this is a new module under development or the verification of temporal properties written in some LTL. This is useful for reasoning about threads and shared variables.

2.5 Experimental Results

The alarms discovered were classified into several categories. Let us give the different categories used, through table 1 below.

Let us add some more explanations. Category 2 is relevant to functions to which we added some pre-condition to strengthen it. The abstract interpreter tries to satisfy simple assertions, and indicates if they are certainly true, but some assertions remain either undefined or false, which falls into this category.

We analysed hypercalls in the following order: first we spent much time with `__start_xen`, and then we analyzed the other less complex hypercalls. For each one, we keep only new errors, leaving out those already reported in previous hypercalls. The bugs discovered in the target hypercalls can be grouped into the categories as indicated in table 2 below. Their status of understanding is given by table 3 below.

Some interesting bugs were detected and confirmed by CUCL, such as:

- **Incompatible declarations** for 7 internal functions: `smp_apic_timer_interrupt`, `smp_call_function_interrupt`, `do_nmi`, `do_memory_op`, `__init cyrix_init_mtrr`, `__init centaur_init_mtrr`, and `__init amd_init_mtrr`. For instance the function `do_memory_op` is declared in file `hypercall.h` as

```
extern long do_memory_op (int cmd,
                          XEN_GUEST_HANDLE(void) arg)
```

and in file `memory.c` as

```
long do_memory_op (unsigned long cmd,
     XEN_GUEST_HANDLE(void) arg)
```

Table 1: Categories of Alarms

Categories	Name	Comments
1	Out of bounds	A variable read or written has values out of its declared domain
2	Pre-condition not satisfied	Assertion certainly not satisfied
3	Missing return statement in function	Self-explanatory
4	Incorrect retum statement	A function that declares to return void has still a return statement
5	Incompatible declaration	A function or variable signature is different from its implementation
6	Different declarations for a global	Some global item is declared several times but with a different type. Severe error.
7	Volatile global variable initialized or missing constants initializations	Volatile variables do not need to be initialized. When this is still detected, the interpreter warns
8	Constant not initialized	Self-explanatory
9	Unknown size	Sometimes variables cannot be initialized because of an unknown size
10	Addresses comparisons	Comparing addresses to fixed values in memory is something very dangerous, as addresses might change or types may change during time, so the interpreter warns
11	Divergence	The interpreter is said to diverge when at some location it bas too much imprecision (generally on some pointer)
12	Others	Other rare alarms are incompatible pointer size, uninitialized constants, etc.

- **Useless return statement**: 8 functions declare to return void but still return some value. These are `hvm_init_ap_context, __serial_rx, __put_user_bad, hvm_vcpu_down, vmx_clear_vmcs, show_stack, __enter_scheduler, and ns_write_reg`. For instance, in file `console.c:166` the body of function `__serial_rx` starts by

```
static void __serial_rx(char c, struct
 cpu_user_regs *regs)
{ if (xen_rx) return
 handle_keypress(c,regs);
  …
```

- **Incompatible global variable type:** in file `setup.c` a global variable `stack` is declared as

```
char * stack[2*STACK_SIZE]
```

where `STACK_SIZE` is a machine-dependent constant, and is also pre-declared in file `mm.c` as

```
extern char stack[];
```

which is a clear error.

One may notice a significant number of unknown alarms, mainly due to redundancies: the same error applies to a group of variables, but at several different lines, such as in the following set of messages:

```
mm.c:3408: Warning: out of bounds read. &pl2e->l2
mm.c:3418: Warning: out of bounds write. pl2e
mm.c:3421: Warning: out of bounds read. &pl2e->l2
mm.c:3433: Warning: out of bounds read. &pl2e->l2
mm.c:3433: Warning: out of bounds write. pl1e+i
mm.c:3437: Warning: out of bounds write. pl2e
mm.c:3443: Warning: out of bounds read. &pl2e->l2
mm.c:3448: Warning: out of bounds read. &pl2e->l2
mm.c:3456: Warning: out of bounds read. pl1e
mm.c:3458: Warning: out of bounds read. pl1e
mm.c:3459: Warning: out of bounds write. Pl1e
```

The detailed list of all warnings can be found in [OPENTC, 07]. The gross ratio of confirmed bugs on the total number of warnings is approx. 10%. Categories 8, 9 and 11 are present here as we had discovered some of occurrences with previous versions of Frama-C or XEN but not in current version. Categories not mentioned contain no errors.

Table 2: Number of alarms per categories

Hyper-calls	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**10**	**12**	**Total**
do_mmu_ update	24	11	2	8	7	1	27	4	1	**85**
do_grant_ table	1	1							1	**3**
do_memory _op	2							3	2	**7**
do_dom_ctl										**0**
do_page_ fault	1									**1**
__start_xen	54	10			1		4	2	3	**74**
Total	**82**	**22**	**2**	**8**	**8**	**1**	**31**	**9**	**7**	**170**

Table 3: Number of alarms per status

Categories	**False alarms**	**Unknown alarms**	**Confirmed bugs**	**Total**
do_mmu _up date	9	61	15	85
do_grant_ table	0	3	0	3
do_memory_ op	3	4	0	7
do_dom_ctl	0	0	0	0
do_page_ fault	1	0	0	1
__star_xen	7	66	1	74
Total	**20**	**134**	**16**	**170**

3 Conclusions

The analysis of the six functions (five hypercalls and the main initialization function) bas produced 170 alarms given above. The details of each alarm can be found in [OPENTC, 07] publicly available. Each of the hypercalls' analyses was produced in less than 30 minutes on a DELL Poweredge 2900 server equipped with 16 Gb of RAM. It took about one hour to analyze (partially) the code of `__start xen`.

We have classified these warnings into several categories and examined each of them precisely, using the source code, its preprocessed version and the publicly available documentation. We could conclude on 36 cases among 170 but left some unknown errors. Their investigation needs further knowledge of the XEN code and higher precision during the analysis. The latter can be achieved by reducing approximations, on tables in particular (by default a table is approximated by the 'sum' of the approximations of its elements. Using the Frama-C option `-plevel <n>` allows to adjust the size of the approximation to some higher value but with a substantially higher analysis time), and during the translation of assembly code into C.

As general conclusions we note that:

- Abstract interpretation is indeed the most promising technique to extract run-time level bugs from the code with little user assistance.
- Extrapolating the results above, 10.5 hours would be required to analyse all XEN hypercalls, which is reasonable.
- The code analysed has a ratio of approx. 1.54 bugs per KLOC. This is rather low compared to other open-source code such as Vgallium, whose ratio is approx. 10.13 bugs/KLOC. XEN can be considered as high quality code.
- In order to increase the quality of the XEN code, all bugs must be examined and, if necessary, corrected. In case of doubt, we recommend to insert all assertions generated by Frama-C, leaving their proof for later.

Acknowledgements

Our thanks go to the Commission of the European Communities, Information Society and Media Directorate for funding the OPENTC project under contract 027635.

References

[Absint, 09] Absint, The Astrée analyser, 2009, http://www.absint.de/astree/.

[CC, 06] Common Criteria for Information Technology Security Evaluation,Version 3.1 CCMB-2006-09-001, 2006.

[CEA, 09] CEA, INRIA, The Frama-C Software Verification Toolkit, 2009, http://frama-c.cea.fr.

[Chrisnall, 08] Chrisnall, D., The Definitive Guide to the Xen Hypervisor, Prentice Hall, 2008.

[Citrix, 09] Citrix, Xenserver, 2009,

http://citrix.com/English/ps2/products/product.asp?contentID=683148.

[Cousot, 77] Cousot, P., Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, POPL, 1977.

[Coverity, 09] Coverity Inc., Coverity Prevent static analyzer, 2009, http://www.coverity.com/products

[Dijkstra, 76] Dijkstra, E.W., A Discipline of Programming, Prentice Hall, 1976.

[EPFL, 09] EPFL, University of Berkeley, The Berkeley Lazy Abstraction Software Verification Tool (BLAST), 2009, http://mtc.epfl.ch/software-tools/blast/.

[Hoare, 69] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, CACM, Vol. 12, n°7, 1969.

[Infineon, 09] Infineon, The TPM Trusted Software Stack, 2009, http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6921ae011f.

[LRI, 09] LRI, The Caduceus C Verification Tool, 2009, http://caduceus.lri.fr/.

[MathWorks, 09] The MathWorks, The Polyspace verification tool, 2009, http://www.mathworks.com/products/polyspace/index.html.

[Microsoft, 09a] Microsoft Research, The Static Driver Verifier (SDV) tool, 2009, http://www.microsoft.com/whdc/devtools/tools/sdv.mspx.

[Microsoft, 09b] Microsoft Research, A Verifier for Concurrent C (VCC), 2009, http://research.microsoft.com/en-us/projects/vcc/.

[Microsoft, 09c] Microsoft Research, The HAVOC tool, 2009, http://research.microsoft.com/en-us/projects/havoc/.

[OPENTC, 07] OPENTC, WP07 Deliverable D07.2, 2007, http://www.opentc.net/deliverables2007/Open_TC_D07.2_V_and_V_report_2.pdf.

[OPENTC, 09] OPENTC, FP6 IST Integrated Project OPENTC, 2009, http://www.opentc.net.

[PR, 09] The Programming Research, The QA C tool, 2009, http://www.programmingresearch.com/QAC_MAIN.html.

[SAT, 09] Static Analysis Tools list, 2009, http://www.testingfaqs.org/t-static.html#PRLQAC.

[Splint, 09] The open-source Splint tool, 2009, http://www.splint.org/.

[TG, 09] Tungsten Graphics, The Gallium3D graphics drivers, 2009, http://www.tungstengraphics.com/wiki/index.php/Gallium3D.

[TUD, 09] Technical University Dresden, The L4/Fiasco micro-kernel, 2009, http://os.inf.tu-dresden.de/fiasco/.