Self-stabilizing Cloud Infrastructure

# Deliverable D1b - Technical proposal for hypervisor design

# 

# Introduction

The goal of this research is to improve cloud computing dependability by ensuring the self-stabilization property of cloud computing infrastructures, presenting autonomic platforms and supplying the required foundation for stabilizing applications.

The aim is to make our system capable of fighting commonly known threats that attack cloud infrastructures (e.g. Denial of Service attacks). In our architecture the hypervisor is enriched with the capabilities necessary to fight these threats.

The first report phase addresses the single host layer, mainly the hypervisor. In the previous report (D0) we introduced the importance of autonomy for cloud computing. We have also reviewed related work and industrial products related to the hypervisor level. We have detailed the settings and fault model that will be used as a basis for the coming stages. We use KVM as a minimalistic hypervisor to illustrate the application of our concept[[1]](#footnote-1). We chose KVM because it is widely used, compact, simple and open-sourced (being a part of the constantly maintained Linux kernel). The last feature allows augmenting KVM with robustness functionality in the coming stages.

The design proposed in this report adds the self-stabilization feature to the common hypervisor architecture. A complementary paper reviews the state of the art of monitoring cloud frameworks as a first stage towards extending the current work from a local node to a full cloud setting.

# The architecture of a self-stabilizing hypervisor

In this section we outline the architecture of a *self-stabilizing* hypervisor (for the sake of brevity referred to as *the architecture*). Self-stabilization means that any hypervisor execution must always end up in a safe state after a finite number of steps no matter what state it is initialized with, as in [20].

Following the requirements stated in [2], a well-formed design must be:

1. **Generic** – a design should be easily implemented in different virtualization architectures (e.g. KVM [5], VMWare [4]).
2. **Portable** – relevant software modules can be easily transferred to another machine with as few modifications as possible in the existing hypervisor architecture.
3. **Configurable** – if possible, the system actions should be governed by user-defined policies and not hardcoded in the software.
4. **Extensible**– the system should be easily augmented with external modules without being re-implemented or reconfigured.

To fulfill the first requirement (**Generic**), we describe the architecture in implementation-independent terms, making it applicable to different hypervisors. The second requirement (**Portable**) is achieved by implementing the system as a set of plugins added to the existing hypervisor architecture. Every plugin must expose certain interfaces. For example, our prototype consists of several Linux kernel modules encapsulating the self-stabilization logic and the traffic monitor.

The analysis of the inter-VM traffic and internal KVM structures contributes to the decision whether the current hypervisor state is safe. This decision is governed by user-defined rules thus making our system **configurable**. As a key requirement, enforcement of such rules should possess the self-stabilization property. More on stabilization preserving rule enforcement see in the Future Work section and in [19].

The system is able to incorporate external “plugins” that encapsulate additional security mechanisms, e.g. to protect the system from certain threats. Vespa [14] is an example for such a plugin. Vespa runs as a user application in a specially designated virtual machine.

The self-stabilization requirement is enforced by the *Stabilization Manager* (SM) that runs upon arrival of a timer interrupt and (in case of need) brings all the relevant system parts into a safe state. Note that the SM itself can be compromised by a malicious party. For example, running within an interrupt handler, the compromised SM might “hang” (for example, after a system crash), thus freezing the system execution. To bring the system into a safe state we suggest the “escape strategy” based on the watchdog[[2]](#footnote-2) , following the ideas presented in [7]. The watchdog in our architecture expects an “I'm Alive” signal from the Stabilization Manager to arrive in every (preconfigured) time slot. The watchdog periodically raises an interrupt, if no “I’m Alive” signal noticed. The watchdog component included in our system handles this interrupt by resetting the system thus bringing the latter into the safe (initial) state. Sophisticated reset rules may be specified by a user (*soft reset*, e.g. reload only certain applications or drivers (i.e. *soft reset*) This is in contrast to rebooting the entire system (*hard reset*).

Note that usually OS kernel modules are not isolated from one another, possessing the same access privilege level (e.g. ring0 in the Linux OS). Thus an intruder that controls the execution of some module will be eventually able to capture the entire kernel, attempting to incapacitate it.

Following [22], the *trusted computing base* (TCB) of a system is the set of hard- and software that enforces its security principles. In our case the TCB is the watchdog subsystem because, by virtue of bringing eventually the system into a safe state, it guarantees the security principle of our system: self-stabilization.

In order to be able to rely on the TCB, the latter is required to run without any influence[[3]](#footnote-3) from the remaining (possibly malfunctioning) system – in other words, the TCB must be *logically isolated* from the rest of the system. In our architecture the logical isolation requirement is enforced by putting the watchdog-related functionality into a separate memory region that cannot be accessed by any party operating at the kernel level. Thus the watchdog is guaranteed to be untampered by malicious agents from within the kernel. More on this issue see in the Architecture Overview section (Low Level).

Architecture Overview

Figure 1 presents the general architecture of the self-stabilizing hypervisor as a UML [3] package diagram. The usage relations between individual components are depicted by the dashed lines. The architecture components are grouped into the following packages:

Userspace

A virtual machine (VM) runs the guest OS with a number of user applications. *A virtual machine manager* (VM manager) creates and deletes VMs; suspends and resumes VM executions. For example, **Qemu** is the VM manager in the KVM architecture.

Any VM is identified by its ID (created by the VM Manager during VM creation) in any system call targeted at the hypervisor. Internally the hypervisor associates VM IDs with the states (CPU registers, memory regions) of these VMs. The VM manager launches a new thread which periodically executes a VM. Responding to the VM RUN requests, the hypervisor executes a VM by advancing the instruction pointer register in the VM state.

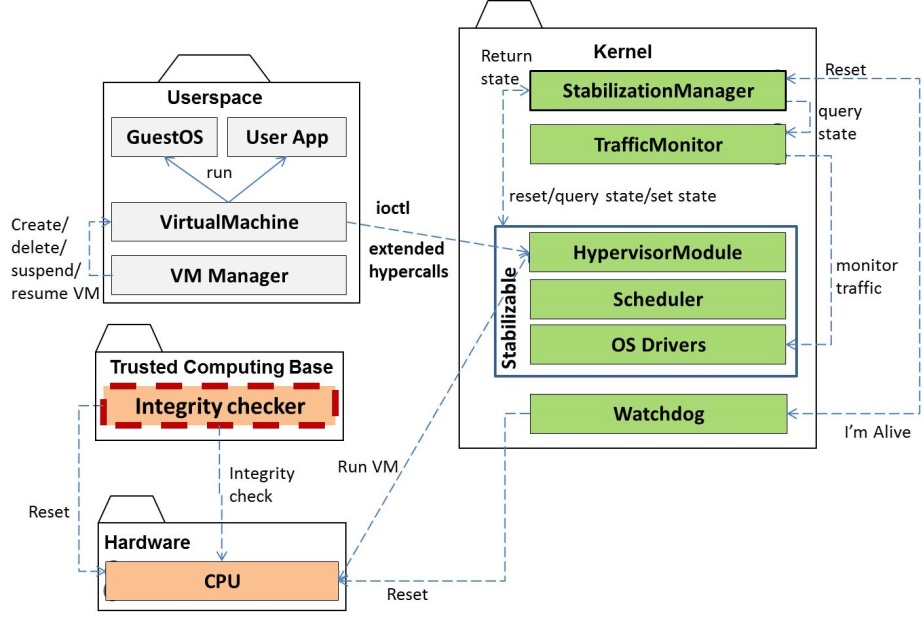


Figure 1: The architecture of a self-stabilizing hypervisor

Low-Level

This part represents the architecture components that provide direct interface to the hardware components. These components are: the watchdog integrity checker and the CPU. The hypervisor instructs the CPU to advance the instruction pointer for a scheduled VM; in such a way CPU facilitates VM execution.

As it was noted in the Introduction, the watchdog functionality comprises the system TCB so it is placed into a memory region that is out of reach of the kernel functions. This region is protected by hardware means. Hardware based protection drastically reduces the chances of an intruder to corrupt the data/code of TCB. For the sake of efficiency our TCB contains only the module that performs the watchdog integrity check. That is, if the watchdog digest (MD5 etc) does not change we assume that the watchdog is not tampered and will fulfil its duties properly. So the control is passed to the kernel-level watchdog. Otherwise the system is rebooted thus being set into a safe (initial) state. In this report we illustrate the TCB by wrapping the corresponding items by the thick read dashed lines.

Kernel

This group consists of the kernel-level components that facilitate self-stabilizing functionality.

*A virtual machine table* (*VMTable*) consists of *virtual machine entries* (*VM Entries*) that relate VM IDs to the *states* of the corresponding VMs. A VM state is represented by the CPU registers and the memory image, as in [7]. For every existing VM the OS *scheduler* runs a separate thread[[4]](#footnote-4) that invokes the *hypervisor driver*. The hypervisor, in turn, causes the physical CPU to execute the code at the corresponding VM.

The *traffic monitor* collects the traffic at the OS I/O drivers. It is done by augmenting the system calls that are responsible for processing the actions (I/O, memory accesses). The *stabilization manager* (SM) acts upon a timer interrupt, checking and enforcing the system state safety. To check the system safety SM gets the state of every architecture component. Once SM is done the signal “I am Alive” is sent to the watchdog. In case of problems (e.g. SM is stuck) these signals are not noticed by the watchdog. In this case the latter interrupts the system, causing the system to reload.

The kernel includes the watchdog module that is fired every predefined time period upon a timer interrupt. The corresponding interrupt handler checks whether the “I am Alive” signal arrived during the recent time; if yes, nothing is done because the system is believed to be alive. Otherwise the watchdog checks whether the overall system state is safe and (in case of need) resets the system.

It should be noted that even if the watchdog code/data appear to be not tampered (and the integrity checker confirms this fact), there are still indirect ways for a malicious party to mislead watchdog executions. For example, an intruder learns the frequency of “I’m Alive” signals sent to the watchdog and then paralyzes the kernel execution, itself sending “I’m Alive” signals to the watchdog. Thus an intruder “impersonates” the system. In this case the watchdog could change the protocol of exchanging ‘I’m Alive” signals, for example, using some numbers (hidden in the trusted data zone). As a future work we suggest to deal with illegal information flow from the TCB, following the suggestions outlined in [25].

Finally, the architecture supports *external plugins.* For example VESPA [14] performs sophisticates anti-virus checks and isolates the traffic from/to certain VMs that are suspected to be malicious. Vespa runs in a separate VM, getting the information required to examine the system through the hypercall interface[[5]](#footnote-5).

In the KVM architecture the hypervisor is an integral part of the Linux kernel itself. The relevant system facilities (scheduler, I/O drivers) used in the architecture form a part of the Linux OS. In contrast, VMWare [4] is not a part of the underlying OS kernel so it installs its own scheduler and I/O drivers.

Supported Security Model

Our architecture is designed to provide defense against the following threats/failures that hit the cloud infrastructures.

* **VM State corruption**. For example, a rootkit modifies the internal structures of a guest OS, preparing to expand the attack surface into the hypervisor, as in [10].
  + At every execution phase the SM checks whether the state of every VM does not change comparing to the previous phase. To accomplish the task the SM accesses VM internal structures, being a part of the hypervisor.
* **Malicious/corrupt inter-VM communication.** A typical example is a VM guest sending malware to another one [15]; this malware attacks the target host, breaching the hypervisor defense.
  + Can be provided by inspecting the traffic between VMs.
* **Greedy VM Allocation**. There are a lot of legal ways for users to get more service from a hypervisor than it is needed thus starving the latter. For example, a guest sending many short requests including sensitive instructions like CPUID causes context switches at the hypervisor [8]; in a short time the latter denies the service to other guests [8]. See also Future Work for more details.
  + Can be provided by inspecting the traffic between VMs.

Detailed Architecture

Specification Approach

To illustrate the details of the self-stabilizing hypervisor architecture we use UML class and state diagrams. In certain cases methods are invoked in a system-specific manner (for example, a system call, an asynchronous call, an interrupt handler). We use the following UML stereotypes to illustrate these specifics:

* **<<int>>** stands for the method serving as an interrupt handler.
* **<<syscall>>** stands for a system call.
* **<<async>>** stands for an asynchronous method invocation.

Userspace

Thevirtual machine manager is represented by the class *VMManager*. It is executed with certain arguments (for example, command-line ones).

The VM Manager creates and deletes virtual machines through its methods *createVM* and *deleteVM*. At the user level virtual machines are represented by the class *VirtualMachine*. When creating a VM a user submits the guest OS image and the initial configuration (e.g. the sizes of its disk and memory spaces) of the VM being created. The guest OS executed user applications.

Yet one duty of the VM Manager is suspending/resuming VMs. VMs may be resumed/suspended at a user request or by a third party (e.g. for quarantining or billing). These actions lead to suspending/resuming of threads that run the corresponding VMs.

Upon a VM creation request submitted by a user the hypervisor driver receives the CREATE control system call and returns in response the ID of the newly created VM (VMID). For a newly created VM a new thread (represented by the class *VM\_Thread*) is launched. This thread sends RUN messages to the hypervisor, addressing the related VM through its VMID. More on executing VMs can be seen in the *Collaborations/Running VMs* section.

Figure 2 illustrates the user-level collaborations.

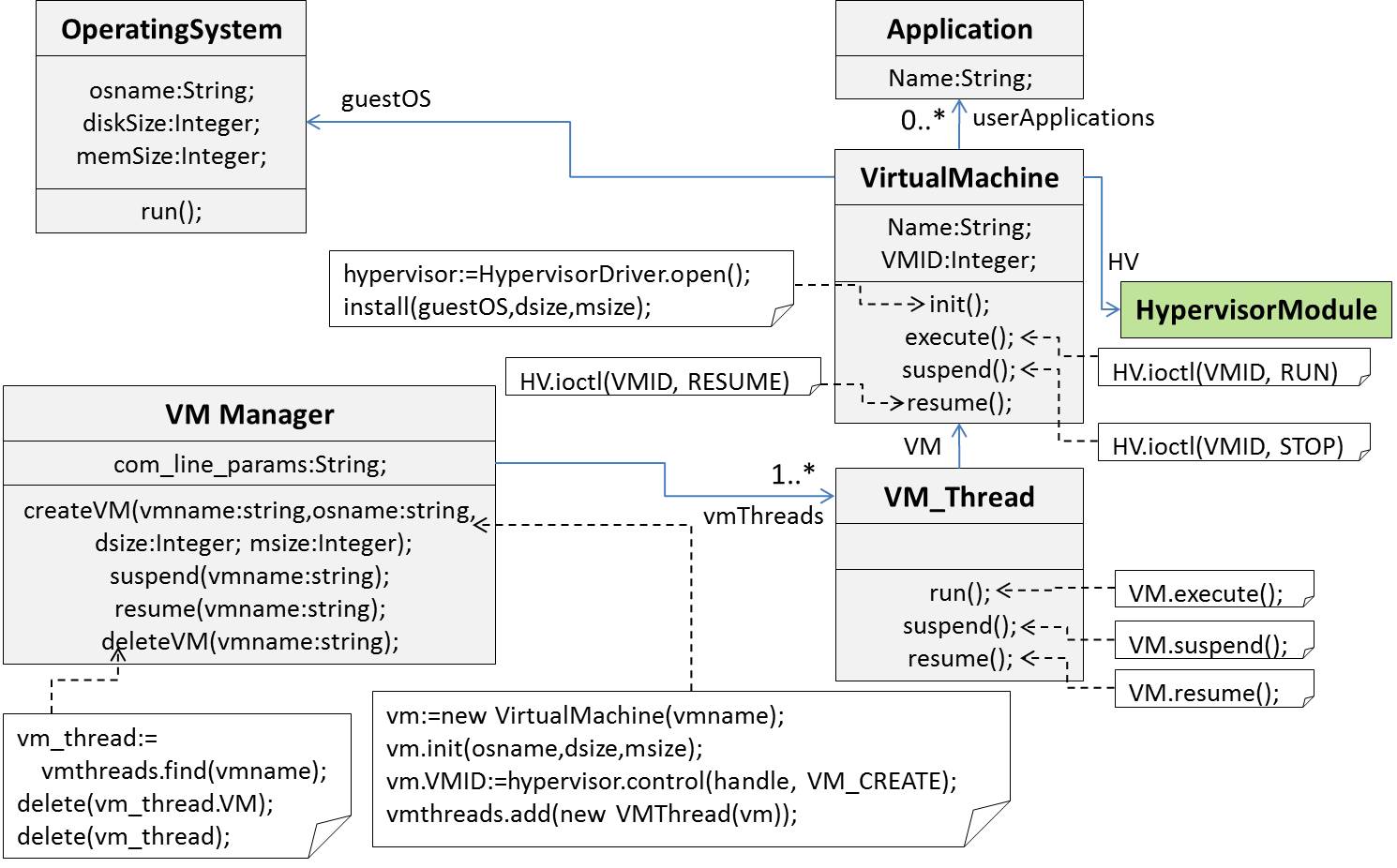


Figure 2: Collaborations in the userspace.

Hardware

This part represents the VM-system hardware and consists of the watchdog and of the CPU. The watchdog (represented by the class *Watchdog*) is characterized by the “kick period” (the time period during which at least one “I’m Alive” signal should arrive). The “I’m Alive” signals are issued by the SM invoking the *i\_am\_alive* method after ensuring that the system state is safe. This causes the watchdog interrupt (handles by the Watchdog) to be postponed. The watchdog interrupt is fired when “I’m Alive” signals do not arrive during a while. If the system contains user-defined reset policies then the Watchdog delegates the system reset task (soft reset) to the SM; otherwise the CPU reboots the OS (*hard reset*).

The CPU (represented by the class *CPU*) is characterized by a) the method *execute(Time:Integer)* that advances the current instruction pointer during the specified time period and b) by the *reset()* method that causes the complete CPU reboot, setting the zero values into all the registers. In the Linux OS this functionality is encapsulated in the *emergency\_restart* kernel API function.

Most of the modern hardware provide memory locking features – starting from external memory chips (Flicker, [24]) up to motherboard-governed memory locking (System Management Memory [16] in the Intel architecture, TrustedZone [23] in the ARM architecture). The common idea of these approaches is that the kernel boot module locks the “trusted region” containing trusted code and data by setting appropriate bits in the processor status register(s); since then the latter remains out of reach for the kernel until the next reboot. Because the kernel boot process typically runs before a malicious party gains control, the latter cannot affect the trusted part. The trusted code is launched upon arrival of a high-priority system timer interrupt. It runs in a restricted context (usually being unable to use the OS infrastructure) and has to operate the hardware directly, usually through I/O ports. A context switch typically precedes the trusted code execution;

It should be noted that the above mentioned context switch saves the full CPU status; thereafter the OS kernel is frozen until the trusted code ends. We therefore attempt to reduce the amount of trusted code. In our case, instead of specifying the whole watchdog driver as a trusted code we place only integrity checking function into TCB. Upon a timer interrupt arrival the integrity checker ensures that the watchdog is not tampered (its cryptographic digest does not change) and, if yes, passes the control to the watchdog running at the kernel level. Otherwise the system is rebooted thus being set into a safe (initial) state.

Figure 3 describes the hardware layer.

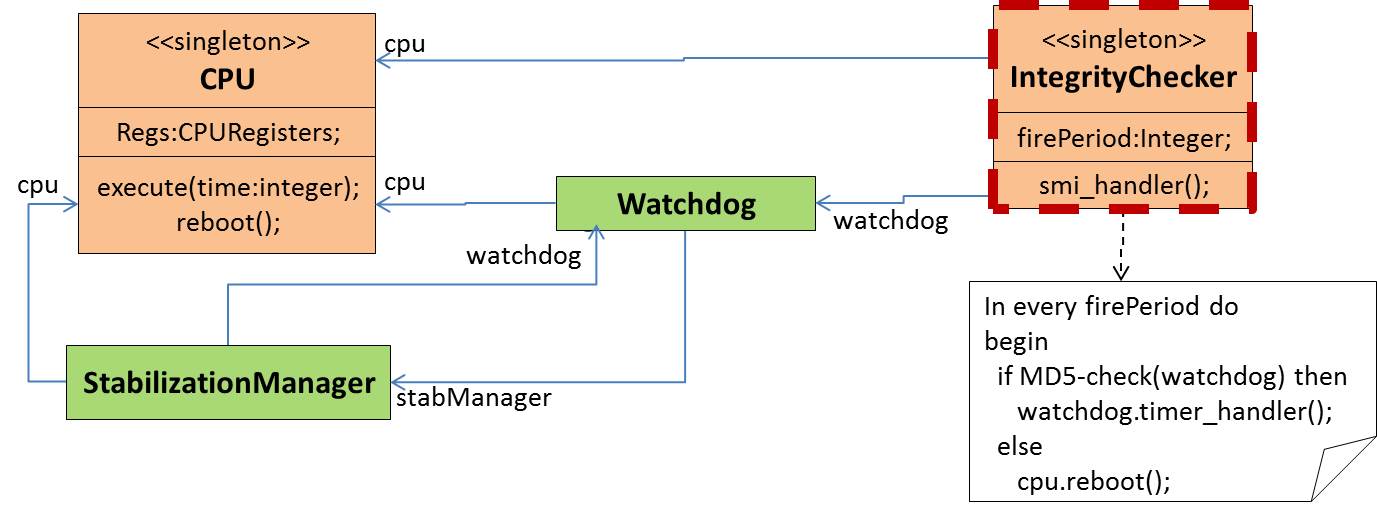


Figure 3: The hardware layer.

Kernel

The hypervisor module is represented by the class *HypervisorModule*. It possesses three (necessary) callbacks for the corresponding system calls: opening, closing a device and getting control messages.

The *system scheduler* executes all processes/threads in the system, including those which manage VM execution. The scheduler holds the collection of the objects (of the class *VMThread*) representing these threads. More on executing VMs can be seen in the *Collaborations/Running VMs* section.

The self-stabilization functionality is encapsulated in the Stabilization Manager (described by the class *StabilizationManager*). The Stabilization Manager checks and, if needed, enforces the system safety in every predefined time period within the timer interrupt handler *timer\_handler()*. The system state is comprised by the states of the following kernel components: the hypervisor, the scheduler and the OS drivers. These components are referred to as the *stabilizable* components because the SM brings them into safe states in every execution phase. To be capable of collaborating with the SM the stabilizable components must possess the following capabilities: return state, set state and sanitization. S*anitization action* cleans the traits of the previous activities (e.g. flushing buffers). A state of every stabilizable component is represented by the class *State* and its subclasses. Every such subclass represents the state of a concrete stabilizable component and (by virtue of owning the isSafe method) is capable of determining whether it is safe. Stabilizable components are provided in Fig. 4.

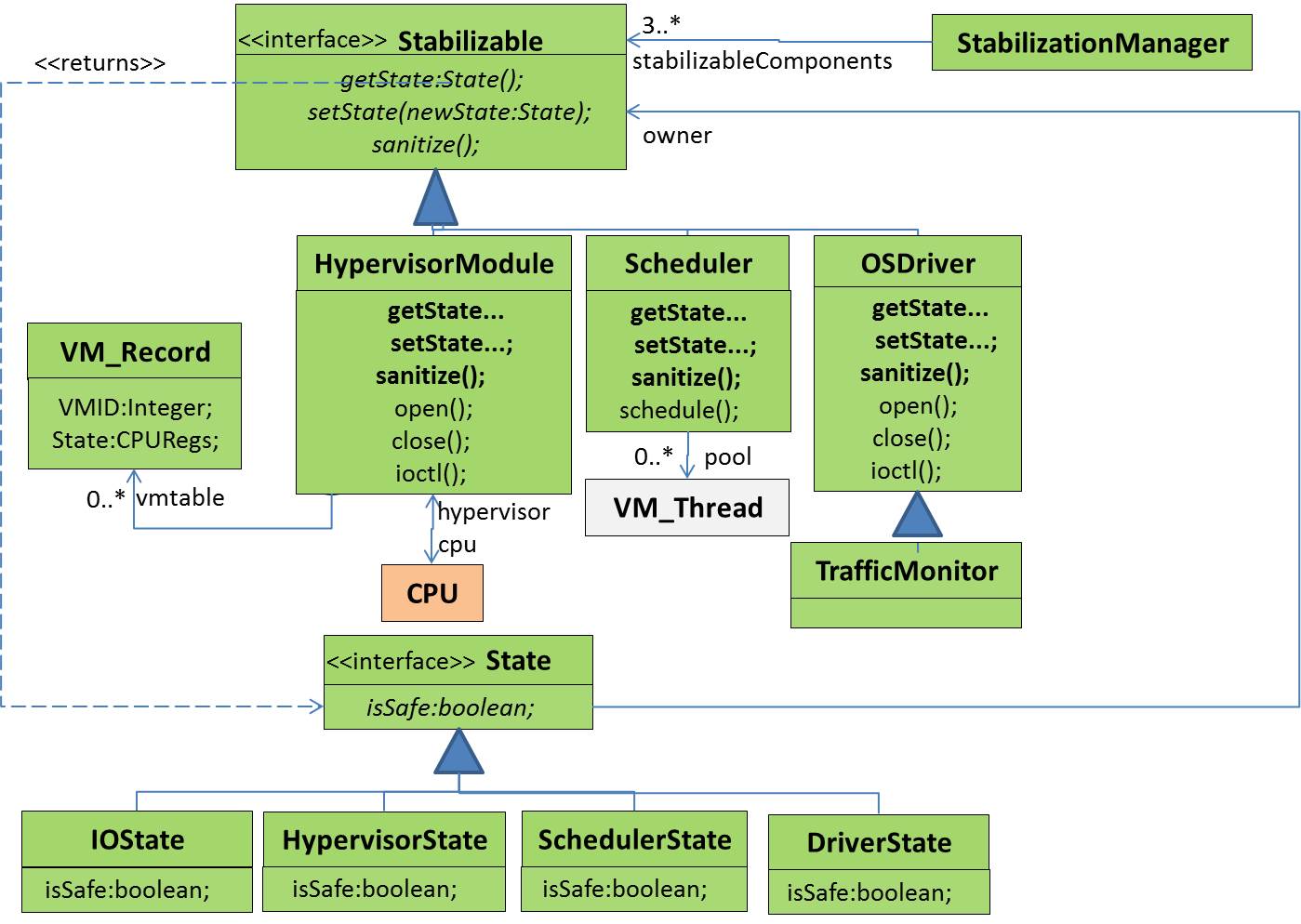


Figure 4. Stabilizable components

The system traffic digest contributes to the decision whether the system state is safe. The Traffic Monitor (see the class *TrafficMonitor*) maintains this digest, collecting the traffic from system I/O drivers. TrafficMonitor connects to the system I/O drivers in order to collect the relevant statistics. In the future we intend to incorporate various issues in system call interception like existence of VirtIO or the issues mentioned in [17].

Once the safety state check successfully passes an “I’m Alive” signal is sent to the watchdog. The latter postpones its “alarm interrupt” because the system is still functioning properly. Once in a predefined period the recent safe state is saved in the storage (represented by the class *StatesStorage*). More on enforcing self-stabilization can be seen in the *Collaborations/Self-Stabilization* section.

Figure 5 describes the kernel layer.

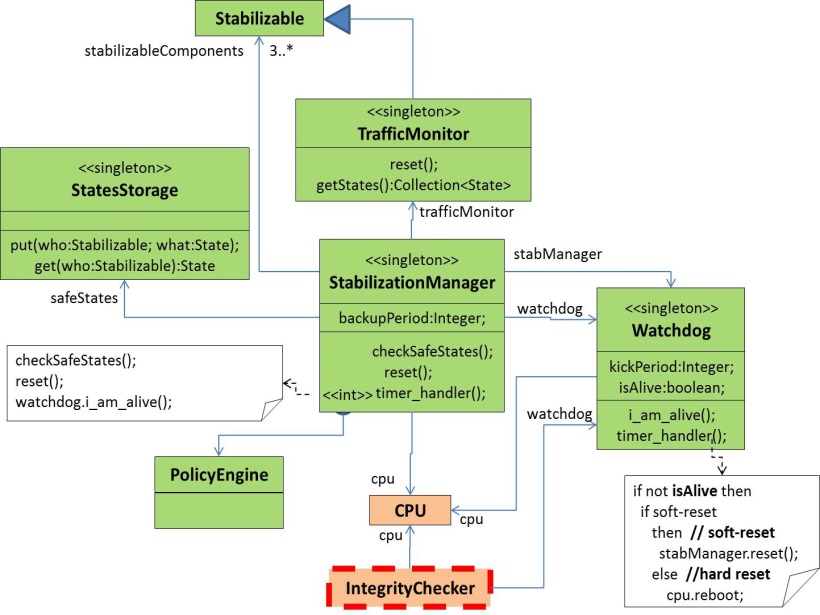


Figure 4: The kernel layer.

Collaborations

Running VMs

Assume a VM with a given identifier ID is about to run. Its thread is scheduled and the system call *ioctl(ID,RUN)*  to the hypervisor module is issued, as it was explained in the “Userspace” section. The control is passed to the hypervisor module which

1. finds the VM entry corresponding to ID (denoted by **vmtable[ID]**).
2. loads the VM state (i.e. CPU registers) from **vmtable[ID].State** into the CPU
3. runs the CPU during a certain time period
4. loads the (up-to-date) VM state back into **vmtable[ID].State**

Figure 6 illustrates this process.

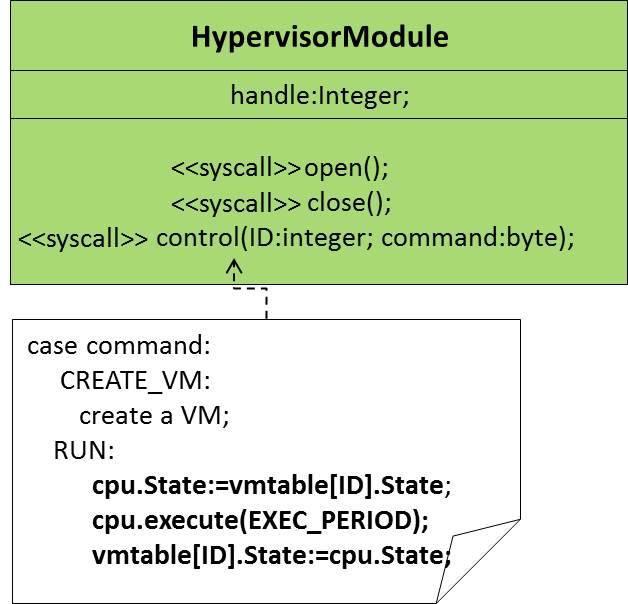


Figure 5: Executing a VM.

Bring the system into a safe state

Figure 6 presents (as pseudocode) the logic of the following methods that collaborate towards bringing the system into a safe state:

* ***checkSafeStates*** collects the states of all the stabilizable components, saving the components whose states are not safe and (if the entire system is in a safe state) creates the snapshot of the recent safe state.
* ***reset*** resets the system
* ***damage*** assesses the damage caused to the system by malicious activities

|  |  |
| --- | --- |
|  |  |

1. check system safety b) reset and assess damage

Figure 6: Bringing the system into a safe state

The first stage in checking system state safety (Fig. 6a) is collecting the states of the stabilizable components. The ones whose states are unsafe are collected into the variable ***unstableComponents***. If the states of all components in the system are safe then the recent system state is saved into the states storage.

The system reset on Fig. 6b is initiated either by the watchdog interrupt or by the SM after scanning the system state. It is referred to as the *software-triggered reset* because the *hardware-triggered reset* (more precisely, reboot) occurs when the Integrity Checker witnesses watchdog tampering (Fig.3).

If the soft reset is supported (e.g. the SM possesses user-defined reset policies) then the system damage is checked. If the damage is **severe** (e.g. critical system components like Scheduler are damaged and—or WD interrupts arrive frequently) then hard reset is done (the system is rebooted). Otherwise the SM performs the soft reset, trying to reset only the components that are in unsafe states. See Fig 6b) for details.

Enforcing self-stabilization might be time-consuming for a task executed within an interrupt handler (like the timer interrupt of the SM) so the Vespa detection/reaction activities as well as collecting states are performed asynchronously.

The system state preservation activities are summarized in Fig. 7 using state machines. The following denotations do not constitute a part of the recent UML standard [21] and explained below:

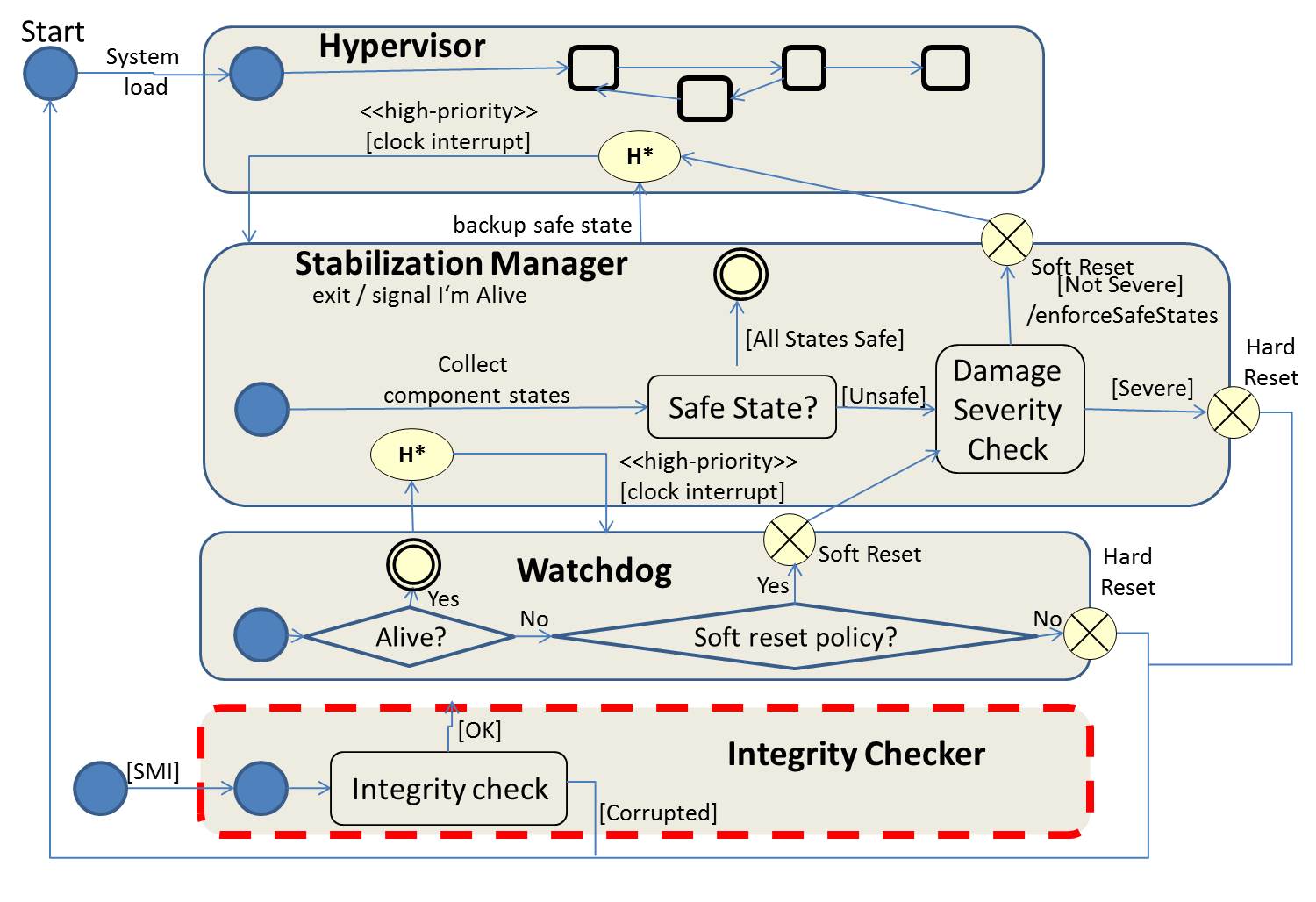
* The Integrity Checker is wrapped by the thick red dashed line; showing that we put it into the hardware-protected memory zone.
* The History pseudonode is used either as the target node (as in the UML standard [21], denoting the current execution context) or as the source node (denoting an arbitrary state in a given state machine). In summary the usage of the History node illustrates that a timer interrupt causes the execution leave the current context at any possible state. For example, the SM timer interrupt passes the control directly to the SM.
* State transition priority. The stereotype **<<high-priority>>** near the interrupt transition means that the latter occurs immediately. 

Figure 7: Self-stabilizing as a state machine

# Implementation Issues

We are implementing the prototype as a Linux kernel module. The current version has the following timer interrupt handlers installed:

1. The SM handler checks whether the system state is safe, if needed brings the system into a safe state, makes the state snapshots and in the end postpones the signal time of the second (watchdog) timer (thus mimicking the “I’m Alive” signal). This is the heart of the Stabilization Manager.
   * The implementations for most of the key functions are so far empty; only diagnostic messages are emitted. The only exception is checking whether the internal structures of the guest Oss (in our case - interrupt description table address) are changed. Any change signals possible malicious activities.
2. The watchdog handler reboots the system.

We have implemented the first version of the Traffic Monitor. It intercepts the network traffic between guests, printing out short statistics about it (e.g. number of bytes). This is done by finding the handlers for the TCP system calls (sendv, recv) in the system call table, following the Nitro approach (see [1], Section 7). To make a full-fledged traffic sensor we still have to treat special cases like the presence of VirtIO infrastructure [13] that consists of special drivers processing I/O at the guest side.

Regarding the issue of specifying some watchdog-related functions as trusted code, we note that most of the modern hardware support hardware-based memory locking. These features are quite various - starting from external memory chips (Flicker, [24]) up to motherboard-governed memory locking (System Management Memory [16] in the Intel architecture, TrustedZone [23] in the ARM architecture). Having an Intel CPU on our computers we implement the watchdog functionality as a code running in a System Management Mode (SMM). This code runs upon the arrival of a System Management Interrupt (SMI) triggered by a hardware timer.

# Future Work

Theoretical

Specifying Device Drivers Formally

Our architecture is a part of the Linux system and conceptually is not object-oriented. To bridge the gap between the conceptions of our architecture (which is not object-oriented) and UML (which describes object-oriented conceptions) is object-oriented) we used: 1) stereotyping and b) pseudocode in UML notes. Both techniques add informality to our specifications.

We intend to learn how to map OS-related concepts into the OO domain following the literature such as [6] and apply the learned approaches in our specification thus making the latter more rigorous.

Guests issuing DOS attacks

One of main challenges posed when detecting DDOS attacks is “moving attackers”, i.e. the party issuing DDOS packets (attacker) frequently changes the source IP. Source-based attacker spotting is therefore complicated. Moving attacks in cloud infrastructures can be easily mounted by placing attackers on migrating VMs (see [12] for further details on VM migration techniques). Making the Traffic Monitor capable of intercepting the information necessary for spotting DDOS attackers is our next challenge.

Comprehensive Security Model

As it was mentioned the security model supported in our architecture covers several common threats/failures plaguing cloud infrastructures: VM state corruption, malicious inter-VM communication and greedy resource allocation. We intend to study various security models used in cloud infrastructures, create a comprehensive taxonomy of VM threats and enhance our security model.

State-Based Security Checks

As it was already noted, our architecture is designed to be configurable by means of user-defined rules.

We suggest using the state machine language as in [19]. The reason behind this choice is that state machines represent a natural way to argue about the self-stabilization property (that is based on the notion of a system starting at an arbitrary state and converging to a safe state). Such state-based rules are converted into low-level specifications using a stabilization preserving compiler. So enforcement of such rules should possess the self-stabilization property.

Practical

Revising the Architecture: Isolating the Stabilization Manager

For now we admit that the components of the kernel level are not isolated from one another; in Linux terms, each component has ring0 privilege. We intend to use the ideas presented in [16] and develop the Stabilization Manager that is isolated from the remaining hypervisor architecture.

Getting Additional Information about Hypervisor Activities

In future we intend to intercept the instructions sent by the hypervisor to the underlying hardware. This information could provide us with more detailed information about malicious activities which try to exploit CPU bugs in order to achieve the goals (such as Intel SYSRET BUG [7]). Possessing this information could let us prevent malicious activities from corrupting the system state. Recall that the defense capabilities possessed by our system are so far *retroactive*, i.e. they let us recognize malicious activities by their traits.

Protecting the Trusted Computing Base

In the architecture the Watchdog serves as the Trusted Computing Base. The validity of system executions relies on the watchdog. The corrupted Watchdog might not bring the “hanging” and corrupted system into a safe state. If the Watchdog resides in a kernel then (regardless of whether interrupts are software – or hardware-triggered) its code/data can be attacked by any malicious party that succeeded to capture the hypervisor.

To tackle this issue we suggest putting the watchdog into the System Management RAM (SMRAM), a kind of memory protected using hardware means. The code in SMRAM runs in so-called System Management Mode (SMM). Breaching into SMRAM is a very complicated (though still possible) goal, as in [16]. However, the execution context of the SMM code is very restricted; for example, it cannot directly access the OS structures. Creating the “gate” that lets SMM code access the kernel structures is a first challenge in our way.

Simulating Threats

We intend to study the behavior of well-known VM threats, using rootkits encapsulating these threats as examples. Then we will implement the simulator mimicking the behaviors of these threats and check how our system reacts on them.

# References

1. Pfoh, Jonas. Leveraging Derivative Virtual Machine Introspection Methods for Security Applications. Technische Universität München, 2013. Doctoral Thesis. <http://www.sec.in.tum.de/assets/Uploads/pfoh2013b.pdf>
2. Agile Software Development: Principles, Patterns and Practices, by Robert C.Martin. Prentice Hall PTR Upper Saddle River, NJ, USA ©2003 , ISBN:0135974445
3. M. Fowler, UML Distilled: applying the standard object modeling language. Addison Wesley Longman, 1997
4. The VMWare hypervisor, [www.vmware.com](http://www.vmware.com)
5. The KVM hypervisor, [www.linux-kvm.org](http://www.linux-kvm.org)
6. Sertic, H.; Rus, F.; Rac, R., "UML for real-time device driver development," *Telecommunications, 2003. ConTEL 2003. Proceedings of the 7th International Conference on* , vol.2, no., pp.631,636 vol.2, 11-13 June 2003.
7. The Intel SYSRET privilege escalation. <http://blog.xen.org/index.php/2012/06/13/the-intel-sysret-privilege-escalation/>
8. TCP DDOS Attack Detection on the Host in the KVM VM Environment. Zhuang Wei; Gui Xiaolin; Huang Ru Wei; Yu Si, "TCP DDOS Attack Detection on the Host in the KVM Virtual Machine Environment," *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on* , vol., no., pp.62,67, May 30 2012-June 1 2012
9. Characterizing the VM-Hypervisor Attack Surface. [www.cs.princeton.edu/~diegop/data/580\_final\_project.pdf](http://www.cs.princeton.edu/~diegop/data/580_final_project.pdf)
10. Automated Defense from Rootkit Attacks. [www.vmware.com/pdf/cambridge\_**rootkit**.pdf](http://www.vmware.com/pdf/cambridge_rootkit.pdf)
11. Managing Shared Resources in Xen. <http://searchitchannel.techtarget.com/feature/Managing-Xen-shared-resources-Credit-scheduler-and-Xen-scheduler>
12. Survey\_of\_Virtual\_Machine\_Migration\_Techniques. <http://www.academia.edu/760613/Survey_of_Virtual_Machine_Migration_Techniques>
13. VirtIO : Towards the De-Facto Standard for Virtual I/O devices. ozlabs.org/~rusty/**virtio**-spec/**virtio**-paper.pdf
14. Vespa: Multi-Layered Self-Protection for Cloud Resources. Aurelien Wailly, Marc Lacoste, Herve Debar. In proceeding of: 9th ACM International Conference on Autonomic Computing (ICAC), 09/2012.
15. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections.A. Srivastava, J.Giffin. In Recent Advances in Intrusion Detection, Lecture Notes in Computer Science, Volume 5230, 2008, pp.39-58
16. HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. <http://discovery.csc.ncsu.edu/pubs/ccs10.pdf>
17. Traps and Pitfalls: Practical Problems in System Call Interposition Security Tools. Tal Garfinkel, Computer Science Department, Stanford University, <http://www.cs.berkeley.edu/~dawnsong/teaching/f12-cs161/readings/traps.pdf>
18. A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platform Main Memory. P.Stewin, TU Berlin. <http://www.google.co.il/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&ved=0CD0QFjAD&url=http%3A%2F%2Fwww.researchgate.net%2Fpublication%2F244484148_A_Primitive_for_Revealing_Stealthy_Peripheral-Based_Attacks_on_the_Computing_Platform%27s_Main_Memory%2Ffile%2F60b7d51d3ef23c9fe2.pdf&ei=UdvXUoDXF9Gu7Aajt4HgBQ&usg=AFQjCNE36yNEwSiD24F5RnOW-asklNOLGQ&bvm=bv.59568121,d.bGE&cad=rja>
19. Self-Stabilization Preserving Compiler (Extended Abstract) by Shlomi Dolev, Yinnon Haviv and Mooly Sagiv. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.74.3653>
20. S. Dolev and R. Yagel. `Towards Self-Stabilizing Operating Systems'. IEEE Transactions on Software Engineering, Vol. 34, No. 4, July/August 2008 (TSE)
21. The Current UML Specification, <http://www.uml.org/#UML2.0>
22. A Trusted Computing Base for Embedded Systems. J.Rushby, Proceedings of the 7th DoD Security Conference, 1984.
23. TrustedZone, <http://www.arm.com/products/processors/technologies/trustzone/index.php>
24. **Flicker**, [https://sparrow.ece.cmu.edu/group/**flicker**.html](https://sparrow.ece.cmu.edu/group/flicker.html)
25. J. Wang, K.Sun, A.Stavrou. An Analysis of SMM-based Integrity Checking Systems and Evasion Attacks. The CS Department , George Mason University, Tech. Report.

1. The abbreviation *VMM* – Virtual Machine Monitor is also used [↑](#footnote-ref-1)
2. Watchdog hardware is available on all the modern computers. [↑](#footnote-ref-2)
3. Neither directly (by tampering code/data) nor indirectly (by learning the TCB behavior and delivering improper data to mislead the TCB). An example of indirect influence is a malicious party impersonating the watchdog by sending “I’m Alive” signals directly to the watchdog hardware (even if the system, including the watchdog software is compromised). [↑](#footnote-ref-3)
4. In reality a VM may contain several virtual CPUs (VCPUS) and a thread is allocated for each CPU, but for the sake of simplicity we assume that a VM contains a single VCPU [↑](#footnote-ref-4)
5. The standard hypercall interface provided by well-known hypervisors is too restricted to supply sufficient information about the host OS so we consider an option to enrich it, obtaining the *extended hypercall interface*. [↑](#footnote-ref-5)