**Self-Stabilizing Virtual Machine Hypervisor Architecture for Resilient Cloud**

(Extended Abstract)

Alexander Binun, Mark Bloch, Shlomi Dolev,

Martin Ramzi Kahil, Boaz Menuhin, Reuven Yagel+

Dept. of Computer Science, Ben-Gurion University of the Negev, Israel  
+ and Azrieli – Jerusalem College of Engineering, Israel

Thierry Coupaye, Marc Lacoste,

Aurélien Wailly

Orange Labs, France

*Abstract*—This paper presents the architecture for a self-stabilizing hypervisor able to recover itself in the presence of Byzantine faults regardless of the state it is currently in. Our architecture is applicable to wide variety of underlying hardware and software and does not require augmenting computers with special hardware. The actions representing defense and recovery strategies can be specified by a user. We describe our architecture in OS-independent terms, thus making it applicable to various virtualization infrastructures. We also provide a prototype extending the Linux-based hypervisor KVM with the self-stabilizing functionality. These features allow augmenting KVM with robustness functionality in the coming stages and moving to cloud management system architectures such as OpenStack to support more industrial scenarios.

Keywords- hypervisor, IaaS, self-stabilization, resilience

# Introduction

The core entity used for facilitating virtualization in cloud computing infrastructures is the *hypervisor* (also referred to as the *VMM*, *Virtual Machine Monitor*) as in [2, 4, 6, 7, 27, 22, 3, 16, 17, 18, 19, 21]. Being a basic part in the virtualization infrastructure, the hypervisor is the most attractive target for attackers. The (steadily rising) complexity of hypervisors ([7]) and unlimited privileges of hypervisors over virtual machines ([23, 29]) aggravate the situation: a successful attack against the hypervisor almost certainly brings the whole system down. In the meantime, threats evolve faster than hypervisor defense mechanisms. Thus a significant part of attacks against the hypervisor succeed. This happens because proactive detection of malicious activities often requires the logic of security modules to be guided by threat-specific behavioral patterns. Detecting threats following the symptoms of (already compromised and malfunctioning) system is a more generic task as in [30].

Trying to keep the situation under control, designers augment hypervisors with additional security functions that are intended to fight new threats. This functionality can set an additional security layer threats attempt to compromise (for example, sandboxing of individual drivers [6] or even running the whole hypervisor in an additional VM [4]). Another way to fight malicious activities is sanitizing the internal structures of (possibly already compromised) hypervisor as in [29] thus mitigating the malicious actions. In any case, once a threat captures the system, the latter faces severe danger, possibly crashing.

Here comes our novelty. Admitting that threats evolve faster than the corresponding defense mechanisms, we shift the focus from preventing/mitigating of malicious activities to building a system that is capable of recovering gracefully after threat attacks and regaining stable behavior. We refer to recovery as the existence of (at least) the possibility for restarting from an initial (stable) state. Graceful recovery is a recovery that tries to keep the system requirements during convergence and converge fast to a stable state, possibly by rolling back (or forward) restarting the system from the stable state that is nearest in the execution history (which can be built by snapshotting and consistency checks prior to reloading). In contrast to proactive defense, this approach does not require the security logic to be guided by threat-specific behavior and thus can be easily adapted to new threats.

We also leverage on the solid background in the area of self-stabilizing systems. Following [9, 12, 10, 11] a system is *self-stabilizing* if every of its executions always ends up in a stable state after a finite number of steps no matter what state it is initialized in. Stable (also called safe) states are distinguished from unstable ones solely through the concrete application logic, namely, any system execution that starts in a stable state exhibits the desired application (also called task) behavior.

**Main Idea**. Materializing the above idea, we suggest a novel self-stabilizing hypervisor architecture. Once in a time period, a special routine, the *stabilization manager (SM)* examines the hypervisor, checking whether the latter is in a stable state. If needed, the system is set into a safe (stable) state. The corresponding enforcement actions range from simple, coarse-grained ones (like restarting suspicious VMs or even the entire hypervisor) to fine-grained ones; for example, stopping individual guest applications with suspicious behavior). Upon success, the SM notifies the *system watchdog* (explained next) by sending the *AmAlive* message. Even if the SM itself is corrupted (e.g., due to a successful attack) and thus does not fulfill its duties, the watchdog ultimately reveals this (either by the absence of *AmAlive* messages or following a system integrity check) and restarts the system. The watchdog module is tamper-resilient because it is write-protected (by virtue of residing in the hardware-protected memory) and is triggered by hardware timer. The watchdog thus forms the trusted computing base of our system. We provide a conceptual description of the architecture and argue that it is self-stabilizing.

As a proof of concept we implement the SM as a separate Linux kernel module collaborating with the KVM infrastructure [13]. We use KVM as a minimalistic hypervisor to illustrate the application of our concept. We chose KVM because it is widely used, compact, simple and open-sourced. The last feature allows augmenting KVM with robustness functionalities in the coming stages.

**Paper Organization.** Section II outlines the architecture of the self-stabilizing hypervisor and main design goals. In section III we are arguing that our system is self-stabilizing and describe the prototype we are developing as the proof of concept. We also detail the attacks we tested so far. Finally, concluding remarks appear in Section IV. Due to space limit we point to related works alongside the discussion.

# Self-Stabilizing Hypervisor Architecture

In this section we outline the architecture of a self-stabilizing hypervisor. Following the requirements stated in [5], a high-quality design must be:

* *Generic*. A design should be easily implemented in different virtualization architectures (e.g., KVM [13], VMWare [28]).
* *Portable*. The software modules can be easily transferred to another machine with as few modifications as possible.
* *Configurable*. If possible, the system actions should be governed by user-defined policies and not hardcoded in the software.
* *Extensible*. External modules should be easily plugged into the system, possibly with minor configuration actions.

The design is *generic* because we describe the architecture in platform-independent terms, making it applicable to different hypervisors. *Portability* is achieved by implementing the system as a set of plug-ins added to the existing hypervisor architecture. Each such plug-in must expose certain interfaces. For example, our prototype consists of several Linux kernel modules that encapsulate self-stabilization logic and monitor relevant intra-system events such as traffic between VMs. Collecting relevant data, our system decides whether the current state is stable and, in case of need, brings itself to a stable state. This process is expressed by user-defined rules, in the form of guarded commands thus making our system *configurable*. Eventually, our system is *extensible* because it can incorporate external plug-ins that contain additional security mechanisms, such as defense against certain threats. The VESPA self-protection framework [29] is an example for such a plug-in. It runs as a user application in a specially designated virtual machine.

Following [12] we make the following assumptions about the system behavior:

* *Byzantine faults at the software level*. The system state (including the CPU and the OS internal structures) may become arbitrarily corrupted, i.e., assigned any possible value due to malicious activities or transient faults.
* *Single security context*. All the OS kernel data and code reside in the same security context so an adversary that gained access to any kernel spot is capable of reaching and incapacitating the entire kernel. User data/code can be then reached.

Self-stabilization is enforced by the *Stabilization Manager* (*SM*) that is triggered by a timer interrupt and (in case of need) resets the system, thus setting the entire system into an initial (i.e., stable) state. Due to the *Single Security context* assumption the SM itself may be compromised by a malicious party (that got into the system at some earlier stage) thus getting incapable of fulfilling its job.

To get the SM out of an unstable state we suggest a strategy based on a watchdog, following the ideas presented in [12]. The strategy consists of two stages:

* *The SM is not damaged*. The watchdog expects an *AmAlive* message to be sent by the SM in every (preconfigured) time period, after the SM enforces system stability. The absence of an *AmAlive* message means a problem, causing the watchdog trigger the system integrity check. This check fails if the system is tampered; in this case the watchdog resets the entire computer, setting a system into an initial (stable) state.
* *The SM is damaged.* Assume an adversary incapacitates the SM. To make the watchdog believe that all is OK, an adversary learns the frequency of *AmAlive* messages and sends them to the watchdog. The system gets out of such a situation after the watchdog is activated by a hardware timer signal arriving through an out-of-band hardware channel. The watchdog then runs the system integrity check which fails if an adversary resides in a system, leading to the system reset.

In both cases the reset is followed by installing a fresh copy of the system from a read-only memory region, neutralizing all changes made by an adversary.

In summary, self-stabilization is guaranteed if the watchdog functions correctly, i.e., it will ultimately run and its code is not tampered. So the final hurdle on the way to achieve self-stabilization is to find a way to guarantee correct functioning of the watchdog.

The solution we suggest to protect the correct functioning of the watchdog is to use a hardware watchdog mechanism so that any Byzantine behavior at the software level cannot affect the watchdog. We put the integrity checking code into a separate ROM region that is write-protected by hardware means. Integrity checking will thus be un-tampered. It will ultimately occur because a hardware signal that triggers the watchdog is out of the control of the CPU and thus cannot be influenced by an adversary.

## Architecture Overview

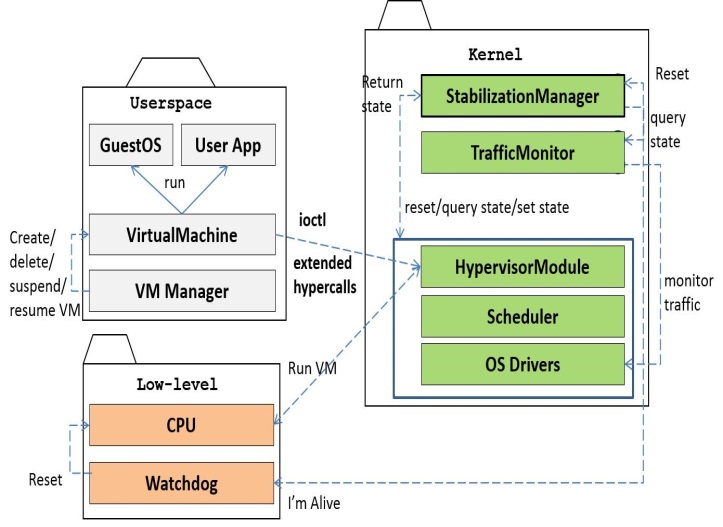
Figure 1 presents the general architecture of the self-stabilizing hypervisor as a UML package diagram. The usage relations between individual components are depicted by the dashed lines. The code that is assumed to be trusted is wrapped by the thick red dashed line in Figure 3 and Figure 5.

Figure 1: The components of the self-stabilizing hypervisor

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 KVM [13]. 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.

***Low-level***. The components that provide direct interface to the hardware components (the watchdog integrity checker and the CPU). The hypervisor instructs the CPU to advance the instruction pointer for a scheduled VM thus facilitating VM execution. The integrity checker comprises the system TCB and is placed into a write-protected memory region.

***Kernel.*** The kernel-level components that facilitate self-stabilizing functionality. A *Virtual Machine Table* (*VM Table*) 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 [12]. For every existing VM the OS scheduler runs a separate process that turns to the hypervisor driver. The latter, in turn, lets the physical CPU execute the code at the corresponding VM. The *Traffic Monitor* collects the traffic at the OS I/O drivers. The *Stabilization Manager* (SM) acts upon a timer interrupt, checking and enforcing the system state safety. The SM gets the state of every architecture component, checking whether they are stable. Once finished the SM sends an *AmAlive* message to the watchdog. Upon the absence of *AmAlive* messages within the SM period the watchdog verifies the system integrity, possibly causing a system reboot.

Last but not least we note that external plug-ins can be added to the system. For example, VESPA [29] performs sophisticated anti-virus checks and isolates the traffic from/to certain VMs which are suspected to be malicious. VESPA runs in a separate VM, getting the relevant system information through the hypercall interface.

## Security Model

Our architecture is designed to provide defense against the following threats and 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 [30]. At every execution phase the SM checks whether the state of every VM is changed comparing to the current state. 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 as in [26]; the malware attacks the target host, breaching the hypervisor defense. The 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; in a short time the latter denies the service to other guests as in [22]. The defense can be provided by monitoring all sensitive but not privileged requests (like CPUID, RDTSC).

## Detailed Desrciption

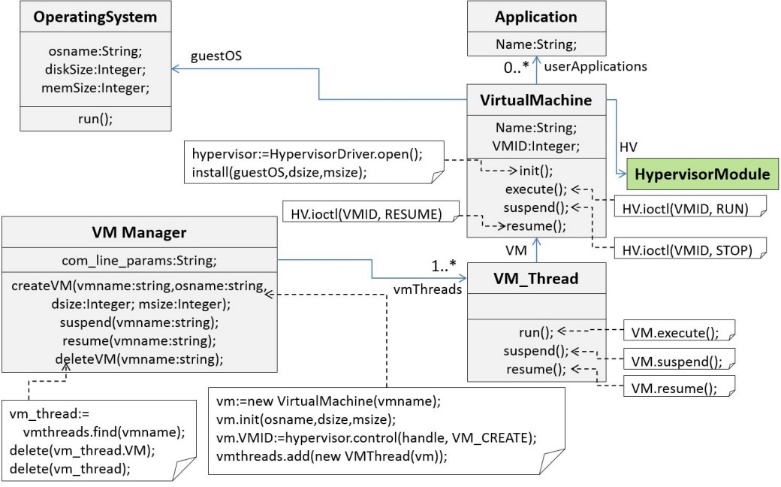
To illustrate the details of the self-stabilizing hypervisor architecture we use UML class and state diagrams. In certain cases methods are invoked in a system-specific manner (e.g., a system call, an asynchronous call, an interrupt handler).

The following subsections provide the detailed description for all the packages included into the architecture.

*C.1 Userspace*

The virtual machine manager is represented by the class *VMManager*. It creates and deletes virtual machines through its methods *createVM* and *deleteVM*. At the user level, VMs 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 executes user applications.

Upon a VM creation request, 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 *VMThread*) is launched. This thread sends *RUN* messages to the hypervisor, addressing the related VM through its VMID. The VM Manager may suspend/resume VMs at a request from a third party (e.g., for performing quarantine or for billing purposes). Figure 2 illustrates the user-level collaborations.



*C.2 Hardware*

Figure 2: User-level collaborations in our architecture

This part represents the hardware abstraction of the architecture: the system integrity checker and the CPU. The *integrity checker* (represented by the class *IntegrityChecker*) is characterized by the **fire period** (the time period in which the hardware triggers the integrity checker code). If the integrity checker reveals modifications in the OS code, the entire system is reset (through a reboot); otherwise the reset task of detecting modifications is delegated to the watchdog.

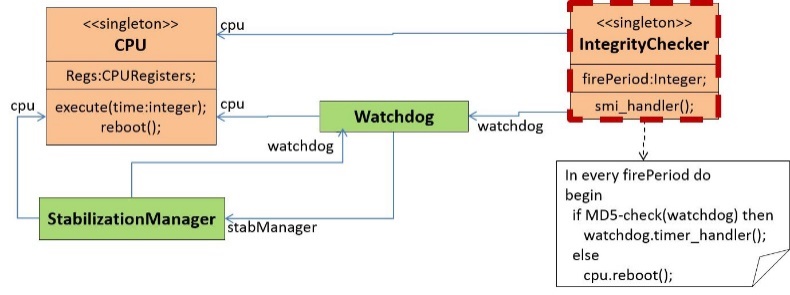
The integrity checker resides in the memory that is write-protected using hardware means. For example, in the Intel architecture this is the memory associated with the System Management Mode (SMM) [2]. This memory can be accessed only by the BIOS itself. TrustZone [1] provides the same features in the ARM architecture. We are working on Intel computers and therefore use the SMM memory.

The OS kernel cannot tamper with the code we inject, thus unless an attacker hacks/changes the SMM code, even on reboot, the Integrity Checker code cannot be changed by any application besides the BIOS itself.

Because the kernel boot process typically runs before a malicious party gains control, the SMM region remains unreachable for adversaries. The SMM code is launched upon arrival of a hardware timer interrupt (System Management Interrupt, shortly SMI) which arrives through an out-of-band channel that is not governed by the kernel. A context switch typically precedes the trusted code execution. Since a SMI is generated by a hardware device and its behavior cannot be changed a hacker gaining control over the system cannot change the SMM or configure the hardware device that triggers the SMI.

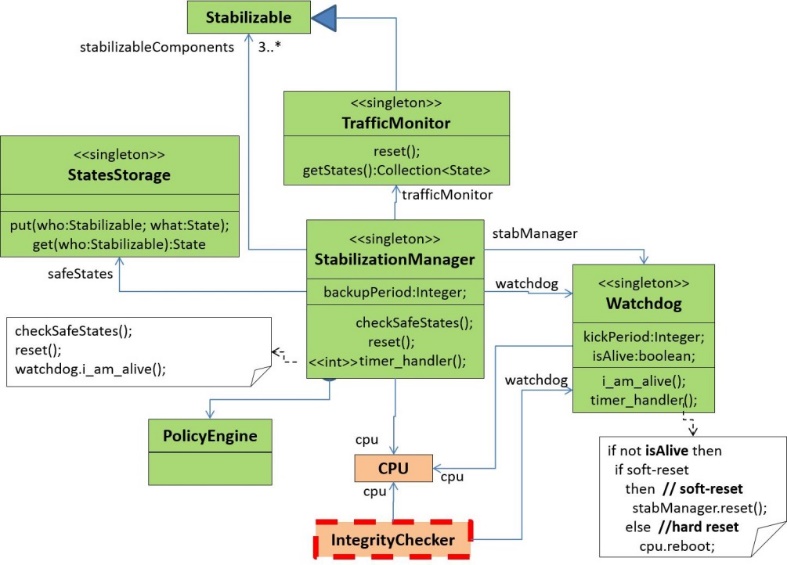
It should be noted that the above mentioned context switch saves the full CPU status, freezing kernel execution until the trusted code ends. Therefore checking the integrity of the whole system is highly inefficient. One of improvements suggested by us is to let the Integrity Checker verify that only the watchdog is not modified. The watchdog, in turn, asynchronously verifies the integrity of the entire system. Figure 3 describes the hardware layer.

Figure 3: The hardware



*C.3 Kernel*

***Stabilizable components***. This subsection describes the architecture components whose state is critical for proper hypervisor operation. The goal of the SM is to make sure these states are stable. That is why these components are referred to as the *stabilizable* components. They include:

* The *hypervisor* which 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* is represented by the class *Scheduler*. It 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.
* The *components related to the system traffic*. I/O drivers and the Traffic Monitor (the class *TrafficMonitor*). The latter monitors the traffic at the system I/O drivers.

The self-stabilization functionality is encapsulated in the SM which is described by the class *StabilizationManager*. The SM checks and, if needed, enforces the system stability in every predefined time period; the corresponding actions are done within the timer interrupt handler *timer\_handler*. Stabilizable components are provided in Figure 4.

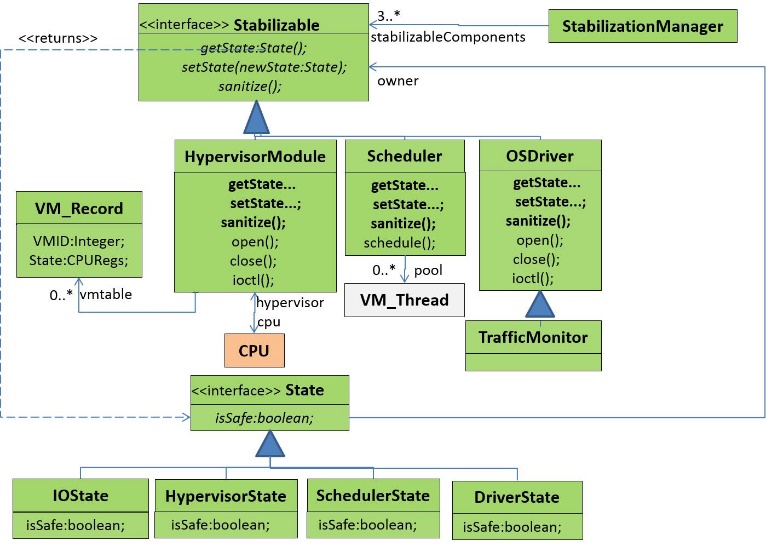


Figure 4: The stabilizable component

***Bringing the system into a stable state***. In order to collaborate with the SM the stabilizable components possess the following capabilities: *get state*, *set state,* and *sanitization*. When checking the system state the SM retrieves current states from every stabilizable component within the system; if needed, malfunctioning components are set into new (stable) states. Sanitization 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 stable.

Figure 5:The kernel level collaborations

The actions for verification and enforcement of system state stability represent defenses against various attacks (e.g., DOS attack, rootkit activity). Defenses may be state-based (i.e., do not kill the VM issuing DOS attacks during several timer iterations but kill it afterwards). We specify these defenses using the language based on guarded commands. Guarded commands are chosen because they are a natural way to express state-based actions as in [11].

There are sophisticated actions for intelligent system reset (referred to as *soft reset*); for example, suspend or kill only malfunctioning VMs instead of rebooting the computer. If no soft reset actions are available in a given situation then the computer is rebooted (*hard reset*). Once in a predefined period the recent stable state is saved in the storage (represented by the class *StatesStorage*) and can be used to recover the recent stable state during the soft reset.

The watchdog (represented by the class *Watchdog*) is characterized by the *kick period*, i.e., the time period during which at least one *AmAlive* message should arrive.

*AmAlive* messages are sent by the SM invoking the *i\_am\_alive* method; this happens after ensuring that the system state is stable. *AmAlive* messages causes the watchdog timer to be postponed. The watchdog timer thus fires an interrupt when *AmAlive* messages do not arrive during a while. The watchdog searches for reset strategies; soft reset actions are executed if supported, otherwise the computer is rebooted.

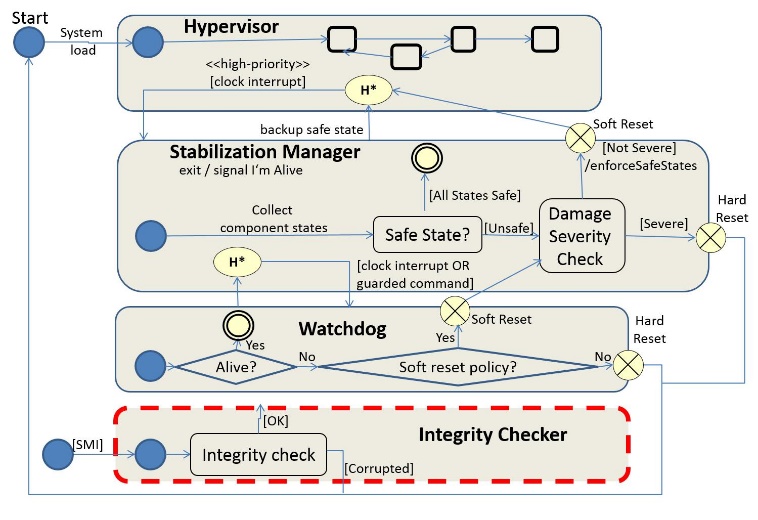
It should be noted that bringing the system into a stable state is too time-consuming for a task executed within an interrupt handler so the SM collects and sets individual states in the asynchronous manner.

Figure 6 summarizes the entire life cycle of the Stabilization Manager using three state machines. The following notations do not constitute a part of the recent UML standard [14] and are 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 pseudo-node serves either as the target node (as in the UML standard, denoting the current execution context) or as the source node (denoting an arbitrary state in a given state machine). A History node illustrates that a timer interrupt causes the execution to leave the current context at any possible state.
* State transition priority. The stereotype **<<high-priority>>** near the interrupt transition means that the latter occurs as soon as possible.

To check whether the system state is stable, the Stabilization Manager uses the following sources of information:

* *System logs*. For example rootkits send sensitive information through the network, causing periodic traffic bursts. System logs can help detect rootkits with quite sophisticated scenarios. However, an adversary might modify system logs.
* *Tracking system model changes*. A system model represents security-related structures typically altered by rootkits (e.g., process table, driver jump table). Without relying on (possibly damaged) system logs, the SM checks whether these structures underwent changes.



The Stabilization Manager brings a system in a consistent state as part of a periodic consistency checks following a timer interrupt. If a guarded command running in the SM context detects severe problems (e.g., inconsistencies in the system scheduler), then the system reset may be fired immediately. For example, a rootkit trying to inject code into the kernel code area (usually write-protected) triggers a protection fault. The latter is intercepted, triggering immediate system reset.

Figure 6: The entire self-stabilization process summarized

# Proof of Concept

## A Attack Scenarios

Our prototype implements two scenarios that illustrate particular cases of attacks against hypervisors: denial of service and rootkit activities. In both scenarios, defense policies are expressed in the terms of guarded commands [8]. Following [11] guarded commands are a natural way to express state-based actions.

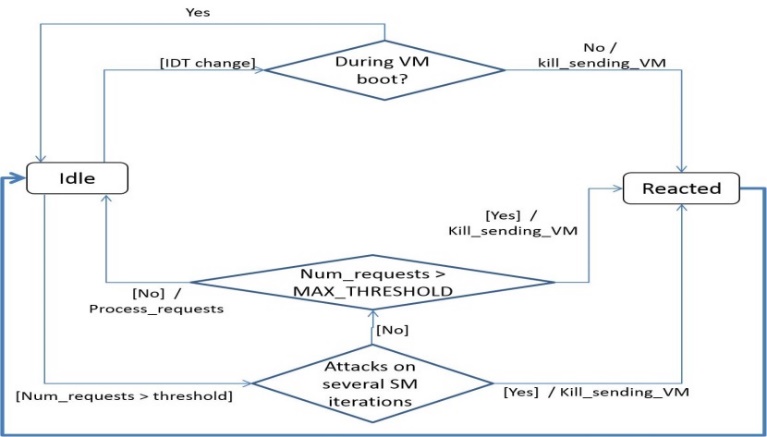
***Denial Of Service (DOS)*** In general, we distinguish between two kinds of denial of service attacks:

* *Local*. A VM sends a lot of sensitive requests (e.g., CPUID *http: //en.wikipedia.org/wiki/CPUID*) to the hypervisor. Handling sensitive (but not privileged) instructions a hypervisor (e.g., KVM) usually gains the execution context, handles these instructions and passes the control back to the sending VM. Such context switches starve hypervisor out of computational resources, see [22] for more details.
* *Network-based*. A VM floods the network with requests starving the TCP-related resources.

At the hypervisor level, our system handles DOS attacks as follows: at every SM iteration, we count the number *NR* of DOS requests. If *NR* exceeds a certain threshold *Threshold,* we suspect a VM to run an attacker. If *NR* remains above *Threshold* during several SM iterations, we conclude that a DOS attacker continues to operate and kill the VM running an attacker. If *NR* becomes very high, exceeding the threshold *MaxThreshold* (which is significantly bigger than *Threshold*), we kill the VM executing an attacker immediately.

***Rootkit activities*** Our system detects modifications of the Interrupt Description Table (IDT) that contains the entries to software interrupt handlers (also to system calls because a system call is a particular case of software interrupt). If a party running at some VM tries to modify the IDT of the corresponding virtual CPU we assume that this party is malicious and kill the corresponding VM. This is because hooking the system control flow (in particular interrupt handlers) is the preparatory stage towards breaching into the hypervisor as in [24, 3, 30].

Figure 7 provides the state machines that illustrate both defense policies.



## B Implementation Issues

The prototype is implemented as a Linux kernel module. Its

Figure 7: The state machines for the defense policies implemented in our prototype

current version has the following timer interrupt handlers installed: the SM timer and the watchdog timer. The SM timer handler does the following:

* In every execution it checks whether the system state is stable and in case of need brings the system into a stable state. After these actions are finished the watchdog timer signal is postponed, thus illustrating the *AmAlive* message.
* The system state is assumed to be stable if no IDT modifications and no DDOS attacks are noticed.
* If the system state is unstable, then the SM brings it to a stable state as it is described in the previous section.
* Once in several executions of the SM timer handles the system state. The watchdog timer reboots the OS, issuing the Linux system call *emergency\_restart*.

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 (i.e., *sendv*, *recv*) in the system call table, following the approach of [23, Section 7]. To make a full-fledged traffic sensor, we still have to treat special cases like the presence of VirtIO infrastructure [25] that consists of special drivers processing I/O at the guest side.

We are running our prototype on an Intel architecture so we are using the memory associated with System Management Mode (SMM, [16]) to provide the hardware-based write-protection of the Integrity Checker code. Most of the modern hardware support hardware-based memory protection. The integrity checker code runs in SMM and is triggered by the hardware timer. Such a signal is guaranteed to arrive regardless of software behavior.

It should be noted that in order to detect rootkits the whole OS kernel (not just the Stabilization Manager) should be checked for changes. The inefficiency of this operation is aggravated by the fact that the Integrity Checker runs it synchronously, freezing the whole OS. We suggest the following improvement that is supposed to make the stabilization check more efficient:

* Let the Integrity Checker verify the integrity of the Watchdog solely because the Watchdog module is relatively small.
* The Watchdog module launches an asynchronous system state check after its integrity is confirmed.

In order to minimize delays caused by the SMM code execution, extensive system state checks may be carried out at idle periods, possibly night time.

# Concluding Remarks

In this work we have suggested a novel self-stabilizing hypervisor architecture which is capable of performing a wide range of stabilizing actions starting from coarse-grained ones (like killing suspicious VMs) to fine-grained ones (e.g., stopping guest applications whose behavior seems to be suspicious). We have provided the conceptual description of the architecture as well as the proof of concept for its self-stabilization property. As a proof of concept we implement the self-stabilization functionality in the KVM hypervisor [13]. The design proposed in this report adds the self-stabilization feature to the common hypervisor architecture. We have also reviewed existing industrial products related to the hypervisor level as a first step towards extending the current work from a local node to a full distributed cloud setting. Finally, we note, that our implementation was tested with several initial denial of service and rootkits attacks and exhibit low overhead in consistency monitoring and guarded commands execution for recovering the system to regain consistency. We plan to expand experiments using known test workloads, e.g. ‎[30] while collecting results.

##### Acknowledgment

This work is supported by Orange Labs under external research contract number 0050012310-C04021. We also thank the support of the Rita Altura Trust Chair in Computer Science.

##### References

1. ARM. Trust Zone. Available at *http://www.arm.com/products/processors/technologies/trustzone/index.php*.
2. A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 10, pages 38-49, New York, NY, USA, 2010. ACM.
3. A. Baliga and L. Iftode. Automated Defense from Rootkit Attacks Available at *www.vmware.com/pdf/cambridge\_rootkit.pdf*.
4. M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The TURTLES project: Design and implementation of nested virtualization. In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation OSDI 10, pages 1-6, Berkeley, CA, USA, 2010. USENIX Association.
5. R. C. Martin. Agile Software Development: Principles, Patterns and Practices. Prentice Hall, Upper Saddle River, NJ, USA, 2003.
6. G. Coker. Xen Security Modules. Available *at http://mail.xen. org/files/summit\_3/coker-xsm-summit-090706.pdf*.
7. P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan,
   1. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 11, pages 189-202, New York, NY, USA, 2011. ACM.
8. E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM, 18(8):453-457, Aug. 1975.
9. S. Dolev. MIT Press, March 2000.
10. S. Dolev. Self-Stabilization. AIAA Journal of Aerospace Computing, Information, and Communication (JACIC), 1:253-255, June 2004.
11. S. Dolev, Y. Haviv, and M. Sagiv. Self-stabilization preserving compiler. ACM Trans. Program. Lang. Systems, 31(6):22:1-22:42, Aug. 2009.
12. S. Dolev and R. Yagel. Towards self-stabilizing operating systems. IEEE Transactions on Software Engineering, 34(4):564-576, July 2008.
13. T. K. group. The KVM hypervisor. Available at *http://www. linux-kvm.org*.
14. T. O. O. M. Group. The UML specification. Available at *http: //www.uml.org*/.
15. H.Sertic, H. Rus, and R.Rac. UML for real-time device driver development. In ConTEL 2003. Proceedings of the 7th Interna-tional Conference on Telecommunications, volume 2, pages 631-636, 2003.
16. A. S. Jiang Wang, Kun Sun. An analysis of system manage-ment mode (SMM)-based integrity checking systems and evasion attacks. Technical Report GMU-CS-TR-2011-8, Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030-4444 USA, 2011.
17. M. Lacoste. What Does the Future Hold for Hypervisor Security. Available at *http://workshop13.tclouds-project.eu/ abstracts/what\_does\_future.pdf*.
18. C. Liu and Y. Mao. Inception: Towards a nested cloud architecture. In Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing , Berkeley, CA, 2013. USENIX.
19. J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for T minimization. SIGOPS Oper. Syst. Rev., 42(4):315-328, Apr. 2008.
20. P. D. Patel, M. Karamta, M. D. Bhavsar, and M. B. Potdar. Ar-ticle: Live virtual machine migration techniques in cloud computing: A survey. International Journal of Computer Applications, 86(16):18\_21, January 2014. Published by Foundation of Computer Science, New York, USA.
21. M. Pearce, S. Zeadally, and R. Hunt. Virtualization: Issues, secu-rity threats, and solutions. ACM Comput. Surv, 45(2):17:1-17:39, Mar. 2013.
22. D. Perez-Botero and R. Lee. Characterizing the VM-Hypervisor Attack Surface. Available at *www.cs.princeton.edu/~diegop/ data/580\_final\_project.pdf*.
23. J. Pfoh. Leveraging Derivative Virtual Machine Introspection Methods for Security Applications. PhD thesis, Technische Universität München, The Computer Science Department, 2013.
24. T. X. project community. The Intel SYSRET privilege escala-tion. Available at *http://blog.xen.org/index.php/2012/06/ 13/the-intel-sysret-privilege-escalation/*.
25. R. Russell. VirtIO:Towards the De-Facto Standard for Virtual I/O devices. Available at *ozlabs.org/~rusty/virtio-spec/ virtio-paper.pdf.*
26. A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection, RAID 08, pages 39-58, Berlin, Heidelberg, 2008. Springer-Verlag.
27. P. Stewin. A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory. In RAID, pages 1-20, 2013.
28. VMWare. The VMWare hypervisor. Available at http://www. vmware.com.
29. A. Wailly, M. Lacoste, and H. Debar. Vespa: Multi-layered self-protection for cloud resources. In Proceedings of the 9th Inter-national Conference on Autonomic Computing, ICAC 12, pages 155-160, New York, NY, USA, 2012. ACM.
30. Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In Proceedings of the 16th ACM Conference on Computer and Communications Security, pages 545-554, New York, NY, USA, 2009. ACM.