Live-upgrading Hypervisors: A Study in Its Applications

EMIL MENG,[†1] MITSUE TOSHIYUKI,[†1] HIDEKI EIRAKU,[†1]
TAKAHIRO SHINAGAWA[†1] and KAZUHIKO KATO[†1]

Hypervisors are currently designed with a single purpose in mind, leading to the development of software which is loaded and remains static since boot time. We propose a method where a hypervisor can be switched out with another without the requirement of a system reset. This provides the ability to refresh or even upgrade a hypervisor without any significant interruption to the user of the VMs/VMM. This process completes within 50 milliseconds and allows for the enforcement of security policies, live-upgrading of hypervisors, and faster VMM development.

1. Introduction

Large organization's IT infrastructure continually seek to reduce the downtime of their machines. While unplanned downtime is usually mitigated through redundant servers, planned downtime for system software is unavoidable. Virtual machine monitors, like commodity operating systems, require maintenance and newer versions are often released that address security issues or bug fixes, which requires updates. Typically, an update to virtual machine monitor requires a system reset, which also forces any guest OSes running to also be shut down.

We also envision that large organizations will install security-based virtual machine monitors, such as SecVisor[9], on regular user workstations in the future. When these virtual machine monitors are deployed across an entire organization, updates need to be scalable and as unintrusive to the user as possible. Rebooting an entire organization's workstations should be avoided whenever possible.

We have created a proof-of-concept that allows a newer hypervisor version to be switched with the current one without the requirement of a system reset. We call this process a *live-upgrade.* Live-upgrading hypervisors can be used in many different applications including: upgrading the hypervisor, faster hypervisor development, and even policy enforcement. Furthermore, our switching process is very fast, and causes minimal interruption to any guests it runs. We have achieved a live-upgrade in less than 50 milliseconds, with the guest's service outage lasting less than 3 milliseconds.

CPU makers have also noted this demand and have focused their attention on offering more features for virtual machine monitors at the hardware level. Intel's Trusted Execution Technology[3] (TXT, formerly known as LaGrande Technology, LT) provides new features in hardware such as only permitting a trusted and measured virtual machine monitor to be executed without requiring a system reset. While TXT is not required for live-upgrading hypervisors, it is useful for ensuring the security of the virtual machine monitor layer.

The rest of this paper is structured as follows: section 2 discusses related works. Section 3 describes the background of the hypervisor that we use. Section 4 details the design of our hypervisor switch process and section 5 discusses implementation issues. Section 6 delves into the applications of our hypervisor switching concept and we evaluate our method in section 7. Future works are discussed in section 8 and we conclude in section 9.

2. Related Works

Microvisor[6] is a virtual machine monitor that can dynamically enable and disable itself from virtualizing resources to its guests. They propose that a virtual machine monitor should be able to shut itself down when it is not needed for performance reasons. Our approach also shuts down itself, but with the only purpose of bringing another hypervisor online.

SecVisor[9] is a hypervisor whose purpose is to protect the guest OS's kernel code from malicious attacks. It utilizes AMD's SVM[7] extensions to provide hardware protection for memory and to track kernel entries and exists. Similarly, our switching method focuses on using a hypervisor that is security-centric and with

†1 University of Tsukuba

the use of TXT, it can be executed in a trusted manner. However our focus is on the security of the hypervisor while SecVisor's security is focused on the guest's kernel. Furthermore SecVisor has no live-upgrade functionality.

Kourai et al.[5] focuses on rejuvenating the Xen[1] virtual machine monitor without a power reset for performance purposes. However, his technique is limited to only rebooting an identical hypervisor. Furthermore, their method further requires suspending all domUs, and a full reboot of the dom0. Our method has the capability of upgrading the switched VMM and does not require any action from the guest VM and executes much quicker.

3. Background

In this section, we will go into the background of our hypervisor, the secure virtual machine monitor, and Intel's Trusted Execution Technology. Those who are already familiar with these subjects are invited to skip these sections altogether.

3.1 Our Hypervisor

Our hypervisor was originally designed with security in mind, and is implemented such that the guest VM sees almost all of hardware directly, with the exception of storage and network devices. The guest OS therefore believes that he sees hardware as is, while in reality the hypervisor is receiving clear-text communications on the storage and network devices and encrypts it before sending them to the real storage or network devices.

Our hypervisor design does not require a guest VM to know about it. This complete separation between the virtual machine monitor and its guest allows for greater security. Currently, live-upgrading our hypervisor requires interaction through the guest via `vmcall` instructions, though this mechanism can be switched freely.

3.2 Intel Trusted Execution Technology

We intend to use Intel's Trusted Execution Technology (TXT) in our implementation in order to heighten the security of our system. This technology is based off of the Trusted Computing Group's[10] (TCG) Trusted Platform Module[11] (TPM) and uses it to discriminate against VMMs. The main purpose of TXT is to make a trust decision on whether or not to allow a virtual machine monitor to boot. An administrator must initially take a cryptographic hash of a virtual machine monitor and load that into non-volatile memory in the TPM. TXT then has a special set of instructions to load a new VMM, and during this time, it checks to make sure that the target VMM matches one that is trusted. If it does not, then the system is reset, as TXT will not execute untrusted code. Since this trust decision is made in hardware instead of software, it is considered more secure.

4. Hypervisor Live-upgrade Design

Our hypervisor live-upgrade technique was designed with type-II virtual machine monitors in mind. These are virtual machine monitors that are completely independent of a host operating system and separate from the guest OS it manages. Furthermore, we limit our discussion to only virtual machine monitors that are capable to execute a guest OS without modifications to the guest OS's kernel. In other words, the guest OS has no idea that it is in a virtualized environment and believes that it is in control of the entire machine.

Furthermore, we limit our design to switch between virtual machine monitors of the same family and architecture. This means that we limit our goal to switch one version of a virtual machine monitor to another version of the same virtual machine monitor. To further clarify, we do not consider switching between heterogeneous VMMs such as from Secure VM to Microvisor.

The interface in which hypercalls are provided to the guest OS must not change, or else applications depending on that interface will cease to function properly. It should be noted here that the the hypercall interface must be limited to only user-space calls. This is a direct result to our type-II virtual machine monitor rule. Since the guest kernel does not know that it is being virtualized, it should not communicate through the hypercall medium.

From here on in, we will use and define a virtual machine monitor that meets the restrictions stated above in this section as a *hypervisor*.

Our goal is to have an executing hypervisor be able to switch itself with another hypervisor in the same family without significantly affecting its guest OS's service. This means that a new hypervisor needs to be loaded without a system reset. This present some unique challenges and we will delve into what considera-

tions need to be made in order to achieve this functionality. Furthermore, our design can be applied to all virtual machine monitors that meet our hypervisor description, and is not specific to a single hypervisor.

One of the most important aspects of switching between hypervisors is to preserve the state of the guest OS and the hypervisor itself. Using our hypervisor definition, preserving the state of the guest OS is a relatively simple task. The guest OS's resources (such as memory, its device's states, CPU state, etc.) cannot be modified or else its state will be fundamentally altered. The hypervisor must be aware of any changes it makes in the guest OS's domain, and those changes need to be reverted before returning control to the guest OS.

The hypervisor itself must be aware of its own state as well. Any stateful information that cannot be initialized by a new hypervisor must saved by the departing hypervisor. Furthermore, this state information (hypervisor and guest OS) must not be lost between the switch from one hypervisor to another and needs to be stored in a non-volatile location.

Since the new hypervisor will be loaded into the same exact location as to where the previous hypervisor is loaded, an independent switching process is needed. If the previous hypervisor attempted to directly load a new hypervisor, the copy process would overwrite code that was currently being executed which would cause improper execution. Therefore, the outgoing hypervisor needs to call the switching code, whose purpose is to copy the new hypervisor into the old hypervisor and jump into the new hypervisor's entry point.

5. Hypervisor Live-upgrade Implementation

While our method can be implemented by any hypervisor that meets our definition, we chose to use the Secure VM as our hypervisor of choice. We chose Secure VM due to the fact we are most intimate with its design and implementation. Furthermore, the Secure VM's state is currently simple, which makes it an ideal hypervisor to test and create a proof-of-concept hypervisor live-upgrade.

Live-upgrading hypervisors can be broken down into three distinct and separate stages. The first stage is the *hypervisor copy*, followed by a *hypervisor switch*, and is ended with a *hypervisor initialization*. In this section we will detail each of these stages and how we implemented them.

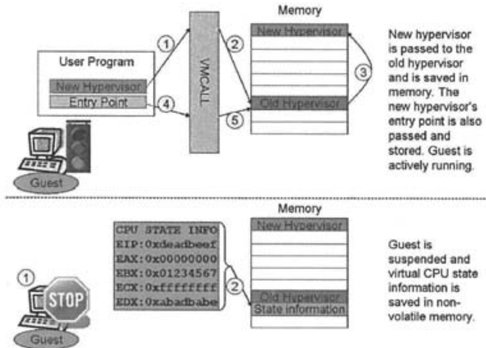

Fig. 1 Hypervisor copy stage

The first thing that needs to be done in order to do a live-upgrade between hypervisors is to provide the running hypervisor with a new hypervisor. This is achieved in the hypervisor copy stage. We created a user-space application that provides the new hypervisor through the hypercall interface. In Secure VM, this interface is defined through the `vmcall` instruction found on Intel VMX capable chipsets. When the guest OS executes the `vmcall` instruction, the hypervisor assumes control of the physical CPU and takes action based on what was passed in the virtual CPU's registers of the guest OS. This is how we pass the new incoming hypervisor to the running hypervisor. Data is passed to the hypervisor on a pass-by-copy basis, and is repeated multiple times until the new hypervisor is copied in its entirety. Furthermore, the entry point of the new hypervisor also needs to be passed via the hypercall interface.

After the hypervisor receives all the information it needs about the new hypervisor, it suspends the guest OS from executing. Since the hypervisor has control of the CPU, we know that the guest is not executing and is in a suspended state. At this point, we preserve the guest OS's CPU state by saving all of its control and general registers. This data is stored in a structure that is located in a non-volatile memory address and supports variable-length data storage. Usually, you would have to save the state of the hypervisor itself, but in our hypervisor, it is not necessary as there is no state that needs to be preserved as of this writing.

Next, the independent switching code is copied from within the hypervisor to its final

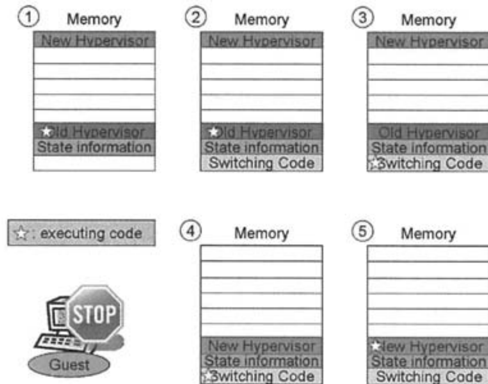

Fig. 2 Hypervisor switch stage

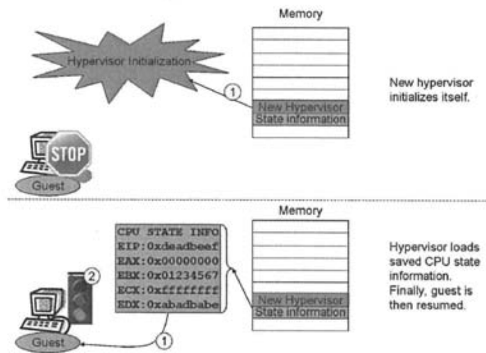

Fig. 3 Hypervisor initialization stage

location. It needs to lie outside of the hypervisor because the new hypervisor will overwrite the old one, and if the instruction pointer points to somewhere that is overwritten, the execution becomes unpredictable and useless. Finally, the hypervisor jumps into the switching code providing information about the location of the new hypervisor, its size, and its entry point.

The next stage, the hypervisor copy, copies the new hypervisor over the existing hypervisor, effectively overwriting it. Before we attempt to do this, we have to create and initialize our own page tables and stacks as the ones in the running hypervisor will be overwritten with junk data. Therefore, the switching code must implement its own simple memory management, which in our case is identical to the current hypervisor and the subsequent hypervisor. Once this is completed, the switching code executes the copy and jumps into the new hypervisor at the entry point provided by the previous stage.

The final stage, hypervisor initialization, differs only slightly to the normal boot process. First, the normal boot process' entry point is disregarded as we are not executing the hypervisor from boot, but rather a system that has been initialized from boot at least once. We achieve this by creating a new function that only initializes the things it needs to and skips initializations that need to be done only once by the bootstrapping hypervisor. This is then entry point that was passed in the first stage.

The biggest change between bootstrapping the system and initializing a once-booted subsequent hypervisor is that instead of initializing the guest VM's processor to a constant state, we reload its general and control registers to exactly what they were before the switch took place. In other words, the virtual CPU's state is exactly the same after the switch as it was before the switch.

Currently our implementation only switch implementation only supports single-threaded execution. In the context of switching hypervisors, it is critical that the second and third stages are run serially and not in parallel. A single processor may only be active during this critical section, and all other processors must idle until they are reinitialized by the new hypervisor. This is achieved through inter-processor interrupts in modern day operating systems, a mechanism that allows processors to communicate and interrupt each other. However, in the case of Secure VM, the inter-processor interrupt table is untouched and given exclusively to the guest OS, and therefore we cannot use it in the hypervisor. Due to this restriction, our hypervisor switching implementation currently supports only single-processor machines.

6. Applications

Switching between hypervisors without significantly interrupting the guest OS offers many different applications ranging from refreshing the hypervisor, upgrading the hypervisor, enhance hypervisor development, and enforce security policies. We will detail each of these scenarios in the rest of this section.

6.1 Hypervisor Refresh

Most production hypervisors are designed to be booted once and run for long periods of time before being reset. This leads to the software aging problem[7] where software performance degrades. Moreover, this degradation occurs in the hypervisor itself, which in turn could possibly cascade the degradation to any virtual ma-

chines it is in charge of.

The traditional solution to this problem is to reboot the entire machine, but that causes significant interruption to the guest OS. A reboot typically ranges on the order of minutes, and a service interrupt of that magnitude negatively affects the user's experience. However, with the live-upgrade approach, the hypervisor can be rebooted very quickly, and its state will be the same as if it were bootstrapped. At this timeframe, interruption to the guest OS is minimal and users do not perceive the downtime.

6.2 Hypervisor Upgrade

An even better solution to the software aging problem, especially when it is caused by programming errors such as memory leaks, is to upgrade the hypervisor itself. Our live-upgrade method is not restricted to only a single-version hypervisor, and therefore we can fix bugs in the hypervisor that lies in a production environment without causing a significant service interrupt. Furthermore the same can be applied when security updates to the hypervisor that are released.

We also envision that hypervisors will become ubiquitous in the future and that virtual machine monitors will be used on all computers, likely in regards to securing the guest OS. In large organizations, supporting these virtual machine monitors must be scalable. It would be a very costly endeavor to have an IT team go to each machine and reset it in order to execute a new hypervisor. Rather, with the live-upgrade method, a team can simply broadcast a live-upgrade command and have all the hypervisors replaced in a timely fashion. We believe that if such a feature was offered to IT managers, it would be received warmly and would accelerate the rate of which hypervisors are used on regular user's workstations.

6.3 Hypervisor Development

One of the most interesting, and currently the most used application of the live-upgrade process as of this writing is the ability to debug the hypervisor by switching the hypervisor from one to another. When one can freely switch a hypervisor on-the-fly, it opens many powerful different debug processes that would be much slower and less effective if it were not on-the-fly.

On more than one occasion, we have used this application to optimize code, and quickly test small changes to see whether it is correct and effective. This can include things such as rewriting inline-assembly, changing a function's prototype, adding or removing printing statements, and so on. Furthermore, developers of the hypervisor can switch between a production and development hypervisor on the fly. For example, if a developer were using the system and suddenly wanted more information about the running hypervisor, he could simply switch to a development hypervisor which outputs more debugging information. Conversely, if a user is using a development hypervisor, and finds that it is too slow, he can freely switch to a production VMM to see if that is a viable workaround.

6.4 Enforcing Security Policies

Our live-upgrading method combined with the security features offered by TXT offers a unique and unorthodox method of creating and enforcing security policies. Traditionally, policies are enforced through modules that are outside of the base system, but loaded on the fly. This makes the software smaller and easier to confirm its correctness. It also allows greater flexibility. However, when policies are enforced from within the hypervisor, live-upgrades are available, and TXT is used to attest it, interesting properties emerge.

Given a policy that is enforced by both a modular construction and a monolithic construction as described above, and looking from the viewpoint of TXT, two different and distinct identities can be cryptographically produced. However, if you change both policies such that they differ from the original, the monolithic hypervisor will generate a unique and distinct cryptographic hash, but the modular hypervisor's identity will not change. In other words, if you switched from policy A to policy B, TXT would not be able to differentiate between the two if the policies were enforced modularly.

When TXT can differentiate between whether a VMM is enforcing a policy or not, the user and administrator can have more faith that the correct VMM is running. On the contrary, if a modular hypervisor were loaded, users and administrators can have faith that the correct VMM is running, but there is no evidence at the hardware level that the correct policy is being enforced. Furthermore, a modular hypervisor's policies are more likely to be able to be changed during run time, whereas a monolithic hypervisor's policies are more likely to remain static.

Policy changes in a modular hypervisor are relatively easy to enforce since they are designed from the beginning to be loaded and un-

loaded. In a traditional monolithic hypervisor, this would be impossible without severe interruption to services, as it would require a reboot. However, with our live-upgrade method, one can easily build a new hypervisor with a different policy and use that hypervisor without interfering with the guest OS. Furthermore, this new hypervisor can be cryptographically attested by TXT to ensure that the new hypervisor conforms with the correct policy the platform is designated to run.

One setback to this approach is the need to keep track of multiple hypervisors, as each one with a different policy will have a different cryptographic hash. If an organization creates policies at an individual platform level, the number of hypervisors that need to be maintained will not likely scale. However if these policies are created so that they have general properties, common to many platforms, it can be managed.

7. Evaluation

We evaluated our switching method based on the time it takes to complete the live-upgrade process as well as its interruption to the operating guest VM. These are the only areas where the live-upgrade method impacts. We conducted a micro-benchmark measured within the hypervisor itself, and a macro-benchmark, measured from an independent machine.

Our test environment is a machine running an ICH8 chipset with an Intel Core 2 CPU running at 2.66 GHz. It has 2 gigabytes of memory and is running the 2.6.20.7 Linux kernel.

Our first benchmark is executed from within the hypervisor itself. This micro-benchmark measures the time it takes to execute each of the three stages discussed in our design section, which include the hypervisor copy, the hypervisor switch, and the hypervisor reinitialization. Accurate time measurements were calculated by the `rdtsc` instruction and stored in memory. Since the time for the calculation of the timestamp itself is negligible, and the results aren't read and interpreted until after the switch is completed, it is safe to presume that it does not add significant overhead to the measurements. The results are listed in table 1.

In the three separate stages, the first does not cause a service interrupt to the guest OS, but the second and third stages do. This is because the first stage is a user application that uses the hypervisor's hypercall interface to pass a new hypervisor to the running one. Since it

Stage	clocks (time)
Hypervisor copy	76342900 (28700 usec)
Hypervisor switch	162710 (61 usec)
Hypervisor initialization	7081820 (2662 usec)
Total switching time	83587430 (31423 usec)
Total interrupt time	7244530 (2723 usec)

Table 1 Switching stage times

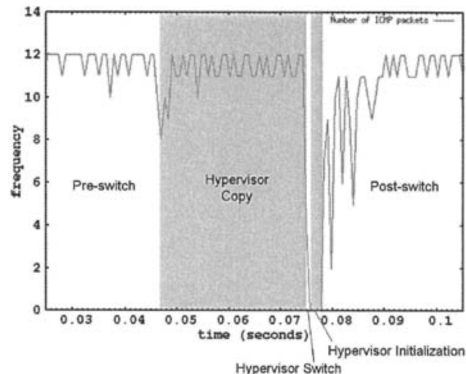

Fig. 4 Guest OS service

is a user application, the kernel may interrupt it and execute other processes and therefore, it does not affect the guest OS's service. However, service is affected in the second and third stages because the second stage shuts down CPU virtualization and it isn't re-enabled until after the third stage completes. These stages total to less than 3 milliseconds, and do not significantly affect the guest OS's service.

We also measure service outage through a macro-benchmark. This experiment is setup such that the machine running the hypervisor is ping-flooding another independent machine. The reason we need an independent machine is because time cannot necessarily be trusted in a virtual environment within the guest and it is safer to have an independent entity to measure time.

In figure 4, we can see that the hypervisor is consistently sending icmp requests before and after the switching takes place. However in the first stage of the hypervisor switch, the hypervisor copy, you can see that multiple hypercalls lead to a partial service slow-down as there is a dip in how often packets were able to be sent in the first four milliseconds. Afterwards, the hypervisor copy routine does not affect the performance of the guest OS significantly. However, once the hypervisor copy is finished, the guest OS must not be able to execute anything

while the hypervisor attempts to switch itself. Therefore in the second and third stages (hypervisor switch and hypervisor initialization), we see the expected outage of the guest. This outage corresponds to the three millisecond outage reported by our results in table 1.

8. Future Work

It is our goal to expand our current method to support multiple-processor systems since our hypervisor switching method currently only works with single-processor systems. Since the inter-processor interrupts (IPI) table is given exclusively to the guest, we cannot use it without violating the guest's resources. Therefore we would like to investigate what methods exist for synchronizing CPUs without the use of IPIs as well as methods to trap IPIs and virtualize a guest IPI table.

Furthermore, we would like to investigate how easily, and the benefits of porting our hypervisor switching method to other small VMMs.[2)4)8)9)] We would like to see our method work on other hypervisors and once this is achieved, we have the ambitious goal of creating a VMM that can be switched with another VMM of a different architecture. For example, we are deeply interested in bootstrapping a VMM, say Microvisor and then switch Microvisor with another one, say SecVisor.

Finally, we also intend to fully support the security features offered by Intel's Trusted Execution Technology. When we do so, we can rely on hardware to validate that the correct hypervisor is being loaded, as described in our applications section. More importantly, when a hypervisor switch or live-upgrade takes place, we can be confident that the correct hypervisor was loaded. The hypervisor's security policy can also be verified through TXT's attestation since the policy is embedded in the hypervisor itself.

9. Conclusion

We have successfully created a proof-of-concept live-upgrade method to switch between hypervisors of the same architecture. It does not require a reboot of the guest OS, and its interruption is not significant. Furthermore, we have studied its applications and detailed how it can be used to switch hypervisors, enhance hypervisor debugging, and enforce security policies.

References

1) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *Proceedings of SOSP 2003*, Bolton Landing, New York (2003).
2) Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M. and Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing, *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)* (2003).
3) Intel Corporation: Intel Trusted Execution Technology Preliminary Architecture Specification, `http://download.intel.com/technology/security/downloads/31516804.pdf` (2007).
4) Kaneda, K.: Tiny Virtual Machine Monitor, `http://www.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/`.
5) Kourai, K. and Chiba, S.: A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines, *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Washington, DC, USA, IEEE Computer Society, pp.245–255 (2007).
6) Lowell, D.E., Saito, Y. and Samberg, E.J.: Devirtualizable virtual machines enabling general, single-node, online maintenance, *SIGARCH Comput. Archit. News*, Vol.32, No.5, pp.211–223 (2004).
7) Parnas, D.L.: Software aging, *ICSE '94: Proceedings of the 16th international conference on Software engineering*, Los Alamitos, CA, USA, IEEE Computer Society Press, pp.279–287 (1994).
8) Ramachandran, M., Smith, N., Wood, M., Garg, S., Stanley, J., Eduri, E., Rappoport, R., Chobotaro, A., Klotz, C. and Janz, L.: New Client Virtualization Usage Models Using Intel Virtualization Technology, *Intel Technology Journal*, Vol.10, No.3, pp.205–216 (2006).
9) Seshadri, A., Luk, M., Qu, N. and Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, New York, NY, USA, ACM, pp.335–350 (2007).
10) The Trusted Computing Group: Trusted Computing Group Web Page, `https://www.trustedcomputinggroup.org/home`.
11) The Trusted Computing Group: *TPM Main Part 1 Design Principles Specification Version 1.2 Revision 103* (2007).