CMU-PT/RNQ/0015/2009

Trustworthy and Resilient Operations in a Network Environment

Deliverable D14

First specification of the enhanced components

Information and Communication Technologies Institute
Carnegie Mellon | PORTUGAL
AN INTERNATIONAL PARTNERSHIP

Executed by: **CISUC**

Direction/Department: **DEI/FCT/UC**

Date: **12-4-2012**

Version Number	Owner	Change Control	Date
0.1	FCTUC	Document creation	12-4-2012

Additional information:

Author/s:	Antnio Casimiro, Amir Soltani, Filipe Araujo, Raul Barbosa
Beneficiaries contributing on the deliverable:	FCUL, FCTUC
WP contributing to the deliverable:	WP3
Estimation of pm spent on the deliverable:	-
Nature of the deliverable:	Report
Total number of pages:	22

Contents

List of Figures

Executive Summary

This deliverable reports the work in Task 3.2, "Plug-in Components for Trustworthy Network Management". This task aims to investigate, enhance and prototype components providing security for the systems studied in the project. These components should work as (possibly optional) plug-ins providing improved resilience to critical but legacy parts of network OAM.

In this text we specify the kind of plug-in components that TRONE will use, what is their purpose and how will they operate, thus paving the way for an ulterior implementation, as foreseen in the description of work.

1 Introduction

In this deliverable we evaluate two different topics that will guide our research for the next year. The first topic concerns the security of the virtualization infrastructure: the hypervisor and the virtual machine manager (VMM). We review existing work, to evaluate the threats that exist to the different components of a virtualized machine: the guest and host operating systems, the hypervisor and the VMM. Along our text, we define what we think is the threat model considered by different researchers. Finally, we propose a defence-in-depth architecture, comprised of multiple operating systems and hypervisors that aims to make attacks much more difficult, as any attacker must penetrate multiple and diverse operating systems and hypervisors, before it reaches the lower layers of the architecture.

In the second part of our deliverable, we evaluate the implications of sharing replicas of the same service in the same physical machine. This may enable consensus algorithms to use fewer messages, thus becoming more efficient. On the other hand, these algorithms are more complex, because several processes may need to share the same physical devices that would otherwise serve isolated processes. Central to a number of consensus algorithms is the reliable broadcast algorithm. Therefore, in this deliverable, we focus on reliable broadcast algorithms. We present one alternative that is specially tailored for the cloud environment. Given the concentration of services and the resulting decrease in robustness, the most important question we need to answer concerns the reliability of this approach. Our results are surprising: we can benefit from concentrating as many replicas as possible in a single physical machine, leaving all the other replicas alone in independent physical machines.

2 Threats and Security Models for Virtualized Environments

2.1 Introduction

Today there are several tools to virtualize a host. In cloud computing, cloud providers prefer to use virtualization in offering their services, because it is an efficient way of using resources. Since virtualization is getting widespread and currently millions of users everyday are being served by this technology, the need for protecting users against attackers is increasingly obvious.

One of the techniques in security is defence-in-depth. Using this technique, we are able to locate several defensive layers and mechanisms against adversaries that wish to attack the system. The task of the adversary becomes very difficult since it must then defeat layers

one by one. To apply the idea of defence-in-depth in virtualization environments, we need to have several consecutive virtual machines as layers. Unfortunately, current hardware does not support this kind of layering. Therefore, we need to emulate it by software.

Recursive virtualization is an approach that lets each virtual machine recursively create another layer of virtual machine itself. So, we can implement the defence-in-depth technique using recursive virtualization. However, there has been always a big obstacle to using recursive virtualization, which is huge overhead. To deal with this problem, [11] introduces an efficient way of implementing recursive virtualization by managing recursive VMs in the root hypervisor rather than repeating the support for nested virtual machines in every layer. This approach has several advantages. Foremost, all virtualization instructions can be already emulated at the root and need not to be propagated trough the hierarchy. Other virtualization events can be directly forwarded to the right upper layer, since the root hypervisor knows all VMs in the system. Furthermore, the root hypervisor can aggressively cache intermediate values to reduce the overhead. Finally, upper layers can be simpler as they do not need to be involved or in most cases even aware of recursive VMs. In the following we will describe how we implement this idea.The approach makes it feasible to implement this type of nested virtualization.

In this report, first we go over works in the literature concerning security in virtualization environments. Afterwards, we propose our threat model. Then, we present an architecture combining the ideas of defence-in-depth and recursive virtualization to make virtualization environments much more resilient to attacks. We conclude this topic with a discussion of our future work.

2.2 Related Work

The theoretical issues of recursive virtualization were first addressed by Popek and Goldberg [16]. However, given that high overhead has always been the side effect of recursive virtualization, to the best of our knowledge our proposal is novel in using recursive virtualization to enhance security in these systems. There is already a considerably large number of papers dealing with security in virtualization and hypervisors, but from different perspectives as explained below. Generally speaking, the main difference between these works in the literature and our work, which is using multi layering, is that those works rely on only a single layer, i.e. hypervisor, for providing security for virtualization environments, while our work relies on multi layers for fulfilling security in such environments.

The first viewpoint according to which there are several works in the literature is shrinking the code of hypervisors. For instance, [17] presents a tiny hypervisor named SecVisor

mostly to prevent attackers from injecting their arbitrary codes into OSes kernel. They use today's CPU capabilities to perform this goal. Briefly, the idea behind this work is that the CPU should run just any kernel-mode code that has been verified by SecVisor. Also, nobody can modify the kernel-mode codes, but SecVisor. On top of SecVisor, a commodity OS such as Linux can sit. The threat Model considered in this paper is that an attacker who controls everything in the computer system but the CPU, the memory controller, and memory. This trusted computing base (TCB) is minimal for the architecture which is used by most computing devices today: the von Neumann architecture (also called a stored-program computer). Examples of attacks the attacker can perform are: arbitrarily modify all memory contents, inject malicious code into the system firmware (also called the BIOS on x86 systems), perform malicious DMA writes to memory using peripherals, and insert malicious peripherals into the system. Also, the attacker might be aware of zero-day vulnerabilities in the kernel and application software on the system. The attacker can attempt to use these vulnerabilities to locally or remotely exploit the system. For the x86 architecture, they assume that the System Management Mode (SMM) handler is not malicious. [18] discusses a small hypervisor (micro hypervisor) that cannot be exploited easily by an adversary. This is because the small piece of code can be easily verified and any bugs can be removed much more easily than in a large hypervisor. The authors move most of the functions, which are not necessary, to the user level. This paper cares more for the security of the hypervisor than for the security of virtual machines. So, briefly, this paper focus on reducing the lines of the code of the hypervisor (system level) by moving most of the unnecessary parts of that such as functions, and drivers into the user level part. So, the system level (TCB) becomes very small and hard to be exploited by an attacker.

The second viewpoint is isolation. It means that we try to isolate the components of a hypervisor such that if one component is compromised, then it will be difficult that the other components get compromised. In this category, we can name [5] that presents a virtualization platform named Xoar based on Xen that has the same huge functionality, but they strengthened the VM control using isolation of the components in VM control. They criticize the designers who start writing a hypervisor from scratch since these new ones do not have the huge features of well established hypervisors such as Xen, Vmware etc. The threat model in this work is that the attacker in their model is a guest VM aiming to violate the security of another guest with whom it is sharing the underlying platform. This includes violating the data integrity or confidentiality of the target guest or exploiting the code of the guest. While we assume that the hypervisor of the virtualization platform is trusted, we also assume that the code instantiating the functionality of the control VM will contain bugs that are a potential source of compromise. Note that in the case of a privileged monolithic control

VM, a successful attack on any one of its many interfaces can lead to innumerable exploits against guest VMs. Rather than exploring techniques that might allow for the construction of a bugfree platform, our more pragmatic goal is to provide an architecture that isolates functional components in space and time so that an exploit of one component is not sufficient to mount a successful attack against another guest or the underlying platform. Additionally, [3] is giving an architecture in which they offer services above VMM (hypervisor) and under guest operating system. In other words, these services are in a layer on top of which there is another layer (VM) including the guest operating system and guest applications. This is because if the operating system is compromised, then the services are still safe. This technique is useful for some sensitive techniques such as Intrusion Detection System (IDS) or logging. One interesting trick in this paper is that a VMM can forward any suspicious events (such as a packet) to a certain VM that is a clone of the real system, and let the VM process the event. If the VM is still healthy, then VMM can infer that that event is not from an adversary. [4] presents overshadow, a layer running on a commodity operating systems, that cryptographically protects and isolates the application's data and codes from the OS on which the application is running. In other words, if an OS is compromised, it cannot corrupt the data and code of the application (running on the OS) that is protected by this layer. This system does this using shadow memory page and cryptographic techniques. The threat model considered in this paper is that a compromised operating system should not access application's data, code and registers. It can acces the data in a cryptographic way, but if it modifies it, then the application notices that. This work does not guarantee the availability, when the OS is compromised.

Another perspective is using another piece of hardware co-operating with the host's hardware to increase security. For example, [10] is presenting a kernel-modifying rootkit detector. They use a coprocessor connected to a monitored host's PCI that runs independently from the monitored host. So, if the kernel of monitored host is compromised, then this method can still identify a rootkit. The previous methods mostly were software run on the kernel of monitored host, and so, if the host is compromised, then the detector might fail to detect it. The brief algorithm is that the coprocessor's program frequently checks the RAM of monitored host where consists of critical part of Kernel, and then makes a hash from it, and compares it to a trusted one. So, if any changes are found, then it means that the rootkit has compromised the kernel of the monitored host.

Furthermore, code verification is another way of increasing security in virtualization environments. As an example, [12] is giving a microkernel named SeL4 that is a member of the L4, which is verified totally by the authors. In other words, this paper is presenting a method of verification for C code of seL4.

Moreover, some works try to periodically check the binary code that is running to detect any code injections into the original code. For instance, [15] is presenting a hypervisor-based system that detects and identifies any binary code that is running, its advantage with respect to the previous works is that it relies on MMU processor to identify the executing processors instead of OS. So, it cannot be fooled by rootkits. Also, Secvisor [17] operates in this way.

In addition, many systems use some operating systems and security techniques to prevent attackers from taking over the virtualized systems. For example, [21] uses two operating system techniques, non-bypassable memory lockdown and restricted pointer indexing, to secure hypervisor. Then, the authors evaluate their proposed system and prove it. The threat Model assumed in this work is that they assume an adversary model where attackers are able to exploit software vulnerabilities in an attempt to overwrite any location in memory. However, to successfully launch an attack, attackers will have to either inject and execute their own code or leverage and misuse existing code. Note this represents a powerful adversary model as attackers can attempt to inject code, modify existing code, and exercise more sophisticated attacks such as return-oriented ones. [9] is presenting a system with multi VM and a VMM as a means for providing security. It serves some security and cryptography techniques to protect users and applications.

There are many other works that are relevent to security in virtualization environments. [19] mostly discusses eliminating the role of the hypervisor. In other words, it wants to remove the contact between VMs and host resources. Briefly, using four techniques, pre-allocation of CPU and memory, use of virtualized I/O devices, minor modifications of guest OSes and avoiding indirection, tries to minimize the role of hypervisor, which can be vulnerable to attacks, and in turn contact between VMs and host. In other words, relying on new technologies and new hardware supporting virtualization, the software layer (component) of virtualization such as hypervisor can be minimized and even removed. The side effect of this job is that resource allocation should be static, and we no longer have the dynamic allocation performed by current hypervisors. The Threat Model considered in this work is that the authors aim to protect against attacks on the hypervisor by the guest VMs. A malicious VM could cause a VM exit to occur in such a manner as to inject malicious code or trigger a bug in the hypervisor. Injecting code or triggering a bug could potentially be used to violate confidentiality or integrity of other VMs or even crash or slow down the hypervisor, causing a denial-of-service attack violating availability. [8] is interesting and well explaining, is introducing an IDS formed by collaboration of a VMM (hypervisor) sit on a monitored host and a network-based IDS. This idea has two properties of high visibility due to full control of VMM over the monitored host and high resilience due to network-based IDS. For the Threat Model, authors assume that the code running inside a monitored host may

be totally malicious. They believe this model is quite timely as attackers are increasingly masking their activities and subverting intrusion detection systems through tampering with the OS kernel, shared libraries, and applications that are used to report and audit system state.

2.3 Threat Model

In this section, we describe the considered threat model. We assume that a user virtual machine can be malicious, and may attempt to compromise other VMs and the whole system. Therefore, our ultimate goal is to increase the protection of the main hypervisor, other defensive VMs and other user VMs against attacks originating from such a malicious user VM. Example attacks include injecting malicious code or triggering a bug in the hypervisor by a malicious VM. An attacker may try to reach operating systems in intermediate layers and install its malicious code completely inside virtual machines. The malicious code could eventually cause bad effects on integrity and confidentiality of operating systems (VMs) in our multi-layer architecture, as well as on the availability of them by some sort of DOS attacks. In this work we do not consider hardware security. In fact, we assume that malicious code cannot tamper with hardware, platform BIOS and other important hardware parts.

2.4 A Secure Architecture for a Recursive Virtualization Environment

At this stage, we only focus on proposing an architecture for virtualization environments, as depicted in Figure 1. Before going into depth, let us remark that in this report, we distinguish between hypervisor and virtual machine monitor (VMM) terms. According to our terminology, hypervisor is a system-level entity of small size that operates only critical tasks required for a virtualized system. On the other hand, VMM is a user-level component that accomplishes complementary functionalities, which were removed by shrinking the size of a commodity hypervisor. Furthermore, we do not have any specific hardware requirements since our proposed architecture can be materialized using current commodity hardware.

As explained earlier, we would like to choose a tiny hypervisor for the lowest (and main) layer of our architecture, and then customize it such that it contains the capability introduced in [11] for a low overhead recursive virtualized system. We believe that SecVisor is a good candidate for deploying the root-layer of our architecture, given its characteristics described earlier. By using SecVisor or any similar hypervisor as our root layer, we can detect any changes in the kernels of the intermediate layers, and thus increase the security of our virtualization system.

On top of this root layer we are able to have several virtual machines. In regular non-recursive virtualizations, we can create our guest operating systems just above the root hypervisor. What we propose is to add several defensive mechanisms on top of the root hypervisor to strengthen the security of the whole system, making it much harder for attackers to take control of the system. On top of the root layer, defensive virtual machines are created. The number of these defensive layers depends on how much security is required for protecting our virtualization environment. In cloud computing, a cloud provider may decide on the number of intermediate layers based on the security level that a user demands.

For the entire architecture, we would like to take advantage of diversity to increase the security level [14]. For example, intermediate layers can be various operating systems along with different small and verified hypervisors such as SecVisor and NOVA [18].

Since we use small hypervisors in our architecture, it means that we cannot expect them to provide all the functionalities, such as Administrative Tools, Live Migration, Device Drivers and Device Emulation, that are available with commodity hypervisors. However, since we nevertheless must provide all these functionalities to users, we need to use a user-level virtual machine monitor (VMM). To this end, using a VMM that has additional resilience against attacks seems a good idea. One possibility is to use a VMM with completely isolated components such that, in the case a VMM's component is compromised by a malicious user, other components will not be easily compromised [5]. Let us remark that this VMM is only needed in the last defensive layer just before the user VM, and other intermediate layers do not need such a VMM.

To increase the security level even more, we can equip intermediate layers with different host-based IDSes. We can deploy network-based IDSes or hybrid IDSes [8] for protecting our system from network traffic. Finally, the top layer is the user virtual machine, which may be malicious.

2.5 Discussion

In this work, we used the idea of defence-in-depth to elevate the security degree of our virtualized environment. For this purpose, we propose to take advantage of recursive virtualization with reasonable overhead, as introduced in [11]. Then we add additional security layers on top, which altogether make it difficult for an attacker to compromise the root hypervisor and other user and defensive virtual machines. As a future work, we are going to implement and materialize this architecture, evaluate the security of this virtualization system, and compare it to other virtualization environments in the literature.

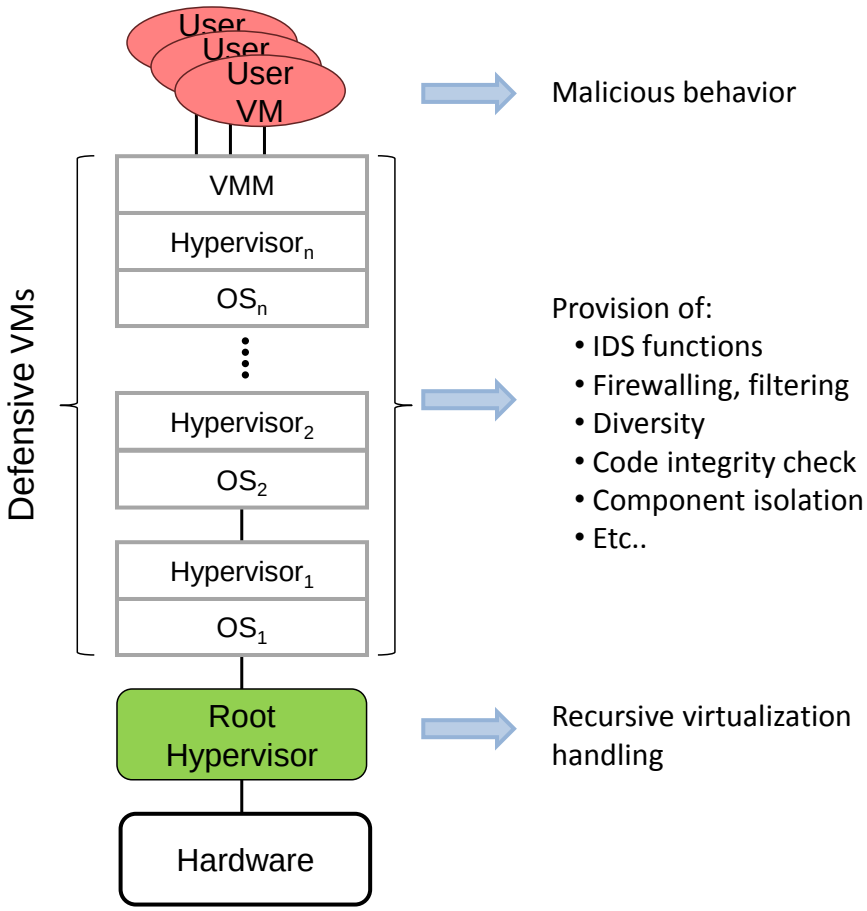

Figure 1: An Architecture for Recursive Virtualization

3 Consensus for Virtual Machines

We consider a set of n processes $\{p_1, p_2, \ldots, p_n\}$. A process that follows the algorithm is correct, otherwise it is faulty. We assume that at most $\lfloor (n-1)/2 \rfloor$ processes may be faulty. These faulty processes can deviate arbitrarily from the protocol, i.e., we assume Byzantine faults [13]. Processes use messages to communicate. Channels are fair. We also assume that the system is asynchronous, although some of the algorithms depend on failure detectors [2], which cannot really be implemented in such environment. In fact, they may need quasi-synchronous networks [20]. This is the case of the muteness failure detectors [7].

In the paper of Correia *et al.* [6], the Reliable Broadcast Algorithm (RBA) is a fundamental component of the Consensus Algorithms that can tolerate $\lfloor (n-1)/2 \rfloor$ process failures out of n. A distinctive feature of the RBA is the Trusted Platform Module (TPM), which can sign messages for the processes. In this deliverable we assume somewhat different conditions: we consider that we have a number of physical machines that can handle more than a single process. Therefore, we want to take advantage of the fact that many processes indeed run on

the same physical machine to cut on the number of messages and improve the fault-tolerance of the system. We first propose a few reliable broadcast algorithms, to support a conjecture about the consensus algorithm.

3.1 Reliable Broadcast Algorithms

We propose three mechanisms for reliable broadcast and mention an additional one that uses a more traditional approach. All these mechanisms use underlying facilities to ensure some basic mechanisms, like signing a message in the simplest case.

Basic RBA In this case, processes are oblivious to the machine where they run and to the fact that several processes may share the same physical machine. A problem with this approach is that each process needs its own sequence of signed message identifiers.

Proxy-based RBA To conserve network bandwidth and speed up the broadcasting process, each physical machine may have a special *proxy* process, which serves to run the RBA on behalf of all the processes in the machine. On the downside, if the proxy process crashes or behaves in a byzantine way, all the other processes that share the same physical machine become deprived of the service.

OS-based multicast One simple alternative is to run RBA on top of a multicast address. This is very likely to work in cloud-based facilities where all the servers reside in the same LAN.

TPM-based broadcast We can minimize the surface of attack, by passing many of the broadcast facilities to dedicated hardware. In this alternative, we consider the minimum set of features that such hardware should posses to run broadcast.

One possibility is to let the processes send messages to multicast addresses assigned to all the processes of the group, including those that run in the same physical machine. E.g., p_i could send a single message to some multicast address, where all the processes of the same machine join, instead of sending one separate message to each one of those processes. This involves protocols like IGMP [1], or some tunnel-based approach [1]. The shortcoming of this method is the difficulty to precisely limit the surface of attack, as the multicast operation, even inside a single machine is somewhat complex. In other words, although such mechanism could save some communication resources, thus speeding up communication, we could not really tell anything useful about how well would it resist to attacks.

[1] We can find this sort of idea in proposals like the Automatic Multicast Tunnel, `http://tools.ietf.org/html/draft-ietf-mboned-auto-multicast-14`, which tries to reach hosts that cannot participate in multicast groups.

3.2 The TPM-Based broadcast

Here, we propose a mechanism that can take advantage of virtualization, the *Trusted Processing Module (TPM)-based broadcast*, which might be implemented using a USB dongle or a dedicated processor for example. The idea is as follows: each physical machine must have a TPM accessible by all the processes, no matter what virtual machine they belong to. This TPM has a memory for messages, the "message bus". This message bus signs and stores messages not only from the local processes on the same physical machine, but also coming from other processes on different physical machines. As this message bus works as a topic for publish-subscribe, any process in the physical machine can receive all the messages that enter the bus. The TPM's signature can prove that a message actually reached the bus. We consider the process of distributing the public keys of the TPMs to be out of scope of this deliverable.

For the overall mechanism to work, we make a number of assumptions:

- Each process has the public key of each TPM of the group.
- The local TPM can authenticate each process of its own machine.
- The local TPM can assign a globally unique identifier to each message, id.

We illustrate the message bus mechanism in Figure 2. The overall idea is the following: process A on machine a sends a unicast message to process B on machine b and does the same to each other machine, picking one process by machine. The TPM of machine a signs this message together with a growing integer identifier to ensure that process A sends the same message to every destination (this is the same approach as in [6]). The receiving process B writes the message to the message bus, which signs it, and starts a broadcast of the message itself. This message will now have the machine b TPM's signature alongside with a's signature. B sends an acknowledgment to A, showing the signed message. This lets A know that machine b already got the message (because it was signed). However, process failures may complicate the algorithm a little further. Processes should periodically check if all the machines confirmed reception of the message. This copes with the case where process B does not reply to process A. Moreover, process A, or some other process forwarding a message may fail, leaving the co-located peers unable to determine to which processes did the message reach. To solve this problem, upon receiving each broadcast announcement or confirmation, process A associates the message to the sending machine. In this way, any process in machine a may determine which other machines did the message reach. The broadcast process is not finished before each machine receives a signed reply from each peer

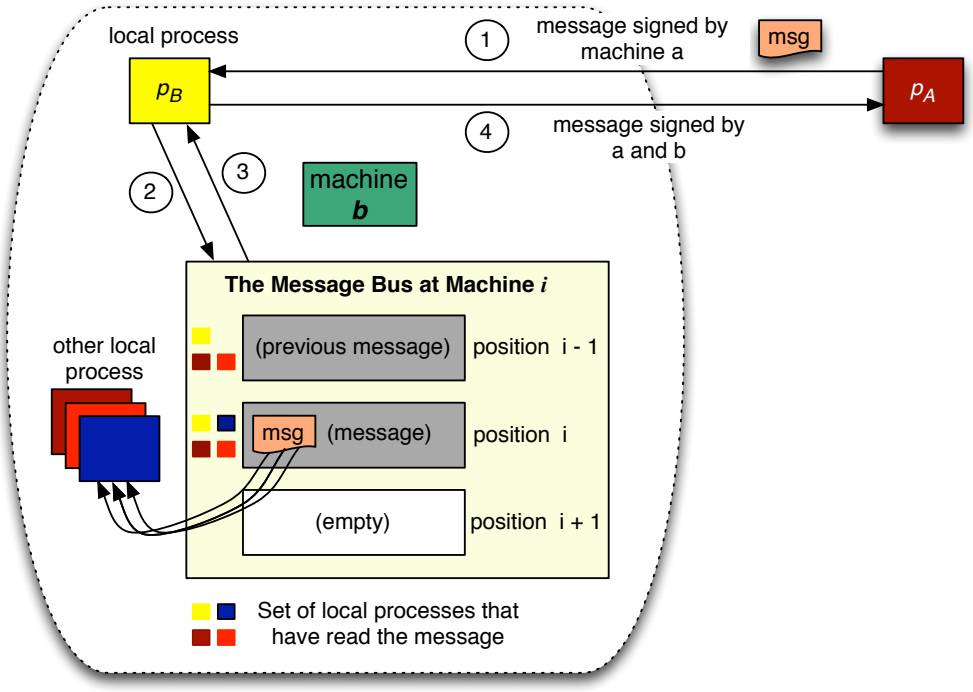

Figure 2: TPM implementation of the message bus

machine. By writing these replies on the message bus, processes will know which replies are missing and by including a signature from the originator of the message and from the machine that sent the reply, they know that the messages cannot be forged.

In formal terms, the algorithm uses the functions of Table 1. To start the broadcasting mechanism, a process p_i must call the algorithm as follows: $broadcast(m, p_i, p_i)$.

Algorithm 1 Algorithm $broadcast(m, p_i, p_j)$ from p_j on machine m_j to p_i on machine m_i

$write(m, m_i)$
if $p_i \neq p_j$ **then**
  $write(m, m_j)$
  $ack(m, p_i, p_j)$
end if
for all m_k from the group of other machines not associated to id on the message bus **do**
  Let p_k be a random unsuspected process on machine m_k
  $broadcast(m, p_k, p_i)$
end for

Algorithm 3 serves to restart the process of sending the message to some machine that has not confirmed reception of the message yet. This could mean that the process that locally started the broadcast failed, or otherwise it should have taken care of sending the message to other machines. In practice timeouts should be set with different expiration timers, to

Table 1: TPM broadcast services

Function	Meaning
$write(m, m_j)$	If the message does not exist on the message bus, writes it. In this case, the TPM signs the message and every process in the machine of the message bus should deliver the message. Furthermore, it associates machine m_j with message m. This serves as an acknowledgment that the message reached machine m_j.
$broadcast(m, p_i, p_j)$	Broadcast message m from p_j on machine m_j to p_i on machine m_i.
$ack(m, p_i, p_j)$	Acknowledgment from process p_i, machine m_i, to p_j on machine m_j.

Algorithm 2 Algorithm $ack(m, p_i, p_j)$

$write(m, m_i)$

avoid different processes on the same machine from repeating the same operations.

Algorithm 3 Upon timeout on process p_i, machine m_i, from lack of confirmation from machine m_k

Let p_k be a random unsuspected process on machine m_k
$broadcast(m, p_k, p_i)$

We now demonstrate the correctness of our algorithm, based on the Validity, Agreement and Integrity properties.

Validity: If a correct process broadcasts a message m, then some correct process eventually delivers m. A correct process always writes the message on the message bus before starting the broadcast. This enforces local delivery, *before* the broadcast.

Agreement: If a correct process delivers a message m, then all correct processes eventually deliver m. Assume that p_i and p_j are correct and that p_i delivers the message, while p_j does not, having both finished execution of the reliable broadcast algorithm. The message must not have reached the message bus of p_j. Hence, p_j must not run in the same machine as p_i. It must also be the case that p_i's message bus has no confirmation from p_j's machine. This could only occur if p_j did not execute Algorithm 1, which implies that neither p_i, nor any other process in the machine of p_i have sent a broadcast message to p_j's machine. From Algorithm 1 they must do it, which is a contradiction.

Integrity: For any identifier id and sender p, every correct process q delivers at most one message m with identifier id from sender p, and if p is correct then m was previously broadcast by p. Each message has a unique identifier and cannot make it twice to the

same message bus. No process can generate a message on behalf of p because the TPM authenticates the processes and will not sign a message from any other process on the same machine.

3.3 Communication Costs

We use two criteria to compare the previous broadcast communication alternatives: communication cost and resilience to (arbitrary) process failures. We discard the messages exchanged inside the same physical machine and consider only message costs among different physical machines, both in terms of rounds and in terms of absolute number of messages. We assume that processes are correct.

RBA For n processes in m machines ($n \geq m$), the total cost of the broadcast depends on the distribution of processes by the machines. Assuming a uniform distribution, we have approximately n/m processes per machine. The cost for this sort of distribution is $n(n - n/m)$. This cost drops to the minimum of $2(n - 1)$, if we have one process in one machine and all the remaining processes in a second machine.

Proxy For m machines, this involves m proxies, thus a total proportional to $m(m - 1)$ messages.

OS-based multicast We cannot asses this value, because this strongly depends on a specific protocol. However, we could argue that given the need to confirm messages, this cost should be around $n(m - 1)$, as each one of the n processes sends one message to each one of the other $m - 1$ machines.

TPM-based broadcast The cost is similar to the proxy case: $m(m - 1)$.

3.4 The Consensus

We assume the settings of the paper of Correia *et al.* [6]. Given a reliable broadcast algorithm (RBA) with the same properties of their paper, which we proved for our own case in Section 3.2, we simply reuse the solutions these authors proposed for the consensus algorithms. As Correia *et al.* [6] show, their algorithms can resist to $(n - 1)/2$ processes failures (including byzantine failures). We call this limitation, condition S, of "standard" (i.e, at most $(n - 1)/2$ failures). However, the fact that each machine may run multiple processes

Table 2: Comparison of the different consensus mechanisms

Algorithm	Messages	Failure Conditions Tolerated
RBA	$n(n-1) \rightarrow O(n^2)$	S, M
Proxy	$m(m-1) \rightarrow O(m^2)$	S, M, P
OS multicast	$n(m-1) \rightarrow O(nm)$	S, M
TPM broadcast	$m(m-1) \rightarrow O(m^2)$	S, M

introduces an additional point of failure, because faulty machines can bring down many processes at once. Hence, from all possible sets of machines, only those that do not sum more than $(n-1)/2$ processes may fail. We call this the M condition, of "machines". Additionally, some algorithms raise other restrictions. We discuss them here:

RBA The reliable broadcast algorithm itself may resist to any number of failed nodes [6]. Therefore, condition S and M apply.

Proxy The proxy-based algorithm is much less tolerant to faults. Indeed, a single faulty proxy prevents participation of all the processes of the affected virtual machine. Therefore, it has the additional restriction that the number of proxy crashes must not affect machines that have more than $(n-1)/2$ processes. We call this the P condition, of "proxy".

OS-based multicast We may assume the conditions S and M.

TPM-based broadcast Again, we have the conditions S and M.

In Table 2 we summarize the evaluation for the different broadcast algorithms. Since the consensus algorithms of Correia *et al.* [6] use a constant number of rounds of reliable broadcast, we consider the costs of the RBA.

4 Conclusion

In this deliverable we presented some challenges that clouds face as they stand on virtualization. For instance, replicas of the same application sharing a physical machine may raise a number of important problems. Algorithms that serve to create consensus among processes or any other algorithms that require special hardware devices, such as a Trusted Platform Modules (TPM) simply cannot run as before. We need to virtualize the access to the TPM. However, it becomes apparent that as we do so, we can improve the existing algorithms, by leveraging on the fact that many processes are co-located and thus can save

on communication costs. To achieve this goal we proposed the TPM-based multicast, where processes rely on a message bus provided by a TPM. With such a consensus algorithm, the number of messages may grow quadratically with the number of machines (instead of processes) without causing any negative impact to the reliability.

The future of this work in the TRONE project will likely consist of picking a particular application running in multiple replicas and implement a consensus algorithm that can make the views of the replicas consistent.

References

[1] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet group management protocol, version 3, 2002.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[3] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HotOS*, pages 133–138, 2001.

[4] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, pages 2–13, 2008.

[5] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *SOSP*, pages 189–202, 2011.

[6] Miguel Correia, Giuliana S. Veronese, and Lau Cheuk Lung. Asynchronous byzantine consensus with 2f+1 processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 475–480, New York, NY, USA, 2010. ACM.

[7] Assia Doudou, Benot Garbinato, and Rachid Guerraoui. *Tolerating Arbitrary Failures With State Machine Replication*, chapter 2, pages 27–56. Wiley-Interscience, 2009.

[8] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[9] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP*, pages 193–206, 2003.

[10] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, 2004.

[11] B. Kauer, P. Verissimo, and A. Bessani. Recursive virtual machines for advanced security mechanisms. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, DSNW '11, pages 117–122, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *SOSP*, pages 207–220, 2009.

[13] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[14] B. Littlewood and L. Strigini. Redundancy and diversity in security. In *ESORICS*, pages 423–438, 2004.

[15] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, pages 243–258, 2008.

[16] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. In *SOSP*, page 121, 1973.

[17] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP*, pages 335–350, 2007.

[18] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys*, pages 209–222, 2010.

[19] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *ACM Conference on Computer and Communications Security*, pages 401–412, 2011.

[20] Paulo Verssimo and Carlos Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, Winter 1995.

[21] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy*, pages 380–395, 2010.