Blind Hypervision To Protect Virtual Machine Privacy Against Hypervisor Escape Vulnerabilities

P. Dubrulle, R. Sirdey, P. Doré, M. Aichouch
CEA, LIST
Point Courier 172, FR-91191 Gif-sur-Yvette Cedex, France
firstname.lastname@cea.fr

E. Ohayon*
Krono Safe
86 rue de Paris, FR-91400 Orsay, France
emmanuel.ohayon@krono-safe.com

*At the time this work was originally carried out, E. Ohayon was working at CEA, LIST; he has joined Krono Safe in the mean time, in June 2014.

Abstract—Hypervision is being widely implemented in an effort to control costs and to simplify management through consolidation of servers. It has been recently unraveled that well over a third of virtualization vulnerabilities reside in the hypervisor, mostly due to hypervisor escape. The exploitation of these vulnerabilities allows an attacker, among other things, to access and/or modify data of other Virtual Machines (VMs) by escaping from its VM and executing malicious code in the hypervisor. This paper introduces the general idea of blind hypervision, a hardware/software co-design to prevent such attackers to access private elements of other VMs. Blind hypervision limits the rights of the hypervisor regarding memory access, so that a malicious agent executing with hypervisor rights cannot access the data of the VMs.

I. INTRODUCTION

Hypervision is an old concept that was used in the early days of computer science back in the late 60s. There was a renewed interest in virtualization and hypervisors during the end of the last decade. This was done in order to expand hardware capabilities, and to benefit from the possibility to have a safe and reliable cohabitation of complex OS-dependent applications. In [11], the authors analyze the benefits and risks of the hypervision approach, and enumerate some of the vulnerabilities of this architecture.

Recently, it has been observed that well over a third of virtualization vulnerabilities reside in the hypervisor, mostly due to hypervisor escape [1]. Hypervisor escape grants attackers administrative-level rights in the hypervisor itself. The attacker is then able to run random code and access private data. This occurs when an attacker escapes from a virtual machine to attack other virtual machines or the hypervisor itself.

In hypervision for mixed-critical environments, several third party VMs can run on the same hardware. The hypervisor escape attacks can lead to one VM trying to steal binary contents from others, which leads to intellectual property issues.

In this paper, we introduce the general concepts of a hardware/software co-design to enforce confidentiality between the VMs and the hypervisor, by protecting them from software threats where the attacker tries to breach the information security of the VMs. An attacker reaching the hypervisor through privilege escalation, or already residing in the hypervisor, will be unable to access or modify data in other VMs, as the hypervisor itself is unauthorized to do so. This segmentation requires some specific hardware and modifications to the hypervisor.

This blind hypervision solution is focused on privacy, and does not address denial of service attacks, where a malicious code manages to get to hypervisor space and prevents other VMs to execute. A VM’s security breaches will not be reduced by our approach, and we do not intend either to protect the hardware itself against side channel attacks. We rather intend to make sure that only a physical attack could jeopardize the information security, and that virtualizing a machine does not increase its attack surface regarding privacy.

II. VIRTUALIZATION ARCHITECTURE

Blind hypervision is based on a network architecture composed of one or more blind hypervision hosts and one virtualization master (respectively named hosts and master hereafter). The hosts are in charge of executing the VMs and the master of the global system state. The network connecting these machines is not considered secured in this architecture.

The master supervises VM migrations from one host to another. It is also responsible for the deployment of a new VM in the system by allocating it to a host. It makes sure that VM transfers over the network are secured, by acting as an authentication authority. The master can be deployed on a standard server machine and does not require additional hardware.

The hosts execute a hypervisor, and receive encrypted VM images to execute from the master. The hosts also receive migration orders from the master. In order to enforce the privacy of the VMs it executes, a host must have some specific hardware components described in section III.

A. Introducing a new image

The master introduces new images into the system, which requires to send them ciphered over the unsecured network, following the protocol illustrated in Figure 1. To do so, it pairs each new VM image \( v \) with a unique symmetric key \( k_v \), and also has a list of one public key \( k_h \) per host \( h \). The new image is ciphered into \( k_{h,v} \).

When a host \( h \) is elected to run a VM \( v \), the master ciphers \( k_v \) into \( k_{h,v} \). It then sends \( k_{h,v} \) to \( h \) who must then prepare to receive the ciphered image \( k_{h,v} \).
Master Host (h)

\[ k_h \rightarrow k_v \]

\[ v \rightarrow k_v \]

\[ k_v \rightarrow v \]

Host (h)

Fig. 1: Protocol to introduce a new VM in the system; the VM image \( v \) is ciphered using a unique symmetric key \( k_v \), the key is ciphered using the public key \( k'_h \) of target host \( h \), the master sends the ciphered symmetric key to \( h \) who deciphers it using its private key \( k'_h \), and finally the ciphered image.

When a host receives an image and its key, a specific hardware component is in charge of loading it to memory. This component, described in detail below in section III, is the only one capable to read and decipher an image’s key. Thus it is the only one capable to read the image’s contents.

This protocol is the minimal apparatus needed to securely exchange images over the network between hosts and master. It could be extended to reinforce the authentication between the hosts and the master, to prevent for example a malicious machine to send fake images to hosts. Another extension could use a complete key exchange protocol.

B. Image migration

Migrating an image \( v \) from host \( h_i \) to host \( h_j \) follows a similar protocol, illustrated in Figure 2. First the master sends \( k'_j(k_v) \) to the target host \( h_j \), who must prepare to receive the image. It then sends a migration order to \( h_i \), who is in charge to send \( k_v(v) \) to \( h_j \). The receiving host then proceeds as when receiving an image from the master.

\[ k_j \rightarrow k_v \]

\[ v \rightarrow k_v \]

Master

\[ \text{migrate}(t) \]

Host (t)

\[ k'_j \rightarrow k_v \]

\[ v \rightarrow k_v \]

Host (s)

Master

\[ \text{migrate}(t) \]

Host (t)

\[ k_v \rightarrow v \]

Host (s)

\[ v \rightarrow k_v \]

Fig. 2: Protocol to migrate a VM in the system; the VM image \( v \) is migrated from source host \( s \) to target host \( t \), the protocol and notations are similar to the one described in Figure 1, the only difference is that \( s \) sends the ciphered image to \( t \), to preserve the state of the image.

III. BLIND HYPERVISION HOSTS

The hosts must have some hardware extensions in order to enforce the privacy of their VMs (for advanced memory management and secure load/unload operations for VM images). It also requires some software modifications to take these extensions into account.

In brief, the hosts have an isolated and trusted component called the loader, in charge of ciphering/deciphering the VM images and loading/unloading them in memory. The hosts also have an extended memory management unit to enforce isolation of all the software components running on the host (hypervisor and VMs), and some specific execution modes to protect these elements from illegal accesses. The memory isolation is done by allocating fixed size partitions, each uniquely identified and exclusively dedicated to one type of software component (hypervisor or VM).

A. Execution modes

The processor of the host machines must implement four execution modes: initialization, hypervisor, VM privileged and VM protected. The latter three are similar to what is commonly found in architectures providing hardware support for virtualization. In particular, they have a decreasing priority regarding interrupts (e.g. execution in VM privileged mode can be interrupted and switched to hypervisor mode either by an external interrupt source or a specific instruction).

1) Processor reset: If the processor has an instruction for a soft reset, it should be unavailable in VM protected mode, and restart the VM in VM privileged mode. If such an instruction is executed in hypervisor mode, or when the reset pin of the processor is asserted, the following actions must take place in order to guarantee that no lingering data in the memory can be accessed after the reset:

1) reset of all the RAM, invalidation of all the caches;
2) reset the processor state (general purpose registers, state registers, etc.);
3) reset the loader;
4) deactivation of the memory management unit;
5) reset of all IO devices (especially buffers);
6) switch to the initialization mode and load PC with the address of the reset vector.

2) Initialization: this is the initial mode of the processor. It allows full access to the memory and to its memory management unit. The loader is not accessible, so no VM can be loaded or executed. Thus, as the memory is always cleared when entering initialization, no private data can be accessed, even though full access to memory is granted. It is possible to initialize the host and its IO devices, and to configure the memory management unit in order to create the isolated partitions, with a mapping of the partition ids to their address space. A specific instruction, illegal in all other modes, allows to leave the initialization and switch to the hypervisor mode (see Figure 3). The switching enables the loader and is one way only (the only way back to initialization is the reset procedure, described above). The switching consists in the following sequence:

1) flush and invalidate all cache memories;
2) reset the processor state (general purpose registers, state registers, etc.);
3) activate the memory management unit and the loader;

1395
4) set the processor mode to hypervisor, with the PC positioned to a fixed address determined by the architecture (for example the address following the instruction).

3) Hypervisor: this is the execution mode of the hypervisor. Its particularity regarding the usual implementations of this mode is that the access to the memory management unit is denied. As the hypervisor has only access to its own partition, it is unable to access the memory areas reserved to the VMs. The only operations on the memory are performed by the trusted loader, which is now enabled and accessible (see subsection III-D for the operations). Starting the execution of a VM is done by a macro-instruction taking as parameter the identifier of a VM’s partition (see Figure 3), which performs the following actions:

1) flush and invalidate all cache memories;
2) reset the processor state (general purpose registers, state registers, etc.);
3) switch context of the memory management unit, as to only allow access to the identified partition;
4) switch to VM privileged mode;
5) restore processor state from a specific location in the partition (entering either VM privileged or protected mode, depending on the restored processor state - see subsection III-E for details on this step).

4) VM privileged: this mode is similar to supervisor modes that can be found on processors with hypervisor support, which is usually used to run the VM’s kernel. It is for example possible in this mode to execute privileged instructions to mask interrupts or modify the VM’s address space (only in its own partition, see below). This mode is left under two conditions, either when a non-maskable interrupt from a timer programmed by the hypervisor is asserted, or when a specific macro-instruction is executed (see Figure 3). Either way, the following procedure takes place:

1) save processor state to a specific location in the partition (see subsection III-E for details on this step).
2) flush and invalidate all cache memories;
3) reset the processor state (general purpose registers, state registers, etc.);
4) switch context of the memory management unit, as to only allow access to the hypervisor partition;
5) set the processor mode to hypervisor, with the PC positioned to a fixed address determined by the architecture (for example stored in a vector table).

5) VM protected: this mode is the typical user mode of any processor. Transitions between this mode and the privileged mode are not mandatory, should be supervised by the execution support of the VM, and have no impact on data security as they occur within the partition of the VM. Yielding the processor in this mode follows the same procedure as in the privileged mode (with a proper transition to the privileged mode first).

B. Memory partitions and protection

The memory is partitioned in several areas, each with a fixed size, and each is exclusively dedicated to the hypervisor or one VM. Only one partition is dedicated to the hypervisor, and at least one is reserved for the execution of a VM. The partitions must not overlap, and must cover the whole memory. It is possible though that some memory areas are shared between partitions, such as memory mapped IOs, and areas shared between the hypervisor and one VM for communication (see subsection III-C).

A specific hardware component must ensure isolation between the partitions. Unlike the usual memory management unit, it can only be configured in the initialization mode, with a mapping of partition ids to their address space. This enforces that neither the hypervisor nor the VMs can access its configuration registers and entries (not even for reading). Its role is:

- to define the number of partitions and their respective sizes;
- to check that outside the initialization mode, access is allowed to only one partition (the hypervisor’s in hypervisor mode, the currently executed VM’s otherwise);
- to automatically select the active partition when a transition between hypervisor and VM execution modes occur (and conversely).

The memory management unit should also have the typical memory protection/translation mechanisms, effective only within the active partition. The execution support of a software component (hypervisor or VM) should then be able to modify its own address space. This two-stage memory management is similar to what can be found in processors with hypervisor support (Second Level Address Translation on Intel processors, or Rapid Virtualization Indexing on AMD’s). The main difference is that the first level cannot be modified by the hypervisor. Another possible implementation is to add another translation unit before a typical memory management unit. This translation unit could be limited to a memory segmentation mechanism as found on x86 processors.
C. Communication

We consider essentially the IO operations corresponding to a network traffic going through one or more interfaces shared between the VMs. But the principles are the same for other IO operations, such as writing to a shared hard drive.

A lot of hypervisor implementations require that the VMs call the hypervisor for IO operations, especially on architectures that do not have hardware support for peripheral virtualization (IOMMU). In our blind hypervision architecture, we consider that the VMs must guarantee data security (confidentiality, authenticity) in the higher layers of the communication protocols. Given that a corrupted hypervisor spying a machine’s IOs have the same capabilities as any other node in the network, we consider that this precaution is necessary, not only due to the hypervision architecture.

This precaution is necessary even if the VM is authorized to access shared devices directly in its partition, as long as they are shared with other VMs or the hypervisor.

D. Loading virtual machines

The loader is a service called by the hypervisor. It transfers data to (respectively from) the memory partition of a VM with deciphering (resp. ciphering) on the fly. This service is the only component of the host that can access the private key used to decipher the symmetric key of a VM (cf. section II). Thus it is the only one able to authenticate to the master and receive the confidential data.

It can be implemented either in hardware as a specialized DMA or in software. Either way, it should be completely isolated from the hypervisor and the VMs. In the case of a software implementation, several precautions must be taken to trust the loader. First, to prevent injection of malicious code, the loader can be loaded to memory using authentication techniques as the UEFI Secure Boot [5]. Second, it should be executed in a dedicated environment that allows access to the partitions of the VMs it will load or unload. This can be achieved through several means, as a reserved memory area and protected in the normal execution modes, a secured VM in an ARM TrustZone [2] etc.

Regardless of its implementation, the loader is accessible in the hypervisor mode through an API described below. This API supervises the partition states following the state machine represented in Figure 4. The states and their meaning are:

- **free**: the partition is free to be allocated to a VM for loading;
- **loaded**: a VM was loaded to this partition, it is the only state that does not trigger an exception if the partition is selected for a transition from hypervisor mode to VM privileged/protected modes;
- **loading**: the partition is being loaded with a VM;
- **unloading**: a VM is being unloaded from the partition.

1) **partitionLoad**(k_e(k_v), p): this function finds a free memory partition and writes its id to the p parameter, resets its contents, deciphers k_v using a private key known only to the loader, and assigns k_v to the p partition. This function is sufficient if all VM partitions have the same size. Otherwise, an additional parameter giving the requested size of the partition is necessary. The function fails if there is no partition to satisfy the request. In case of success, the selected partition is *loading*.

2) **partitionWrite**(p, k_e(d), s): this function decipher the block of data k_v(d) of size s using the key k_v assigned to partition p, which must be *loading*. It writes the resulting clear block d to the memory partition p, after the last block written to this partition (at partition start if first call). The function fails if the partition is not *loading* or if it has not enough available space for writing the block.

3) **partitionRelease**(p, CRC): this function triggers the transition from *loading* to *loaded* for given partition p. It performs a CRC check on the deciphered data present in the partition. The function fails if the partition is not *loading* or if the CRC check fails. In the latter case, the partition’s content is reset and the partition state is set back to *free*. In case of success, the partition is *loaded*, and ready to be executed. In case of **partitionRelease** failure all commands except **partitionLoad** will fail.

4) **partitionUnload**(p, h): this function triggers the transition from *loaded* to *unloading* for given partition p. The second parameter gives the identity of the target host that will receive the ciphered image. The function fails if given partition is not *loaded*. In case of success, the partition is *unloading*.

5) **partitionRead**(p, k_v(d), s): This function reads the contents of the unloading partition p. It reads a clear block d of size s after the last block read from this partition (from partition start if first call). It then ciphers d using k_v and writes the result to the parameter k_v(d). The function fails if the partition is not *unloading* or if it has not enough available data for reading.

6) **partitionDelete**(p, CRC): this function triggers the transition from *unloading* to *free* for partition p. Using the clear contents of the whole partition, it generates a check sum and writes it to the CRC parameter. Finally, the partition is erased entirely and is *free* again for allocation. The function fails if the partition is not *unloading*. 

![Fig. 4: States of a VM partition in the loader; the free partition is available to be allocated to a VM, it is loading while the machine image is written to it, loaded when the machine is available for execution and unloading when the image is migrated to another host.](image-url)
E. Hypervision

Any hypervisor software can be used to run in the hypervisor partition on the hosts. The only condition is to modify the hypervisor to respect the memory protection and the interface to launch the VMs.

Among these modifications, the context switching must be adapted to the specific processor modes described above in subsection III-A. This means that the code for saving/restoring the processor state must identify the location of the execution context in the exact same way (for example a fixed address in the VM partitions). It must also use the same code. The best way to do this is to have context saving implemented as micro-code in the processor itself. If this is not an option, the code for these operation should be stored in a location that is protected from all kinds of modifications (can be configured in initialization mode), or the whole system becomes vulnerable to denial of service attacks from within a VM if a modification to the context saving code prevents the return to the hypervisor.

IV. APPLICATION TO MANYCORE PROCESSORS

The blind hypervision architecture can be slightly modified to be applicable to manycore processors with a hierarchical architecture. As an example, we take the Kalray MPPA processor, with 256 computation cores spread over 16 clusters, each cluster having an additional core for control [3]. The processor has 4 IO clusters that can be used to address external memory, and communicate over a PCIe bus or an Ethernet link. All the clusters are connected by a Network on Chip (NoC).

On this kind of architecture, the memory partitioning is physically implemented by the clusterization, as there is no possibility for clusters to access directly the memory of one another. So the IO clusters run the hypervisor to monitor the peripherals and handle the migrations, and the other clusters run VM images.

The VM loader can then be distributed over all the clusters as trusted code running on the control core. No other software component is allowed on the control core. The host key is then stored in a system register of the control core, guaranteeing its absolute privacy. The loader’s API can be implemented by a specific protocol on the control lines of the NoC. This way the hypervisor can initiate a transfer to a cluster, send the ciphered data blocks to the target loader who will decipher the block and write it to the cluster’s memory. Once the transfer is complete, the VM is started an executes on the 16 computation cores of the cluster.

The processor modes are not necessary in this kind of implementation. The initialization process takes place before loading the hypervisor by sending the host key and the loader code to the clusters. The hypervisor mode is a privileged mode on the IO clusters, and the VM privileged/protected modes are the privileged/protected modes of the processors on the computation clusters.

The memory protection inside the clusters remains an issue. With the code of the loader residing in memory, it could be accessed/modified by the VM running there. To prevent this, the code of the loader and some integrity information can be ciphered. When it is executed, the control core uses the host key stored in its private register to decipher it on the fly and check to code is still valid. If the loader appears to be corrupted, the cluster has to be considered no longer available for VMs.

To conclude on this alternative, it is interesting to note that it is similar to adapting blind hypervision to a distributed architecture, with several physical machines connected by a private sub-network. Vulnerability of the NOC are similar to the one on an external network and should be addressed with the same techniques.

V. RELATED WORKS

Hardware architectures for security have been largely investigated over the last decade [14], [10], [4]. There are several approaches, each trying to solve different security issues. A vast majority relies on a cryptographic coprocessor [8], [4], or a virtual processor in complete isolation from the rest of the architecture [2]. These approaches intend to guarantee security properties for some sporadic critical operations, such as loading system code, exchanging sensitive data, or in a more general fashion the execution of elementary cryptographic operations. They are definitely not designed to run a complete application from end to end, let alone a virtual machine in a secure environment.

What we intend to achieve is partially introduced in [12] as root security: even the administrator is denied the right to read or write the sensitive data stored on the host. But Smith applies the principle to data, stored on a database server, not to the execution of a virtual machine. This is taken into account in [6], but the proposed solution relies on the hypothesis that the hypervisor is in the Trusted Computing Base (TCB).

Much closer to what we propose, the hardware architecture eXecute-Only-Memory (XOM) offers superior confidentiality by allowing the blind execution of a VM [14], [10], [7]. The code and data are encrypted and stored in memory, and ciphered/deciphered on the fly at execution time inside the processor. The same principle is applied in [13], with a broader threat model, or in [9], in a more distributed fashion. The solution introduced here has a weaker threat model, as we consider the memory and memory buses to be trusted (once again, we do not pretend to prevent physical attacks). On the other hand, we believe we can achieve superior performance for a lesser hardware cost, as we don’t need additional operations on the code and data.

VI. FUTURE WORK

In this paper, we describe the general idea of the blind hypervision. The privacy properties are demonstrated, but the overall system performance remains to be evaluated as to compare this approach to the hardware implementations that perform on the fly ciphering/deciphering at execution [14], [10]. To achieve this, the next step is to implement blind hypervision by extending an existing hypervisor on a representative target.

As a proof of concept it could be possible to implement the Blind-Hypervision principles on an ARM based architecture where all hardware candidate extensions will be implemented by software in TCB, executed in the ARM TrustZone. In this
design, the selected VM Monitor has to be rearchitected to make use of those defined Blind Hypervision Host services. To fully demonstrate the concept only the advanced memory management, secured load/unload operations and VM context mode switching have to be trusted code. As much as possible, remaining Hypervisor code should run in user mode.

Beyond the proof of concept, this implementation would allow evaluating system performance and should help making trade-off between HW/SW implementation of parts of this concept.

VII. CONCLUSION

We introduced the general idea of a blind hypervision architecture that enforces the privacy of VMs in the case of a corruption of the hypervisor. This approach relies on software and hardware extensions, under the form of a specialized memory management unit and a trusted component to move VM images in memory, that are inaccessible to the hypervisor, relieving it of these responsibilities. By doing so, the hypervisor cannot access private data of the VMs, even if corrupted.

By removing on the fly ciphering/deciphering of the VM data during execution, we believe that this approach will reach the same data security level as other approaches while showing better performance. This is still to be verified by proper experimentation, that we intend to realize by extending an existing hypervisor and make a prototype on a representative target.

REFERENCES