Towards Bridging the Gap... [TBD]

Alex Palesandro
Orange Labs, France
palexster@gmail.com

Aurélien Wailly
Orange Labs, France
aurelien.wailly@orange.com

Marc Lacoste
Orange Labs, France
marc.lacoste@orange.com

ABSTRACT

1. INTRODUCTION

Cloud computing platform are a solid reality nowadays. Modern cloud platforms are able to provide elastic use of resources, optimizing their allocation in order to meet the user needings. However, currently deployed cloud platforms present still several limitation.

First, it is still difficult to deliver cloud applications which are able to transparently leverage over several platform resources. The concept of SuperCloud [?, ?] addresses this kind of problems. Multi-platform coordination primarily for different standard and several format lock-in. Unfortunately, those lock-in are not only due to market strategical decisions but they also reflects the virtualization technology heterogeneity. For example, each cloud provider could use a different storage system or a different format to store VMs images. Different hypervisors have nowadays a low degree of compatibility between them and this generally prevents leveraging on some popular features like " VM live migration" between different platform. Moreover, this incompatibility could also be enlarged if we consider that several platforms adopts one or more different virtualization techniques. In addition, this situation induce a tight dependency between the customer and the cloud provider and hardens the possibility to guarantee the availability of service in presence of a certain provider or to put in place some geographical optimization due to cost or traffic origin. The result was that some data server failures in the past has been catastrophic, causing a prolonged outage of service for a plenty of applications.. In the remaining part of the work, we will refer to those issues as "horizontal" requirements, that shows straightforwardly their multi-platform nature.

The second class of issues concerns platform integrity and user data privacy. On one hand, from an user-centric point of view, the user would like to preserve privacy of its data when it stores data on the cloud platform. However, hypervisor is normally the most privileged task executed in the platform and therefore a curious or malicious administrator could easily retrieve those data [?]. Moreover, in multi-tenancy cloud platform, a user shares the same physical resources with other users. Several works [?, ?] showed the potential threat and risks of attacks issued by other users located on the same physical resource. On the other hand, from a user-centric point of view, it is difficult to observe and inspect deeply the functioning of the whole platform, leveraging commodity general purpose hypervisors (GPHs).GPHs are executed at the most privileged level and so, it is complicated to detect and intervene on them in presence of malicious attack or misbehavior, without service interruption for all VMs hosted on the server. To sum up, both visions collide with the execution of the hypervisor at the highest level of privileged. This attitude prevents the platform administrator from a real monitoring of the whole platform, but also the possibility for users to concretely inspects actions that platform administrators do on user data. A major challenge of new generation cloud platform is to alleviate this different tension, finding an acceptable trade-off. In the following paragraphs, we will make reference to this class of problem as "Vertical" requirements.

To resume, in order to present a complete view of the architecture, we identified five key design principles, that a cloud computing platform has to implement in order to solve the cited problems. In the following paragraphs, those principles will represent a grid of analysis of different architectures.

DP1 User-Data privacy A curious or malicious administrator (or another platform user) would not have the right to inspect and analyze user data and instances, without the explicit consensus with the user above mutually trusted services. User-data privacy is a major challenge of modern cloud platforms. It is important to identify a trade-off that allows providers to identify and intervene on possible menaces, but also guarantee the impossibility for a curious or malicious operator to access to them without restrictions. In order to respect this principle when a platform administrator manages the platform it has not the highest level of privilege and each action on users object that he wants to deliver has to be validated by a mutually-trusted core, executed at the highest level of privilege. Moreover, this mutually-trust core has also to apply memory page cipher/uncipher in order to prevent memory page snooping when a page is swapped out.

DP2 Fail-Safe modular architecture In order to recover soft failures, the archtecure is supposed to be fail-safe, being able of automatically migrate instances or user-data in presence of a menace, but also reactivating the correct

.

Id	Functional design Principle	Class
DP1	User-Data privacy	Vertical
DP2	Fail-Safe modular architecture	Vertical
DP3	Small TCB	Vertical
DP4	Inter-platform support	Horizontal
DP5	Backwards 3rd part compatibility	Horizontal
NDP1	Minimize code addition	

functioning in presence of a software failure. In order to satisfy this property, the amount of safety-critical components has to be as lowest as possible and the overall design has to be kept as modular as possible

DP3 Small TCB The amount of lines of code executed at the highest level of privilege is normally proportional to the number of potential failures and bugs in the platform. According to this, an architecture presenting a tiny TCB will improve its safety and integrity properties by design.

DP4 Inter-platform support The cloud architecture has to be designed in order to support directly different platforms and service providers. The same user virtual instances have to be executed (and potentially migrated) unmodified over different platforms.

DP5 3rd part compatibility Avoid disruption of existing 3rd part applications compatibility could represent a major principle to consider. Platform interfaces has to be kept as close as possible to the existing one, preventing application rewrite.

Moreover, beside this functional design principles specifying which features have to be achieved by the platform, another set of non-design principles could be identified. Those principles does not concern directly features or aspects of the architecture of the platform, but the effective feasibility of the architecture.

NDP1 Minimize code addition a major issue could be represented by the amount of code lines that have to be added and maintained to an existing architecture to meet the functional requirement. Even it does not represent strictly a technical limitation to a certain proposal, it could become a "show-stopper" factor for its utilization.

In our analysis, it will be kept in consideration not only the overall suitability of the architecture, but also a quantification of the effort needed to develop and deploy it. In table 1 it is possible to observe a resume of previous cited principles.

The reste of the paper in organized as follows. In Section 2 we start analyzing related work. In section 3, we analyze current limitations of classical nested virtualization approach and in section 4 we analyze the microvisor approach. In section 5 we analyzes cloud architectures propositions, combining nested virtualization and microvisors, leveraging the design principles grid introduced before. Furthermore, in section 6 we define a security architecture for multi-cloud scenario, with an analysis of necessary software components to satisfy vertical design principles. In section 8, we conclude the paper with future works in section in section 6 we define a security architecture for multi-cloud scenario, with an analysis of necessary software components to satisfy vertical design principles. In section 8, we conclude the paper with future works in section 8.

2. RELATED WORK

Several works presented in literature integrates these a subset of this design principles in their architecture, leveraging nested virtualization [?, ?, ?] Cloud computing feature set could be enlarged exploiting two layers of virtualization instead of just one. The presence of two different hypervisors allows the possibility to specialize hypervisors functions. Instead of having a general purpose hypervisor we could split the hypervisor role in two different layers, interconnected with a well defined interface. Each layer will have to care about a subset of features, simplifying overall development. However, nested virtualization on x86 architecture faced impeding performance issues [?].

However, several works demonstrate that it is possible to put in place low overhead nested virtualization using special approaches [?] . Moreover, other works try to recover performance loss exploiting multi-level consolidation, introduced in nested virtualization [?]. In conclusion, nested virtualization slowdown could be considered at now an acceptable price to achieve advanced features and, in addition, hardware manifacturers are developing and proposing new hardware facilities to reduce this overhead [?].

To sum up, no architecture had been presented to address vertical and horizontal class of problems at the same time until now. XenBlanket does not investigate cloud architectural security concerns (DP3,DP2), limiting the focus on new potential feature of a coordinated multi-provider cloud. Cloudvisor proposed a security architecture that leverages nested virtualization without allowing the possibility of executing more than one nested hypervisor at time, that is a crucial requirements of a multi-provider cloud platform (DP4). In a SuperCloud scenario, each user has to run its own L1 hypervisor instance to enforce a compatibility layer, overcoming incompatibilities between different providers.

Other approaches leveraging only one layer of virtualization does not achieve the whole design-principle satisfaction. Self-service cloud approach (SSC) [?] design a complete security architecture to overcome major security and privacy issues of currently deployed cloud infrastructures, in order to alleviate the different needing user-centric and provider-centric vision of cloud computing presented in ??. Even if this architecture reproduces an approach similar to the SuperCloud without leveraging an extra layer of virtualization (DP4), the approach changes the behavior where the user acts introducing new components that have to be directly managed by the user. Therefore, the main drawback of this solution is to change the interface with the user, preventing the possibility to straightforwardly move to the new platform without having to change its applications (DP5).

3. CLASSIC NESTED VIRTUALIZATION APPROACH

Nested virtualization refers to a system architecture with two layers of virtualization. It consists in a "virtualized" virtualization, where the guest system virtualize a nested guest. The concept could be generalized to an arbitrary number of nested guest layers, introducing recursive virtualization.

Introducing nested virtualization, we have to identify the two main actors involved in the virtualization process: baremetal hypervisor (L0) and nested hypervisor (L1). In addition, we will refer to the nested guest as L2 guest. However, virtualization introduces a computational overhead. It is small when we are dealing with single layer virtualization, but it grows exponentially when we try to add further levels [?]. This huge performance overhead has prevented nested virtualization adoption during last years and had been finally addressed by Ben Yehuda et Al [?] proposing an architecture with acceptable overhead.

Nested virtualization feasibility opened a plenty of new possibilities and perspectives, that have been largely discussed in literature [?, ?, ?]. The first aspect in common among different works proposed in literature is to diversify hypervisor functions, introducing new features but keeping the existing ones. In a general analysis of existing works, it could be possible to identify the general role of each different layer. On one hand, L0 is charged of the vertical aspects of the platform. It has to achieve the best possible performances and enforcing the strongest possible level of isolation. L0 represents the last line of defense of the platform and a privileged point of global status observation.

On the other hand, L1 is responsible for horizontal feature set. it has to provide the widest possible support to different platforms and the capacity to virtualize adopting different techniques (para-virtualization, hardware-assisted, dynamic binary translation). According to these requirements, Williams et al. [?] propose a patched version of Xen, Xen Blanket, to accomplish a L1 virtualization layer, designed to run over different platforms. This concretely allows to run over different physical platform the same VM, and live-migrate it at the occurrence.

3.1 Classical nested virtualization limits

Nowadays, nested virtualization presents several limits related to the hardware support and software state of implementation. At first, to provide nested virtualization, hypervisors has to provide an exact copy of hardware virtualization facilities. The most exploited technique, full virtualization with hardware assistance (HV), relies on a quite complex dedicated support. This set of facilities assists instruction emulation, memory address translation and device management. Both Xen and KVM propose nested VMX/SVN features, that consists in the possibility to efficiently handle nested hardware virtualization, in presence of a single-layer hardware facilities. Although nested virtualization performance are still not good and the feature is not yet stable on modern hypervisor [?], the feature is under heavy development and it will be hopefully in a usable state soon.

Secondly, due to the monolithic architecture, each feature introduced in GPH enlarges the TCB. Therefore, leveraging GPH, a huge amount of code is running in the most privileged mode and has to be trust. Several works discuss and analyze the overall TCB of modern GPH [?, ?], considering GPH TCB always bigger than 100 KLoC. This approach directly collides with one design principles introduced in the introduction paragraph (DP3),

Finally, the above architecture does not improve the overall stability of the infrastructure. The underlying core, L0, has the same problem of single-layer GPH architecture: the large TCB implies also that a failure experienced in one of the critical components will simply produce a generalized failure of the whole platform, not satysfing DP2. Moreover, in GPH standard architecture, the administrator has the entire control of the platform and could potentially snoop and inspect user VM status and data.

This kind of architecture, in particular, does not leverage completely the possibility to hide behind a compatible layer, a really modular architecture with only a tiny critical core. To sum up, current nested virtualization limits are represented by a certain lack of dedicated feature in modern hypervisor and the impossibility to rely on a tiny TCB (DP3,DP4). The first one is being addressed by hypervisor developers. The second one could simply not be achieved with modern hypervisors.

4. MICROHYPERVISOR

Consequentially, several works focused this problem leveraging a micro-hypervisor architecture, with a traditional microkernel-like design that tries to consistently reduce the amount of code running with the highest privilege level. A micro-kernel traditionally consists in a tiny core running in Kernel Mode, offering only few fundamental services to other modules running (scheduling, IPC, memory management) in a less privileged domain. All other components, in particular device drivers, are moved out in user-space. Microkernels had been questioned because of the wasteful performance of first generations, due to IPC intrinsic cost. This argument had been completed confuted by Liedtke in 1993: [?, ?].

Microkernels have the capacity of easily redesign the hierarchy of privileges. A user application has normally to trust the tiny micro-kernel and only the set of user-space software it uses. However, this set of trusted set of drivers and tools have not to be trusted by every application executed in the system. Normally, this allows microkernels to implement different “personalities”. User applications shares the system but could have to deal with completely different system services in a transparent way. However, “personalities” interest had been hugely reduced by the fact that system services, device drivers and even application had to be completely re-conceived to fit this new design pattern. Therefore, with the introduction of VMs, microkernels lost the battle in favor of hypervisors, in terms of popularity. However, the great advantage of tiny TCB, provided by microkernel, was not provided also by hypervisors. As we analyzed before, they present normally a monolithic architecture, like the great part of modern OS. This feature kept alive the interest in microkernels, in particular trying to couple microkernel benefits with virtualization.

Therefore, leveraging this concept of microkernels, a micro hypervisor (MH) is composed from a small kernel providing only essential functions, like scheduling, IPC, memory management and VMX extension. Several micro-hypervisor have been proposed in literature [?, ?]. They rely on the micro-kernel paradigm, extending it to virtualization. MH naturally satisfy vertical design principles (DP1,DP3, DP4). The tiny core could be considered the only system-critical componente of the architecture and, due to its relative small dimension and the consequent minor attitude to vulnerabilities,it could represent the “last line of defense”.

In addition, MH inherits the concept of micro-kernel to implement different “personalities”. Those “personalities” are concretely a set of “processes” handling specific functions running in user-space. As those functions are implemented outside the kernel, they are not unique but they are changeable and could coexist at the same time. This feature was seen with interest because of the possibility to move some virtualization-related from the kernel space to user-space and to strictly separate piece of software executed for different users. This extraction process drives to a more solid platform, considering that user-space crashes are generally not fatal for the entire system integrity. A possible example, that will be analyzed in next section, is the user-space VMM component of Nova, Vancouver, that resides in user-space [?].

4.1 MH general issues

Despite the positives, MH approaches had demonstrated some limitations. Firstly, the straightforward one is the initial lack of hardware support, lack of device drivers are still a relevant Achilles' heel for this solutions, inherited from its ancestors microkernels. Fortunately, this kind of issue could be addressed leveraging I/O passthrough techniques, although it will cause exclusive allocation issues.

Moreover, as reported in previous sections, several cloud computing issues are simply not addressable with general purpose hypervisor (GPH). However, they are commonly widely adopted and represent the bedrock of existing platform. Therefore, GPH exports wide and solid interfaces leveraged by a plenty of different software that is able to properly communicate and act through them on the virtualization layer. On the contrary, MH lack of cloud toolkit integration, requiring an explicit support to be written, not satysfing 3rd part compatibility requirements (DP5). GPH are tightly connected with cloud toolkit and could present solid interfaces to communicate with different tools.

Finally, microkernel modularization introduce a major challenge in architecture design. Due the increased complexity, for sake of kernel size, MH could generally lack of several virtualization features, commonly available on modern GPH hypervisor. For instance, XMHF does not support multiple guests. This kind of drawbacks could limits the potential interest in MH, even they are able to introduce several important features.

5. ARCHITECTURE COMPARISON

The adoption of an extra layer of virtualization is capable to introduce a remarkable set of new features but it does not reduce the dimension of Trusted Code Base (TCB). MH could represent a new approach to improve the potential benefits provided by the adoption of nested virtualization. In the following subsections, it will presented the four possible architectures achievable "nesting" of GPH and MH, underlying potentialities, drawbacks and potential cost of implementation.

5.1 GPH over GPH

The first analyzed architecture concerns only GPH. At first, it is important to recall the most interesting achievable by such architectures. In particular, several remarkable features are live migration over different platform, transparent migration of virtualized clusters between different datacenters (horizontal requirements DP1). Supposing the completeness and the correctness of nested virtualization hardware facilities in widespread hypervisor, GPH over GPH satisfies all horizontal design principles, but it lacks in vertical DPs as the size of TCB is the same of currently deployed architectures.

To sum up, it could be possible to achieve live-migration between different platform and other features related to VM management flexibility but it could not still be possible to reduce the TCB. Analyzing MH approaches, the aim is try at most to retain to features, trying to introduce isolation and privacy improvements due to the lower TCB promised.

5.2 MH over MH

MH on MH in reported only to see the orthogonality between this approach and GPH over GPH. A theoretical implementation of MH over MH will provide the vertical requirements that are missing in the previous approach. The tiny core could be able to overcome monolithic problems due to privilege level of adminitration tasks (DP1), the possibility to improve the resilience of the platform to a large set of software failures (DP2) and finally reduce the possibility of the vulnerabilities of the most privilege part of the software (DP3). However, the interest on such architecture is limited because it is not able to overcome any cited issues of single-layer MH. Moreover, several horizontal properties are theoretically achievable but they will require a huge effort in developing functions of interest. Moreover, the lack of nested VMX functionalities prevents to concretely test this kind of architecture, because MH normally rely on VMX facilities in order to create a nested HVM guests. To conclude, nested virtualization is exploited in order to fill eco-systems gaps with GPH but MH over MH does not provide any improvement in this sense.

5.3 MH over GPH

The third architecture, proposed in order to reduce security concerns, is built combining an GPH at L0 and a MH at L1. An interesting point of this approach is the fact of being easily be deployed, without requiring any extra code addition to the GPH, L0. Existing GPH-based platforms could potentially exports nested VMX features to those guests. Even if this is not so usual in modern cloud platform, this lack is considered temporary and related only to the experimental status of those features.

However, the amount of code running in L0 kernel space is still as big as before and, therefore, no TCB reduction. Finally, adopting a MH as L1 prevents to use inter-platform enhanced features. As analyzed when dealing with MH design, the price of having a tiny core is normally paid with a lack of certain functionalities. For example, XMHF only supports one single guest. It could be easily noticed that this approach is not architecturally able to provide any new "horizontal" features, leveraging on nested virtualization. Therefore, the architecture is only able to provide the same features that are provided by today deployed ones, with an enhanced but not sufficient level of isolation.

5.4 GPH over MH

In this configuration, the TCB is smaller than other architectures, due to the modular structure of micro-kernels., executed at L0. Leveraging as L1 a GPH, like Xen or KVM, provides to the user an environment apparently identical to the one used in production today, satisfying DP5. The L1 GPH is normally integrated with modern cloud toolkits like OpenStack and will support 3rd part applications. Therefore, the architecture presents a stronger isolation leveraging on nested virtualization, providing in particular a strong tiny core(DP3). The MH layer will be transparent from the user and it will not not any change in the functioning of the platform.

The first class of possible drawbacks of this architecture is represented by the necessity to expose to the L1 nested

Id	Functional design Principle	Class	GPH over GPH	MH over MH	MH over GPH	GPH over MH
DP1	User-Data privacy	Vertical	×	✓	×	✓
DP2	Fail-Safe modular architecture	Vertical	×	✓	×	✓
DP3	Small TCB	Vertical	×	✓	×	✓
DP4	Inter-platform support	Horizontal	✓	×	×	✓
DP5	3rd part compatibility	Horizontal	✓	×	×	✓
NDP1	Minimize code addition	-	✓	×	×	✓

hypervisor the nested VMX extension to let him "nest" another HVM guest. This is experimentally done by GPH like Xen and KVM, but it is generally not included in the essential features provided by a minimal micro-hypervisor. As reported in XMHF paper [?], the amount of lines of code for nested VMX is around 1KLOC. More generally, from an architectural point of view, the general MH minimalist attitude could represent a limitation to general preservation of existing platform functions. The constraint of privileged code size forces normally MH developers to drop some features, considered not essential. Therefore, several limitations could arise not only for nested virtualization proper features, but also for other class of features. For instance, XHMF is able only to virtualize one L1 instance at time. This is acceptable if the platform objective is limited to be provided of a secure core, but dramatically limits the possibility of having a platform model able to give the user the possibility to run its proper hypervisor as L1.

Another important class is the necessity to explicit hardware devices support. Device drivers represents a general Achilles' heel of MH, due the general cost of development. However, this class of problem could be addressed leveraging device assignment. L1 GPH hypervisor will handle the real device, using the proper driver now demoted to run in a lower level of privilege. It has to be underlined that this solution could represent a feasible approach only if it possible to overcome the exclusive allocation trouble (SR-IOV).

5.5 Conclusion

To conclude, this section provides a short comparison between different platforms, contrasting architecture features above the two main different types of benefits given by nested virtualization, as reported widely in literature [?, ?, ?, ?]. The first could be identified in the augmented degree of isolation the architecture could enforce between host OS and the "customer" point of access, allowing to redesign the security architecture of the whole platform. The second one is represented by the possibility to leverage multiple virtualization layers to easily define inter-platform policy and operations between different platforms. In table 5.4, it is possible a summary of the previous analysis. To conclude, GPH over MH is the most interesting one in order to enlarge nested virtualization promised benefit. This architecture with a solid and secured MH as L0 is the only able to reduce the platform TCB and this could be a key point to obtain more protected infrastructure, . Firstly, adopting a MH as architecture trusted core, we will have a secure layer guaranteeing that administrator could not lose the control of the system. Klein et Al. [?] have demonstrated that a kernel with less than 10KLoC could be formally verified, under several conditions. This is far from be achievable today with standard GPH over GPH approach where the core is not tiny and the TCB is an order of magnitude bigger than MH [?].

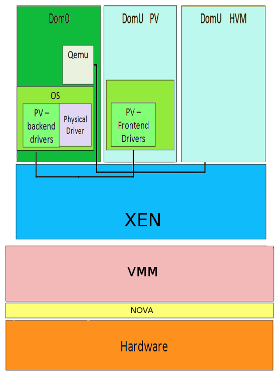

Figure 1: Xen over NOVA architecture

However, the effort to set up a GPH over MH is strongly dependent on which technique leveraging for virtualizing L2. As stated in paragraph 5.4, with L2 guest virtualized with the hardware assitance technique, the most popular one, the MH needs to export nVMX at least. On the contrary, adopting paravirtualization/Dynamic binary translation for L2 does not necessitate a particular hardware support and so it could not be necessary patch the MH to set up a platform. However, it has not be underestimated, however, that para-virtualization supports only a subset of OS, and could not be considered a general solution and DBT has shown important overhead.

To sum up, the adoption of MH could increase feature set introduced by nested virtualization adoption. The first impeding problem is the implementation of MH, able to propose to guest to virtualize with hardware assistance. In next section, we present an implementation of GPH over MH, that try to circumvent this limitation leveraging paravirtualization.

6. IMPLEMENTATION

In the previous sections we proposed a functional analysis of cloud architecture in order to respect design principles that we identified in the introduction. In this section, we propose a GPH over MH implementation, based on Xen over NOVA.

Inter-platform support Executed as L1, Xen would virtualize its nested guest initially using PV delivering the compatibility layer.

Tiny TCB The NOVA kernel will represent the only part of the platform executed at the highest level of privilege.

Even the VMM, Vancouver, will be executed in the L0 user-space. The micro-kernel in L0 enforces a strictly modular approach to L0. Therefore, the VMM will present distinct processes instances for each L1 guest, enforcing a context division.

3rd part compatibility From a user point of view the architecture still exports the same interfaces provided by the Xen hypervisor, allowing the user to keep its applications unchanged.

Fail-safe components architecture Moreover, the tiny core is the only safety-critical software component. The system could potentially be able to recover any crash in different user-space components.

7. CONCLUSION AND FUTURE WORKS