From Nested Virtualization Architecture To 2D Security Management for Distributed Clouds

Alex Palesandro
Orange Labs, France
alex.palesandro@gmail.com

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

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

ABSTRACT

Distributed cloud computing is now moving user-centric. The promise of this evolution is: (1) unified resource management across IaaS providers, e.g., live migration independent from hypervisor technology; (2) increased customizability to choose the hypervisor services needed to realize fully à la carte, self-service clouds. However, a reality check shows that major interoperability and security barriers must be lifted, and complex trade-offs be found between functionalities provided and security assurance. Nested virtualization (NV) brings both a homogeneous interface for distributed hypervisor features and enhanced IaaS security, protecting VMs even in case of hypervisor compromise. However, practical NV adoption is still hindered by code additions to an already very large hypervisor TCB and by limited hardware support. New distributed IaaS architectures are therefore required.

In this paper, we argue that micro-hypervisor (MH) architectures are the key to overcome such NV limitations. From the previous challenges, we derive a set of design principles that a distributed IaaS architecture should satisfy. We compare possible architectural designs combining NV with micro-hypervisor features to find the best trade-off between user data privacy, modularity, TCB size, and compatibility with multiple platforms. We sketch how the resulting architecture may serve as a basis for 2D security management of a cloud of clouds: vertically, across layers for hardened IaaS security, and horizontally spanning cloud domains for end-to-end security.

1. INTRODUCTION

Distributed cloud computing is now becoming a reality. Modern cloud platforms are able to provide elastic use of resources, optimizing their allocation to meet user needs. However, currently deployed distributed cloud infrastructures present several limitations.

DCC does not handle heterogeneous infrastructures which secludes providers in non-evolving hardware. Also, security remains the last hurdle toward wide adoption of DCC. The user data are no more self contained but distributed among all parties without any proof of confidentiality, integrity or availability.

We classified problems related to DCC horizontally and vertically. The horizontal view gathers features related to interactions between IaaS stacks, such as live migrations. The vertical view addresses features close to the isolation of execution environment on the same physical machine. This classification provide a global coverage of problems to solve to deliver a secure, stable and robust architecture.

As a first element of answer, the nested virtualization [1] performed a breakthrough with enhanced isolation capabilities on the same physical machine. The hypervisor is now virtualized by a thin layer [18] to counter failures. Furthermore, the horizontal challenges were also included in this tiny layer to deliver an homogeneous interface for commodity hypervisors [14, 15]. However these classic approaches suffers from several shortcomings detailed in Section 2. Thus we propose a novel architectural approach to undertake remaining problems.

The previous works created both horizontal and vertical basic blocks from scratch. While being a necessary step, these approaches does not inherits the advances performed in terms of operating system security. The software architecture are mainly monolithic and yet-another-layer fends off security problems without figuring them out. Our approach learns from security issues fixed by OS architecure [] and apply these patterns to leverage nested virtualization. In this paper we show that component-based architecures are a promising design for the next generation of clouds.

This paper is organized as follows. Section 2 details the constraints of nested virtualization for the two dimension of our solution. In Section 3 we analyze the nested virtualization ecosystem and position existing solutions regarding our design principles. In Section 4, we details limitations of classical nested virtualization approach and in Section 5 we present the microvisor approach. In Section 6 we compare cloud architectures propositions, combining nested virtualization and microvisors, leveraging the design principles grid introduced before. In Section 10, we conclude the paper with future works and our vision of the SuperCloud.

2. CHALLENGES AND DESIGN PRINCIPLES

First, deploying cloud applications able to transparently leverage resources of several heterogeneous platforms is still difficult. The concept of super-cloud [14, 15] tries notably to overcome such issues. Multi-platform coordination is hard, primarily due to lock-in to different standards such as VM formats. Unfortunately, such lock-in is not only due to market strategic decisions, but also reflects heterogeneity of virtualization technologies. For example, each cloud provider may use different storage systems or different formats to store VMs images. Different hypervisors have nowadays a low degree of compatibility between them. This generally prevents leveraging some popular features like VM live migration between different platforms. Incompatibility is even greater if we

.

consider that each platform may adopt several types of virtualization techniques. This situation induces a tight dependency between the customer and the cloud provider. It may prevent guaranteeing service availability by relying on alternative providers, or to set up geographical optimizations due to cost or origin of traffic. This has resulted in some dramatic data server failures in recent years, causing prolonged outage of service for many applications. In what follows, we refer to this class of multi-platform issues as *horizontal* challenges.

Second, a number of issues have been raised about user data privacy and platform integrity. From the user perspective, users would like to preserve privacy of their data stored on the cloud platform. However, due to overprivileged hypervisors, a curious or malicious administrator could easily retrieve such data [18]. In a multi-tenant cloud platform, physical resource sharing between users may lead not only to data leakage or but also attacks against the IaaS [8, 10]. From the provider perspective, a key objective is to thus guarantee platform integrity against rogue VMs. However, commodity general-purpose hypervisors (GPHs) make it difficult to observe deeply all layers of a running IaaS platform. Indeed, running GPHs at the highest level of privilege hampers detection and reaction on the hypervisor itself in presence of faulty or malicious behavior – unless to cause interruption of service for all hosted VMs. Thus, both user-centric and provider-centric visions of the cloud have a problem with overprivileged hypervisors. A major challenge for new generations of cloud infrastructures is thus to find acceptable trade-offs between such conflicting requirements. In what follows, we refer to this class of layer-oriented issues as *vertical* challenges.

We identified five key design principles (DP) that a IaaS architecture should implement to meet the previous challenges. In what follows, those principles will serve as reading template to analyze different architectures.

DP1 User Data Privacy A curious or malicious administrator (or another platform user) should not be authorized to inspect and analyze user data and VM instances without explicit user consent relying on mutually trusted services. A trade-off should be found to allow providers to detect and react to potential threats, but also to prevent unlimited platform and user data access in such inappropriate operator behaviors. Thus, a platform administrator should not manage the infrastructure directly using the highest level of privilege. Instead, his actions on user data objects should be validated by a mutually-trusted kernel run in the most privileged mode. This kernel should also enforce memory page encryption to prevent snooping when pages are swapped out.

DP2 Fail-Safe Modular Architecture The architecture should be fail-safe: for instance, to enable automatically migration of instances or user-data in presence of a threat and to recover from software failures. To reach this goal, the amount of safety-critical components should be as low as possible to mimize the attack surface. Similarly, the overall design should be as modular as possible to enable platform reconfiguration.

DP3 Small TCB The number of potential failures and bugs in the platform is directly linked with the code size executed at the highest level of privilege. Thus, an architecture with a tiny TCB (Trusted Computing Base) will improve safety and integrity by design.

DP4 Inter-Platform Support The cloud architecture should allow supporting directly different platforms and service providers. The same user virtual instances should be allowed to be run (and potentially migrated) unmodified over different IaaS platforms.

Table 1: Design Principles for a Distributed IaaS Platform.

Id	Design Principle	Class of Requirement
DP1	User Data Privacy	Vertical
DP2	Fail-Safe Modular Architecture	Vertical
DP3	Small TCB	Vertical
DP4	Inter-Platform Support	Horizontal
DP5	Compatibility with Legacy	Horizontal
NDP1	Minimal Additional Code	–

DP5 Compatibility with Legacy Evolution in the IaaS architecture should avoiding disrupting existing third party applications. Thus, platform interfaces should be kept as close as possible to existing ones, to prevent code rewriting notably for applications.

In addition to the above functional design principles that specify which features have to be achieved by the platform architecture, another set of non-design principles (NDP) should be considered regarding effective feasibility of implementing the architecture. We just give an example in terms of platform usability:

NDP1 Minimal Additional Code A major issue could be represented by the amount of code to be added and maintained to an existing architecture to meet functional requirements. Despite not being a strictly a technical limitation of the architecture, it could become a show-stopper to its practical use.

In our analysis, we will consider not only the overall suitability of the architecture, but also the amount of effort needed to develop and deploy it. Table 1 summarizes the previous principles and which class of requirements they address.

3. RELATED WORK

Several works presented in literature integrates these a subset of this design principles in their architecture, leveraging nested virtualization [1, 14, 18] 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 [4].

However, several works demonstrate that it is possible to put in place low overhead nested virtualization using special approaches [1] . Moreover, other works try to recover performance loss exploiting multi-level consolidation, introduced in nested virtualization [14]. In conclusion, nested virtualization slowdown could be considered nowadays an acceptable price to achieve advanced features and, in addition, hardware manifacturers are developing and proposing new hardware facilities to reduce this overhead [7].

To sum up, no architecture had been presented to address vertical and horizontal class of problems at the same time until now. Xen-Blanket 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) [3] 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 1. 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).

Furthermore, DeHype [16] presented a dissected version of KVM hypervisor, trying to reduce the TCB of this popular hypervisor. The approach consisted in demoting a great part of the hypervisor codebase to a user-level execution. DeHype satisfies all vertical design principles listed before (DS1,DS2,DS3), reinterpreting the microkernel approach defining the greatest part of code to userspace. DeHype increases a lot the robustness of a cloud platform leveraging KVM, limiting for example the possibility of attack in multi-tenancy platform, but it does not address horizontal design principles about a multi-provider service.

4. CURRENT NESTED VIRTUALIZATION

4.1 Principle

Nested virtualization (NV) refers to a system architecture with two layers of virtualization [1]. It consists in "virtualized" virtualization: the guest system virtualizes a nested guest. The concept may be generalized to an arbitrary number of nested guest layers, leading to recursive virtualization [4, 9]. (Full) virtualization is able to execute tasks in a deprivileged context that normally requires a privileged execution. With NV, this approach may be extended to virtualization itself: a guest hypervisor may be run with nested guest VMs without requiring any change to any of them.

Two layers should be distinguished when migrating to an NV architecture: (1) the hypervisor running above the hardware (*L0 hypervisor*); and (2) the nested hypervisor (*L1 hypervisor*). Nested guests are generally referred to as *L2 guests*. Unfortunately, virtualization comes with a price due to the extra layer of indirection introduced: the computational overhead is generally acceptable when dealing with single-layer virtualization, but grows exponentially when adding further levels [4]. This huge performance overhead prevented NV adoption during many years. However, Ben Yehuda et al. finally proposed an architecture tending towards more reasonable overheads [1].

NV feasibility opened plenty of new possibilities and avenues for research that have been largely discussed in the literature [1, 4, 14]. Such works all tend to diversify hypervisor functionalities, introducing new features but also keeping existing ones. It turns out that each layer addresses clearly different sets of issues.

L0 tackles vertical aspects of the platform. It aims to guarantee the best performance and strongest level of isolation possible. L0 represents the last line of defense of the platform and a privileged point for monitoring of its global status.

L1 is responsible for horizontal feature set. It has to provide the widest possible support for different platforms and the ability to virtualize using different techniques e.g., para-virtualization, hardware-assisted virtualization, dynamic binary translation. To address such requirements, Williams et al. [14] proposed a patched version of Xen, Xen Blanket, to realize an L1 virtualization layer designed to run over different hypervisors. This notably enables to run and to perform live migration of the same VM over different physical cloud infrastructures.

4.2 Limitations

Today, NV presents several limitations related to hardware support and software state of implementation.

First, to support NV, the hypervisor has to provide an exact copy of hardware virtualization primitives. Full virtualization with hardware assistance (HAV) is the most popular technique, but requires quite a complex dedicated support. HAV primitives assist instruction emulation, memory address translation, and management of devices. Both Xen and KVM propose nested VMX/SVN features that allow to efficiently handle nested hardware virtualization on single-layer hardware primitives. Full virtualization is currently the only virtualization technique supported for L1 virtualization due to its ability to run unmodified guest OSes – and thus hypervisors. Paravirtualization (PV) would on the contrary require an ad hoc modified hypervisor to act as L1 (thus violating the NDP1 principle). Despite still poor NV performance and not being yet stable on modern hypervisors [17], such feature is under heavy development and will hopefully be in a usable state soon.

Second, due to current monolithic GPH architectures, each feature directly introduced within the hypervisor such as security enhancements enlarges the TCB [18]. Therefore, a huge amount of code runs in highly privileged mode and must be trusted. Several works discuss and analyze the overall TCB of modern GPHs, always bigger than 100 KLoC [11, 13]. This approach directly violates the DP3 design principle. Moreover, the previous architecture does not improve overall stability of the infrastructure. The bottom-most hypervisor L0 faces the same issues as single-layer GPH architectures: a large TCB implies also that a failure experienced in one of its critical components may trigger a generalized failure of the whole platform (violating the DP2 principle). Besides, in standard GPH architectures, the administrator has the entire control of the platform and could potentially snoop and inspect user VM status and data (violating the DP1 principle). Such designs notably do not take advantage of the possibility to use a platform compatibility layer, and more modular architectures with only a tiny security-critical kernel.

Overall, current NV architectures lack some of the dedicated features found in modern hypervisors and the impossibility to rely on a tiny TCB (violating principles DP3 and DP4). If the first one is currently being addressed by hypervisor developers, the second simply has not be achieved with modern hypervisors so far.

5. MICRO-HYPERVISORS

5.1 Principle

To solve the TCB size issue, several works have introduced the idea of *micro-hypervisor (MH)* architecture [11, 12, 13]. To reduce the hypervisor size, such new hypervisor architectures draw inspiration from evolution in OS architecture (e.g., extensible, micro-, exo-, component-based kernels). Within the hypervisor are distinguished security-critical modules from non-sensitive code. The aim is to expel as much code as possible from the TCB and target ultra-thin hypervisors. For instance, some components of the hypervisor (e.g., device drivers, VM address space management) may be virtualized outside the micro-hypervisor, isolated, and restarted in case of compromise [11]. The extreme case is to remove the virtualization layer altogether [12].

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 minimal kernel running in privileged mode, offering only a few fundamental services to other modules (e.g, scheduling, IPC, memory management) running in a less privileged domain. All other components, such as device drivers, are moved out in user-space. While the benefits of such a design has been questioned due to the cost of IPCs in first generations of micro-kernels, such arguments have since then been completely dispelled [6, **?**].

One of the benefits of micro-kernels is to enable to easily re-design a hierarchy of privileges. A user application normally trusts the micro-kernel and only the set of user-space services it requires. However, such drivers and libraries may not be trusted by every application run on the system. User applications share the system but may need to support completely different set of system services. This allows micro-kernels to implement different "personalities". However, the interest of such "personalities" has been largely reduced by the fact that system services, device drivers and even applications have to be completely re-designed to fit this new architectural pattern. Therefore, with the introduction of VMs, micro-kernels 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 [13, 11]. 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, each user could rely on a different instances of the hypervisor user-space process, that could coexist at the same time. This process separation between user could enforce a straightforward mechanism to neutralize VM-to-VM attack from a malicious user. Moreover, user-space processes are easier to debug with commodity tools (GDB, valgrind). Finally, 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 [11]. In addition, this approach is not far from the one proposed in DeHype [16], where a consistent part of the KVM hypervisor has been demoted in user-space.

5.2 Limitations

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.

6. ARCHITECTURE COMPARISON

Adding an extra layer of virtualization enables to introduce a wide range of new features in the IaaS infrastructure, but does not reduce the TCB size. As already discussed, introducing MH design within a nested IaaS architecture could represent a new approach to improve NV benefits by overcoming such limitations. In what follows, we analyze the four possible architectures to achieve nesting of GPH and MH, highlighting their potential, and assessing their benefits and weaknesses w.r.t. to the design principles of Section 2, as well as their cost of implementation.

6.1 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 data-centers (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.

6.2 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 administration 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

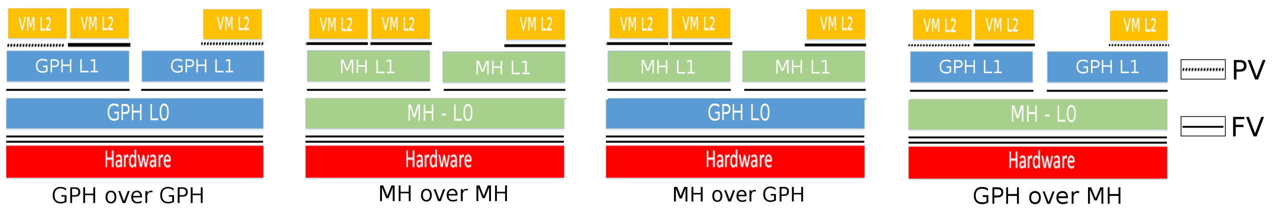

Figure 1:

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.

6.3 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.

6.4 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 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 [13], 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).

6.5 Lessons Learned

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 [14, 2, 18, 1]. 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 6.4, 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. [5] 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 [11].

However, the effort to set up a GPH over MH is strongly dependent on which technique leveraging for virtualizing L2. As stated in paragraph 6.4, 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

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	-	✓	×	×	✓

implementation of GPH over MH, that try to circumvent this limitation leveraging paravirtualization.

7. SECURITY ARCHITECTURE

8. VERTICAL PROPERTIES

9. 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 [11].

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 microkernel 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.

10. CONCLUSION AND FUTURE WORK

In previous sections we provided a global analysis of modern challenges of cloud computing and how those could be globally addressed addressed leveraging nested virtualization and different hypervisor architectures.

Future works will be focused on two complementary parts. Firstly, they will concern the completion of the architecture of security in order to concretely meet horizontal design principles, identifying key components, similarly as done for the vertical properties. At the same time, a major effort will be directed on enlarging the initial implementation.

11. REFERENCES

[1] M. Ben-Yehuda, M. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[2] O. Berghmans. Nesting Virtual Machines in Virtualization Test Frameworks, 2010.

[3] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 253–264, New York, NY, USA, 2012. ACM.

[4] B. Kauer, P. Verissimo, and A. Bessani. Recursive Virtual Machines for Advanced Security Mechanisms. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) Workshops*, 2011.

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[6] J. Liedtke. Improving IPC by Kernel Design. In *SOSP*, pages 175–188, 1993.

[7] J. Nakajima. Making nested virtualization real by using hardware virtualization features, 2013.

[8] D. Perez-Botero, J. Szefer, and R. B. Lee. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *Proceedings of the Workshop on Security in Cloud Computing*, SCC, May 2013.

[9] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[10] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, 2009.

[11] U. Steinberg and B. Kauer. NOVA: A Microhypervisor Based Secure Virtualization Architecture. In *ACM European Conference on Computer Systems (EUROSYS)*, 2010.

[12] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[13] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. *2013 IEEE Symposium on Security and Privacy*, 0:430–444, 2013.

[14] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *ACM European Conference on Computer Systems (EUROSYS)*, 2012.

[15] D. Williams, H. Jamjoom, and H. Weatherspoon. Plug into the Supercloud, 2013.

[16] C. Wu, Z. Wang, and X. Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2013.

[17] R. Yongjie. nested virtualizaiton test report for Xen 4.3-RC1.

[18] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011.