Whitepaper:

The future of PaaS in confidential computing:

a customer perspective

This technical document serves two purposes:

1. To provide a first clear and broadly shared customer view on what it takes for a PaaS compute service like AWS ECS or Azure AKS to become a production-grade Confidential Computing (CC) solution by articulating the most stringent IT security requirements customers could express ;
2. To propose a technical solution to showcase that there is no technological blocker to these requirements.

The original version may be found on linkedIn:

https://www.linkedin.com/pulse/towards-confidential-paas-detailed-whitepaper-christophe-parisel/

Part 1 - hard customer requirements

When considering CC, two extremely common pitfalls of Cloud customers are:

- *they embrace the feature eagerly and without questions*: we have already seen several big companies running confidential workloads into production on Confidential Computing IaaS pointing out the many benefits but without highlighting the caveats of present-day state of the art;
- *they deny the possibility of using CC on insufficiently detailed grounds*, which doesn't help for making current solutions any better;

I believe it is critical to set industrial standards which are commonly shared and approved among high-demanding customers (i.e., from regulated industries), to avoid that the definition of such standards is left to providers and chipmakers' sole judgement. The lack of customers implication is likely to bring CC into some dead end where the delivered solution is not being adopted by the customers who, as it turns out, need it most.

So here is a proposal of hard (non-negotiable) customer requirements:

REQUIREMENT #1: CONFIDENTIAL BOUNDARY

If you've followed by linkedIn posts on CC, you may have noticed I keep claiming that to design a proper confidential PaaS, one must design a proper confidential IaaS first. What do I mean by this?

In Iaas, the confidential boundary is clear: it is the whole VM itself (except for SGX enclaves, which are only sparingly used). It is called a **CVM** (Confidential VM).

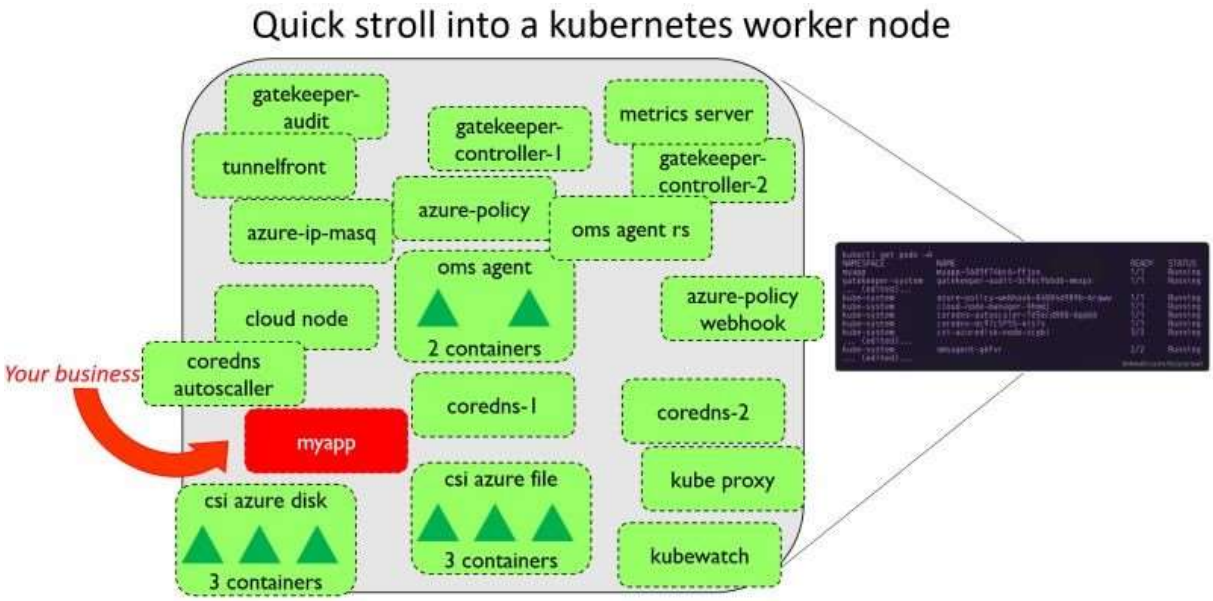
The reason for it is that no Cloud Service Provider (CSP) code must be allowed to be stored or executed into a customer compute environment: the CVM image must not contain any CSP products or libraries, including firmware itself.

In PaaS, the compute environment is split into worker nodes (aka 'data plane') and master nodes (aka 'control plane'). If the latter is under full control of the CSP, the former is a customer compute environment and must be enclosed into a CVM.

This constraint on the confidential boundary dictates the following requirements:

Requirement 1.1: in PaaS, the confidential boundary is the worker node. (Put it another way: each worker node is a CVM).

Requirement 1.2: at build time, no CSP code (including firmware) must be stored in worker nodes. At run time, the worker nodes must only execute customer code. (The numerous pods executed by the CSP in PaaS offers must not run in a confidential worker node).



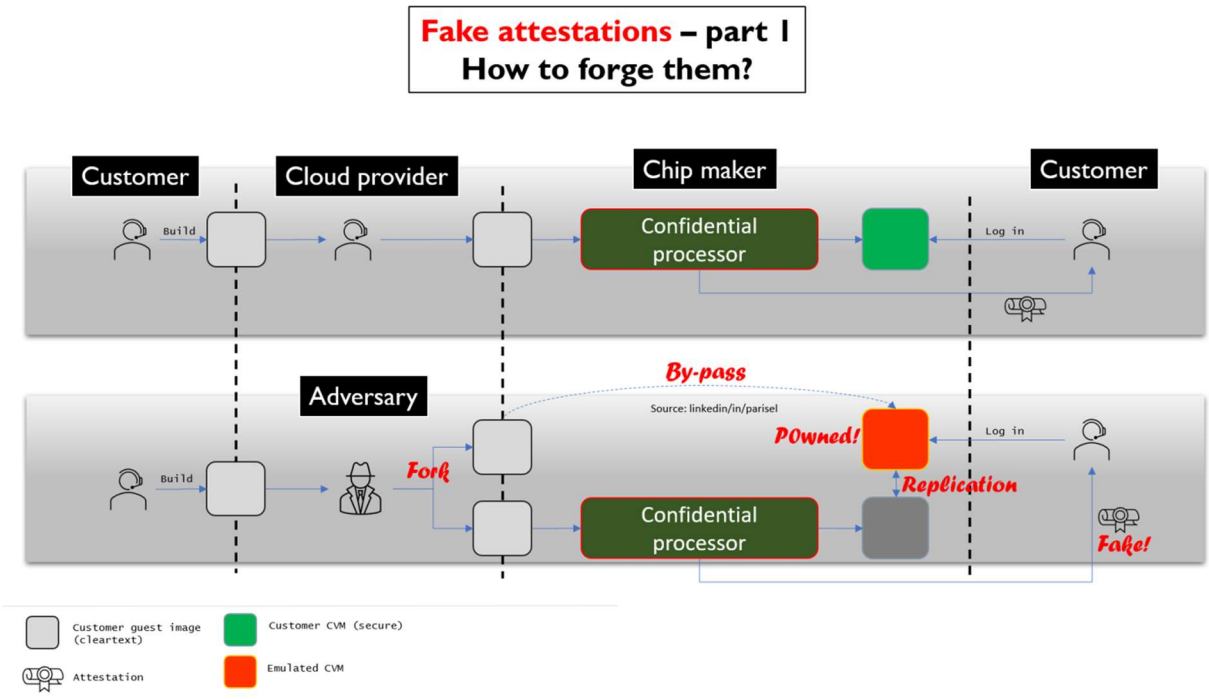
All CSP managed pods must be removed from a confidential PaaS offer. (AKS example)

Requirement 1.3: the memory and registers of worker nodes must only be readable and writable in the clear from within the CVM. (Memory and register must be encrypted and signed by the confidential processor).

REQUIREMENT #2: SECURE BOOT

Each time a CVM is spun up, customers must be offered absolute guarantees that the instance is theirs, that it has not been duplicated and that it is not being emulated out of band (i.e., outside a confidential processor).

Here is an illustration of CVM emulation and why it is dangerous:



Requirement 2.1: CVM images (including firmware) must be encrypted with a key pair not belonging to the CSP.

Requirement 2.2: The confidential processor and the customers are the only actors able to decrypt the CVM image. This decryption must happen outside of provider reach.

Encrypted images may be stored by customers in an online gallery managed by their CSP.

REQUIREMENT #3: HERMETIC SYSTEMS

To customers, CVMs must act like any "normal" VM: they have usual egress/ingress network connectivity, they can have external disks mounts, ...

To any other entity, they are **hermetic systems**, meaning two things:

Requirement 3.1: By design, no communication can be established from the outside of the CVM (except by the customer herself).

Requirement 3.2: By design, communications established from the CVM to the outside of customer environment mustn't leak any customer information.

Notice: this definition of a hermetic systems is somewhat stronger than AWS' own definition. For AWS, a hermetic system corresponds to requirement 3.1 with a focus on built-in noOps for CSP activities.

Part 2 - solution design

In this section, we discuss "a" solution among many possible ones. The purpose here is not to find "the" perfect design, but a design that would fill the bill.

Everybody is invited to challenge this design and to improve it wherever it makes sense, provided all hard requirements are met of course.

Confidential boundary

Requirement 1.1 (worker node is a CVM) is already implemented in a PaaS offer: Azure confidential Kubernetes Service.

For the IaaS, requirement 1.2 (no CSP code in CVM) is currently in preview in Azure (in-guest custom firmware).

For the PaaS, requirement 1.2 is not yet implemented to the best of my knowledge and it is prone to impact the shared responsibility model in a couple of ways:

- the CSP will not be able to deploy its own images, including the kubelet and CSP-operated pods (like the OMS agent or the Azure Policy agent, to name a few)
- instead, the CSP will maintain a list of supported OSes and kubelet versions
- **the most disruptive change** will be on how the CSP will monitor and manage the lifecycle of pods and worker nodes (see Hermiticism section below).

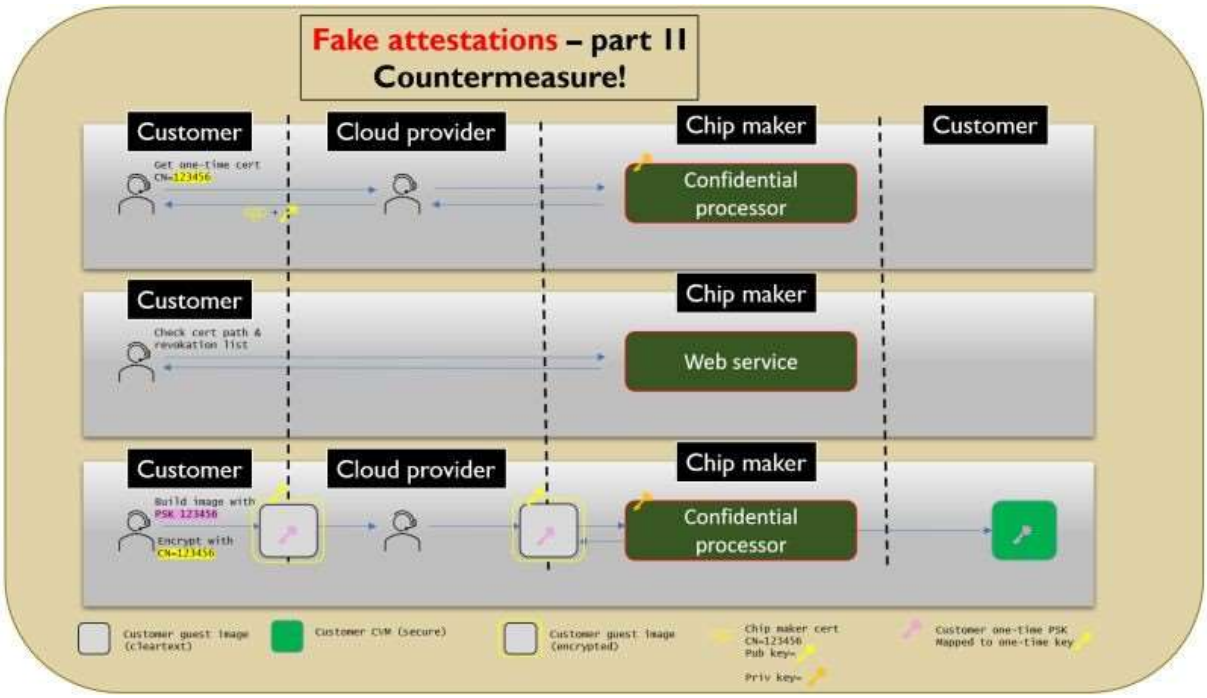
Requirement 1.3 (protected memory and registers) is already implemented in AMD-SNP chips.

Secure boot

Requirements 2.1 (on premises images encryption) and 2.2 (CSP-proof decryption) are currently not supported by chipmakers for IaaS, hence not for PaaS either.

I have already shared a possible solution:

- leverage AMD processors to run autonomous (noOps) agents for uploading and booting customer-encrypted images
- use a PKI with root CA hosted by the chipmaker to ensure end-to-end encryption, that is from on premises to the confidential processor (to prevent man in the middle)



Preventing MITM and CVM emulation

HERMETIC SYSTEMS DESIGN

This remains by far the most complex and innovative topic to tackle. Globally it amounts to a **guest-host secure communication problem**. Let's decompose this problem into its necessary subcomponents:

1. Control flow: master nodes must pass orders to worker nodes (in Kubernetes, this means control plane to kubelet communication)
2. Results logging: master nodes must know the outcome of the orders transmitted through the control flow
3. Events notification: worker nodes must have a way to notify the control plane of some events unrelated to the control flow (e.g.: a pod has crashed, an imaged couldn't be retrieved from the registry because the disk is full, etc.)

You may have noticed with have excluded all pod/worker node/kubelet to kube server API communication. Such flows won't be possible anymore, and it's quite a breaking change.

CONTROL FLOW DESIGN

To ensure perfect air-gapped isolation between the data and control plane, several concepts must be introduced:

- Worker nodes always fetch information from the control plane, never the opposite;
- To fetch information, worker nodes make an hypercall (to the hypervisor)
- Since the hypervisor is not to be trusted, it only sees encrypted data. But it is configured to forward the trap to a new in-guess entity named the Executive ;
- When triggered, the Executive performs APIC emulation on behalf of the worker node: it makes the necessary hypercalls to the hypervisor on behalf of the worker node to fetch actual control plane data.
- Before forwarding control data to the worker node, the Executive runs a critical routine called the Servicer (more of that in a moment)
- To make sure worker node calls are properly trapped by the Executive, the former is set to run at privilege level 3 (VMPL3) and the latter at privilege level 0 (VMPL0).

The good news is that all the above sequence is already supported by AMD-SNP technology.

The bad news is that two key ingredients must be designed: the Executive and the Servicer.

The Executive

The Executive is a new micro operating system hosted in the CVM, which acts as a proxy between the worker node and the control plane. As the custodian of hermiticism, *the Executive ensures the communication channel is both safe from intrusion and safe from leakage.*

The first outstanding feature of the Executive is that **it must be provably secure** under a CTL model checking (see for, example, http://www.cs.technion.ac.il/users/orna/FMCO05-survey.pdf)

To that end, the Executive must have the smallest footprint as possible. I believe that, given the number of operations required for operating a worker node, such a criterion will be easy to meet.

The second outstanding feature of the Executive is the unique way it handles I/Os: recall that the only I/O's it has to deal with are control flow orders. They can be normalized to follow a very simple syntax based on only three terms: a VERB, a NOUN and a UID.

For example, download image "ubuntu:20.04" from the registry can be translated as [VERB=4506, NOUN="ubuntu:20.04", UID=0x5f6b39201]

Here, verb 4506 corresponds to a registry pull. It is highly recommended that the control plane uses the same UID for idempotent requests, because they are cached in Executive memory for a reason that will become obvious in the log results section.

If the syntax is correct, the Executive runs the Servicer routine for further analysis.

The Servicer

The role of the servicer is to maintain its own copy of the state of the worker node. All orders issued by the control plane to the worker nodes involve a graph transition between valid states. Upon receiving a VERB/NOUN order, the Servicer checks that the resulting state is valid, and if so updates its internal representation of the worker node state. It then returns controls to the Executive that informs the worker node to perform the order.

To avoid the Servicer and the worker node be out of sync, an UPLINK from the worker node to the Servicer must be done at regular intervals. Details of the UPLINK won't be discussed here because they are not on the critical path.

RESULTS LOGGING

When the worker node executes an order transmitted by the Executive, the outcome must be communicated back to the control plane.

To make it 100% leakage free, the executive uses APIC emulation and the following straightforward syntax to convey the outcome to the control plane: [STATUSCODE, UID]

The UID plays a crucially important role here:

- It lets the control plane correlates the status code with the order, to put it in context;
- It prevents any leak, because the Executive polls its internal cache to check pre-existence of the UID before passing on the information to the control plane.

Events notifications

When an event is fired by the worker node, it must be channeled all the way to the control plane.

How to make it so that zero information is leaked? We don't have a UID here, because the event is not directly correlated to a control flow request.

One answer is to define another routine in the Executive: **the doorbell routine**. The doorbell simply converts events received from the worker node into messages of type [EVENTSTATUS].

When the control plane receives such message, it must then poll the worker node depending on the nature of the event: say the event is a pod crash [EVENTSTATUS=1008]. To determine exactly which pod has crashed, and assuming 45 pods are running on the worker node, the control plane issues 45 requests of type [VERB, NOUN, UID]. 44 of them will return an OK code, and one will return a KO. The UID of this request will let the control plane pinpoint the only pod which has crashed.

Last point I would like to share: the origins of the Executive and the Servicer. I think the problem at hand is quite similar with the kind of challenges NASA engineers had to meet when they launch the Apollo shuttle to the moon. They had to run a low footprint, highly optimized and resilient piece of software called the AGC (Apollo Guidance Computer).

One of the AGC main responsibility was to keep the state vector of the spacecraft constantly up to date thanks to a routine called the Servicer (similarly, we have to keep the state of the worker node accurate at all times). The AGC's operating system was called... the Executive. It accepted inputs of the form: VERB/NOUN. No need to reinvent the wheel!

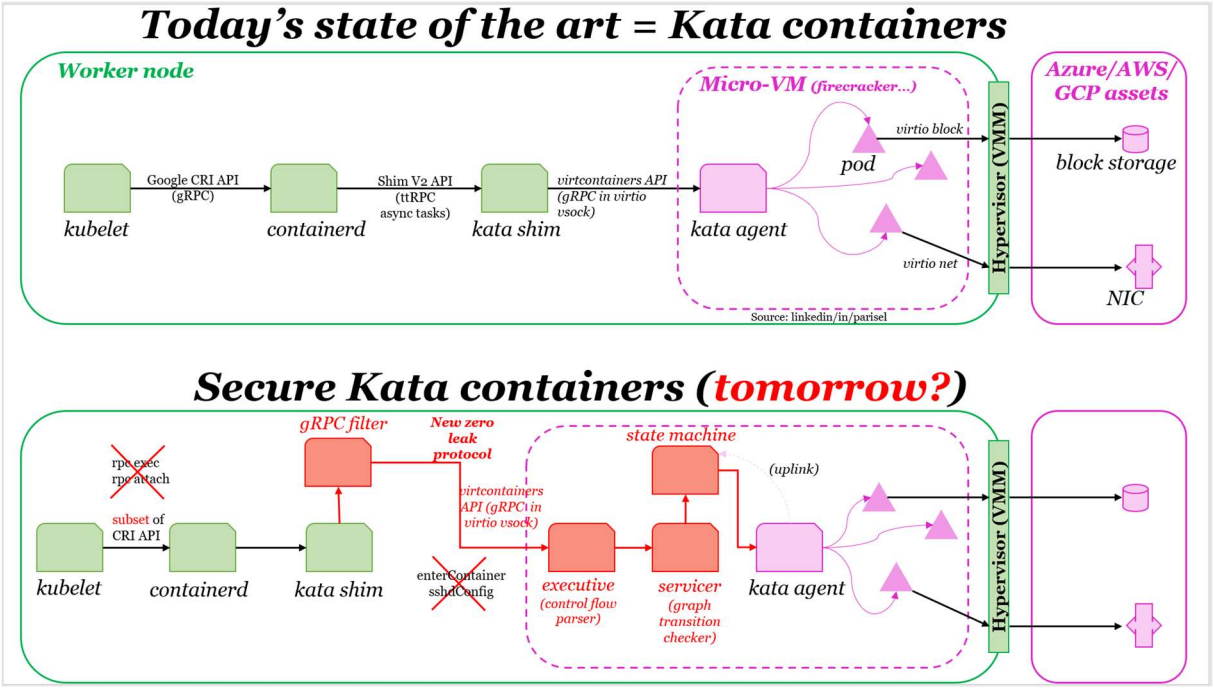
PART 3 – IMPACT ON CURRENT PAAS INITIATIVES

Accounting for hard customer requirements will have deep consequences on two main open-source projects:

- **Kata containers**, which aims at wrapping containers into micro-VMs, including Kubernetes Pods.
- **Confidential containers**, which aims at wrapping containers into a Trusted Execution Environment (TEE)

KATA CONTAINERS

Kata containers provides state-of-the-art pods isolation. But there is still a long way to go to reach multi-tenant security: for that, Kata containers must be turned into truly hermetic systems.



Kata containers before (above) and after (below) the hard requirements are accounted for

This transformation implies a hardening of both sides of the worker nodes' trust boundary:

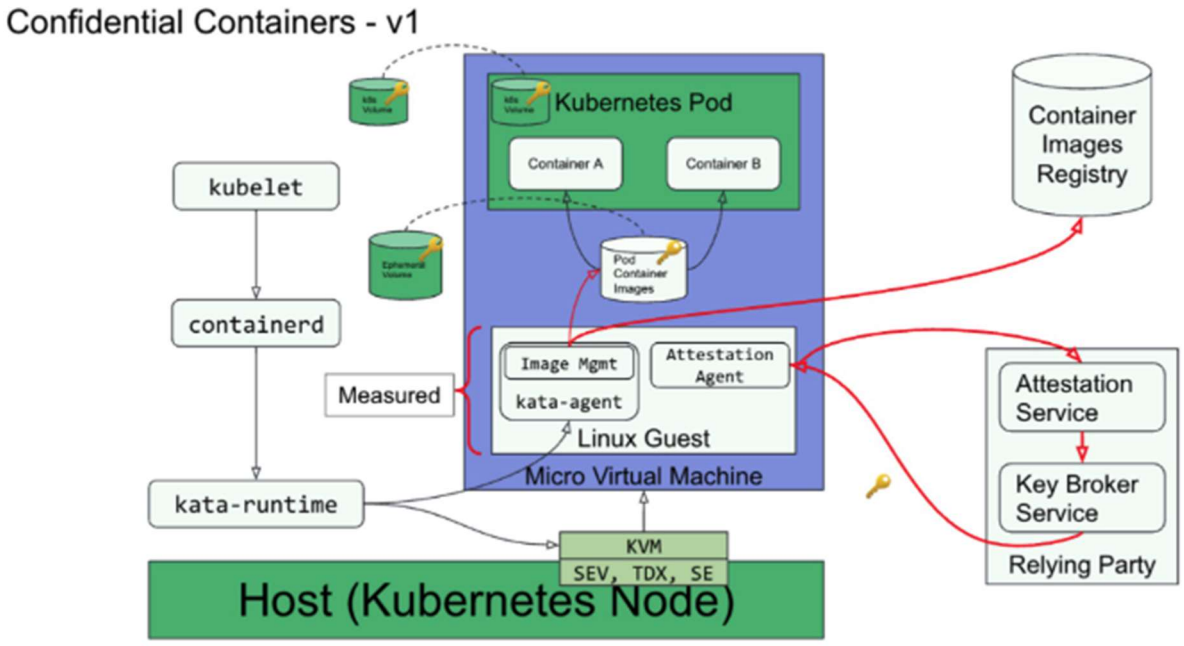
- on the provider end, the virtcontainers API, which is in charge of issuing control commands from the provider environment to the customer environment, must be revamped to issue zero-leak and innocuous commands

- on the customer end, the kata-agent, which takes direct orders from virtcontainers, must be firewalled by a provably secure control flow parser: the Executive and its ancillary Servicer routine. The Executive sanitizes virtcontainers data to make sure they adhere to hermetic systems requirements. The Servicer verifies that orders received from the provider mute the pods into an expected and compliant state.

CONFIDENTIAL CONTAINERS (COCO)

Confidential Containers is a project managed by the Cloud Native Computing Foundation (CNCF). It can be seen as a dedicated extension to Kata Containers.

COCO's V1 architecture for AMD SNP is as follows:



raw.githubusercontent.com/confidential-containers/documentation/main/images/COCO_ccv1_TEE.jpg

The soundness of this design relies on the assumption that all our requirements, especially requirements 2.1 and 2.2, have been applied to the micro-VM (in blue) itself. We have seen that, unfortunately, this is not currently possible.

Conclusion

I hope to have helped the community of potential Confidential PaaS customers get a clearer understanding of what is at stake, and how they could get more involved into the target design (since they will be most concerned when it is generally available!)

I also hope this whitepaper help chipmakers and providers alike have an idea of what level of security is expected from a production grade solution, so they do not waste time spending efforts in stillborn solutions.

Some requirements will have deep consequences on the architecture of container orchestrators and schedulers, so the sooner they are being addressed, the best. It is also an opportunity to make some of such orchestrators truly multi-tenant, but this is another topic!

Afterword: attestations

What happened to attestations? In this whitepaper, attestations have been replaced by images encryption (requirements 2.1 and 2.2).

Customer-controlled encryption provides a more secure way to prevent any risk of CVM emulation and/or man-in-the-middle than Provider-controlled attestations.