Building Trust in Cloud Computing – Isolation in Container Based Virtualisation

Ibrahim Alobaidan
Department of computer science
Liverpool John Moores University
Liverpool, UK
i.m.alobaidan@2012.ljmu.ac.uk

Michael Mackay
Department of computer science
Liverpool John Moores University
Liverpool, UK
M.I.Mackay @ljmu.ac.uk

Nathan Shone
Department of computer science
Liverpool John Moores University
Liverpool, UK
N.Shone @ljmu.ac.uk

Abstract— Cloud computing is now a mature technology that provides a wide variety of services. However, a challenging issue that remains for many users is choosing the best cloud service for a specific application and in many cases, one of the key factors to consider is security and trust. For example, ensuring data privacy is still a main factor in building trust relationships between cloud service providers and cloud users. In this paper, we propose a security system to address the weak isolation in container-based virtualisation that is based on shared kernel OS and system components. We address the isolation issue in containers through the addition of a Role Based Access Control model and the provision of strict data protection and security.

Keywords-Cloud computing; Container isolation; RBAC.

I. INTRODUCTION

Adding new resources and services in a highly scalbe shared tenancy environment is a key feature of cloud computing [1], which has now become a ubiquitous technology in all areas of computing. However, one constant issue faced in cloud computing is that of data security, which has long limited the adoption of the approach in certain areas. It has always been the responsibility of the cloud user to ensure that the selected cloud enviroment provides a reliable data privacy, integrity and trust model through its data storage security framework. However, there has also always been a corresponding trade-off to be made by the Cloud Service Provider (CSP) in the need for security versus the performance overheads this introduces on the system. One example of this trade-off is in the move away from traditional full-stack virtualization towards Containers. Performance, isolation, security, networking, and storage are five factors that are commonly used to compare between Virtual Machines (VM) and Containers [2]. In the Virtual Machines (VM) each guest VM has its own operating system and kernel built on top of the virtualized hardware, while container-based systems share the kernel OS and virtualize the environment above it.

Containers provide better performance compared with Virtual Machines because of this reduced overhead compared to full virtualization but may provide less isolation, and therefore be less trustworthy, as a result. The isolation aspect is increasingly important in cloud computing to ensure the users' data privacy and integrity. Due to shared tenancy, which is a central feature of virtualised infrastructures, providers need to enforce strong mechanisms to ensure that virtual services running on the same physical server do not interfere with or impede each other, and that users cannot break out of their allocated virtual machine (VM). As a result, in this paper we propose a system to improve the isolation of users in container-based virtualisation with the aim of improving privacy of these services and therefore the trustworthiness of the whole infrastructure.

In the remainder of this paper, we first describe some of the related work on trust in cloud computing, an evaluation of hypervisor vs container isolation, and an overview of container security mechanisms in section 2. In Section 3, we then present our proposed approach that would help to build trust relationships between CSPs and users by solving the isolation issue in container-based virtualisation. Next, we present our system architecture that focuses on provider a Docker plugin using Role Based Access Control (RBAC) in Section 4. We briefly present the current implementation of our proposed system in Section 5 and finally, we conclude in Section 6.

II. RELATED WORK

In Cloud Computing the cloud service provider (CSP) is responsible for providing a trusted computing platform to guarantee privacy and security for the users [3]. This has been an active research area since the inception of Cloud Computing and many works in academia and industry have aimed to address this issue. We will first discuss this in the context of full stack virtualisation before analysing the changed introduced with containers.

A. Cloud Security

CSPs typically deploy strong security mechanisms to protect their infrastructure and by default use, encryption to secure the remote connection to the user, but limited external accountability has led to a lack of trust in the safety of data and services entrusted to the Cloud by users. A few critical issues for building trust in cloud computing were identified by The Cloud Security Alliance [4] where different levels of security are required in public and private clouds. Data integrity and confidentially and building trust between providers and users were the critical security issues identified in every case. Another study [5]reported that trust was a vital component to be combined into cloud systems, and security is one of the key factors that many users and providers are often concerned about.

Fundamentally, the fact that clouds use a remotely administered shared virtual infrastructure often requires a higher level of trust to exist between the CSP and the cloud user. Therefore, having authorization as a form of security

 ISBN: 978-1-61208-703-0

measure is not only useful, but also highly necessary in order for trust to exist between these two parties. For example, the provider could use some approaches to limiting system access to authorized users, such as through Role Based Access Control (RBAC). Another mechanism to build up trust is through a formal Trusted Computing Platform (TCP), which can be used to ensure that only the customers can access their data and the administrator has no access to any of the customer's secured data and cannot damage its contents. This also provides assurances that user's computations are running on a trusted platform by validating whether the VM is operating on a trusted implementation or not.

B. *Container vs hypevisor based isolation*

Container-based virtualisation differs from that in VM based virtualisation in that the latter is applied comprehensively down to the hardware, whereas containers use shared Operating System components. As such, the hypervisor approach provides inclusively complete isolation of the user applications and services, but incurs a comparatively large performance overhead through the additional management. In contrast, containers have become very popular due to their improved performance and relatively low overheads, but may offer less isolation to users as a result. Some work has been done to measure the difference in isolation between containers and hypervisors.

A study by IBM provided a comparison of isolation in Linux containers and full Virtual Machines (VM) [6] where the goal was to evaluate efficient methods of resource control using the two different methodologies. The level of resource isolation was evaluated between traditional VMs and Linux containers when handling various workloads that were particularly CPU, memory, and network intensive. The results concluded that container-based technologies did offer reduced isolation in some cases but ultimately provided a superior alternative for cloud-based solutions because of their better performance and easier deployment.

The authors in [7] also present results from testing the isolation properties of VMWare, Xen, and OpenVZ through various performance stress tests. Here, both VMware and Xen operated perfectly in isolating the VMs in all the tests with little resource degradation. However, OpenVZ containers displayed a significant impact in comparison, particularly where no resource-sharing controls are applied. The results showed that the networking tests resulted in the biggest impact in container isolation and therefore provided the weakest isolation between virtual instances. This could be a result of the network-oriented measurements using SPECWeb, which were the benchmarking tools used. There was also some impact on the disk intensive tests, especially given the limited load the test introduced in the normal servers. However, a significant shortcoming of the testing was that it only considered a single type of container virtualisation For example; Docker provides a much more lightweight environment then OpenVZ and is still the default solution for this type of virtualisation.

To evaluate the isolation performance of Docker in this context, we replicated the test above to evaluate the performance impact on a HTTP server in one container while the other ran the above-mentioned isolation benchmarking tests [2]. In this case we used Httperf for our testing because it is a more open and flexible approach. In this test, we created two Raspberry PI hosts connected via a local Ethernet connection running at 1GBps, one as a client and the other as a server; both are running Raspbian OS and Docker. The client is running Httperf in a single container while the server is configured with two containers, one with an Apache2 webserver and another with the isolation-benchmarking suite. The isolation benchmark tests were compared to the Httperf-only test to highlight any discrepancies. In particular, the fork bomb intensive results showed significant degradation in the presence of the stress tests and demonstrates that Docker containers are also susceptible to the same weaker isolation and performance.

C. *Container security features*

When reviewing Docker security, the Kernel namespace, control groups and the Docker daemon itself are the three major areas to consider. This is because Docker shares access to the underlying Linux Kernel between the host and the containers and therefore the responsibility of enforcing isolation is also shared between the host and the platform. Figure 1 shows the location of the main Dockers security features.

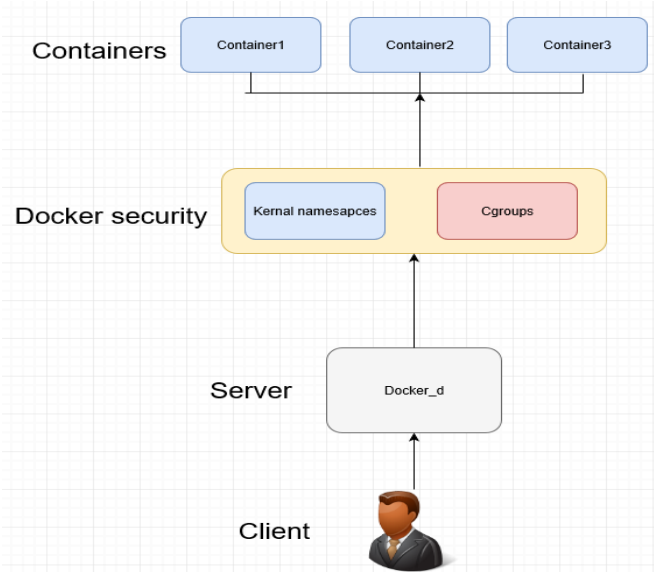

Figure 1. Kernel Namespaces and Cgroups

The Linux Kernel has Namespaces features, which is a fundamental aspect of containers on Linux [8]. Layer isolation is provided by these namespaces, which ensure that Docker users can only access particular containers. Docker creates these namespaces when the container is started, which then isolates processes running within the container from other containers and the host [2]. Each container has a separate process ID (PID), network artefacts (e.g. routing table, iptables and loopback Interface), and Inter-Process Communication (IPC) mechanisms namely semaphores, message queues and shared memory segments. Each

container also has its own mountpoint, which is provided by the mnt namespace. Finally, hostnames for different containers could be supported by the Unix Time Sharing (UTS) namespace. Cgroups also provide many useful metrics for container isolation [2]. Access to memory, CPU, disk I/O and other system resources can be equally distributed on the host, which aims to prevent a container from crashing the system by exhausting its resources.

However, the focal point of all communication to and from containers is the Docker daemon itself [9]. This program runs on the host machine and provides a central point of interaction between the system and the containers. The users do not directly interact with the Docker daemon, instead this is done through the Docker client, which provides access to the daemon through sockets or a REST API.

III. PROPOSED APPROACH

Given this reliance on the underlying Linux mechanisms in Container-based virtualisation, and the limitations in isolation this introduces, this paper proposes the development of an enhanced security system to address the issue by using Role Based Access Control (RBAC). RBAC policies will be configured for each container using an authorisation plugin running within the Docker daemon with the not only to isolate each container from the other and the underlying systems but also to isolate user resources in the same container from each other.

In our proposed system, the containers trust the host to make and enforce authorisation decisions as an extension of the existing system without the need to introduce additional components in the architecture. The plugin will be registered as part of the Docker daemon, which resides on the host and the containers have no access to this. Therefore, access can be granted only to resources when authorised by the plugin. The Docker daemon obtains this request through the CLI or via the Engine API as before, which passes the request to the authorisation plugin. The authorisation plugin will obtain the user request data and provide a decision according to the user policy. Figure 2 shows a typical authorisation scenario for a user request.

A user request should contain information on the username, policy, container ID, the object path, and action. Then, the authorisation plugin will make a decision whether to accept or deny the user request. For example, user Bob is part of the HR user group. Bob wants to access the employee database that is stored in a HR container that has the ID 495ad09fc530. A typical request in this case would include the following information:

Subject: = "Bob" // the user that wants to access an employee database.
Object: = "495ad09fc530" // the container that is going to be accessed.
Path: = "/H/employee-database" // the path for the resources within the containers that is going to be accessed.
Action: = "read" // the action that the Bob performs on the employee database.

The benefits of this centralised approach are that it reduces complexity and resource usage, as only one security mechanism will be required per host. Further, due to the centralised nature of data stored in cloud infrastructures, our proposed design would minimise data leakage and improve monitoring. Developers can already add access control in the Docker daemon through a number of existing authorisation plugins. However, this authorisation is currently performed on a very coarse level and does not support the centralised management of this process across the entire cloud infrastructure.

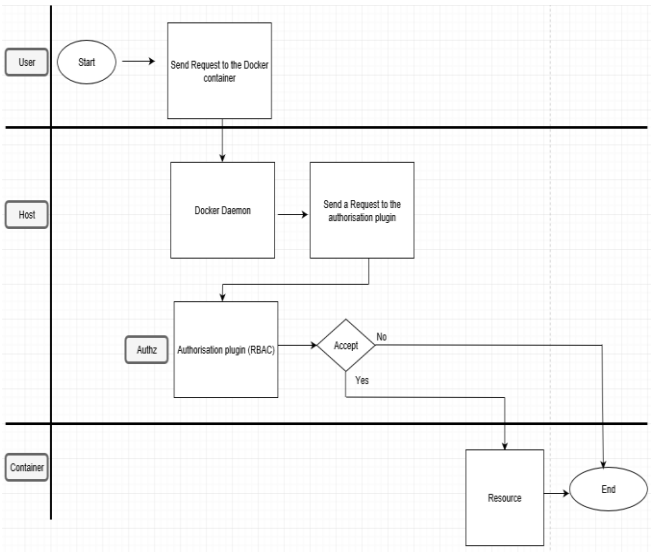

Figure 2. Authorisation Scenario

The system we propose includes the ability to allow or restrict access to specific containers, or the resources contained within those containers on a per-user basis using RBAC. The RBAC model has been the standard authorisation approach for more than two decades [10]. However, RBAC has been deemed unsuitable for further use, according to the continuously evolving access control requirements of emerging computing paradigms. These RBAC drawbacks have been addressed by Attribute Based Access Control (ABAC), which has appeared as a powerful alternative to RBAC. As such, it is necessary to explain why we have not adopted this approach in our work. In our analysis, we can determine that each container image will be created in advance of deployment and so an appropriate set of policies will be developed as part of this process. Then, whenever an image is deployed in a container, these policies can simply be imported into the authorisation plugin in the host. This makes RBAC more scalable in situations where large numbers of containers are expected to be deployed and more performant with fewer overheads in resource-constrained environments.

IV. SYSTEM DESIGN

We have created a first design of our security system based on the approach outlined above. We first describe the

 ISBN: 978-1-61208-703-0

system architecture before focussing specifically on the design of the plugin.

A. *System architecture:*

In the cloud datacentre, each Docker host is configured with the authorisation plugin such that any container that is deployed on is subject to the same process. Now, users who utilise the datacentre can specify user authorisation policies and associate them with any container images that they configure on the system. This will provide a consistent model of access that determines which users can access which resources within that specific image. Thereafter, any time an image is deployed into a container on any host with the datacentre, the associated policies will be deployed into the authorisation plugin alongside the image to control access, as shown in figure 3. This system provides a scalable point of control, such that the user roles and access can be administered centrally and dynamically applied with each update. Once a container is removed, the associated policies are also simply deleted from the authorisation plugin on the host.

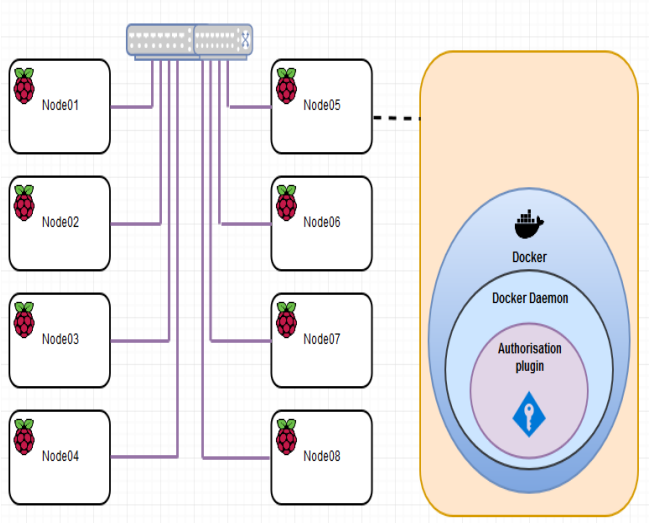

Figure 3. System Architecture

In this approach, the authorisation plugin in the host does not require any knowledge of the resources inside the container, but the administrator of the account can control which users (or roles) can access which data, files or services. The advantages of this is that only one authorisation plugin has responsibility for each host, which may be running a number of containers from many users. Moreover, regardless of how the user resources are deployed in the data centre, the policies that control user access are consistent and controlled by the account administrator. Finally, the underlying CSP does not need to understand how these policies are configured to control access to resources, only that the mapping between the image and policy is maintained.

The users can then request access to specific applications within a Docker container, which is approved or denied utilising the RBAC-based authorisation plugin. Each user has a unique username that is used to access any host in the data centre and the RBAC policies governs what actions users can perform based on their assigned roles. The authorisation process is shown in Figure 4 below. As outlined in the previous section, the user accesses the deployed container via a client, which will provide access to the Docker daemon. The daemon will pass the request on to the authorisation plugin which will process the request against the current policy base. If a positive match is found then the request is granted or, as shown below, the request is denied if no matching policy is in place.

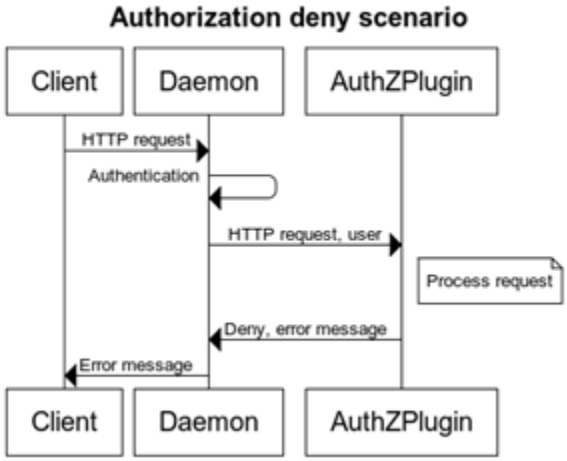

Figure 4. User Authentication via the plugin [11]

B. *Authorisation plugin*

The authorisation plugin runs directly on the Docker framework and makes use of the intrinsic plugin support offered by the daemon. The authorisation plugin is registered as part of the Docker daemon at start-up and contains a user policy file, which allows the administrator to set specific permissions for the users. For example, a container might have three objects groups that can be labelled objectgroup1, which starts with /H in the file system, objectgroup2, which stars with /W and objectgroup3, which starts with /F. Now, user (Bob) belongs to usergroup1 that has some policies to access objectgroup1 resources within a container. A policy should be defined that ensures that usergroup1 has access to all resources that have paths that start with /H in the file system on the specific container. In this case, a typical policy for the system would be as follows:

P, /v1.38/usergroup1/container/id//H/start, POST
P, /v1.38/usergroup1/container/id//H/attach, POST

The policy file contains rules that are specified according to the following format. P is the policy type that is the first field in each line. This project has one policy type, which is P (policy_definition) that contain subject, object, path, action) but it is possible to add more than one policy type in the model such as P, P1and P2. For example:

[policy_definition]
P = subject, object, path, action
P1 = subject, object, action
P2 = object, action

 ISBN: 978-1-61208-703-0

The policy definition is matched by policy type so, for the following policy definition:

P,/v1.38/usergroup2/container/495ad09fc530//W/start, POST

P is the policy type and v1.38 is the Docker API version. The subject is usergroup2 and the object is the container that has ID 495ad09fc530. The path is /W and the action is start. All rules in the policy file should follow the Docker API references. For example, /containers/id/start, POST is to start a particular container. Request data from containers is provided by GET. Send data to server to stop, start or attach containers is provided by POST.

The plugin model consists of a *request definition*, *policy definition*, *role definition*, *policy effect* and *matchers*. Role definition is represented by the letter G in the trust model, which is based on the definition for RBAC role inheritance relations. Each user will have one or more roles in the predefined RBAC policy file. For example, the system has a role named Role1 that is related to usergroup1, which allows all users who are related to HR to access HR resources. If user Ibrahim is part of the HR user group then we can define the following policies:

P,/v1.38/Role1/container/495ad09fc530//H/start, POST
G, Ibrahim, Role1

In the first policy, the subject will allow all users who are part of Role1 to access all resources that begins with /H within the container that has ID 495ad09fc530. In the second policy we simply add the user Ibrahim to Role1 which means that he can access the resource. The action is set to read only here because in container virtualisation, users should not have permission to delete or edit the Docker image that contains all the user data. In practice, this can be overcome through the use of local caches that can be committed back to the image over time. However, this functionality goes beyond the scope of our work at this stage.

V. IMPLMENTATION

The trust architecture is designed to be run in a Cloud Data Centre (CDC) cluster, which may be comprised of a large cluster of servers. As such, the first stage of implementing our work was to build a realistic data center cluster by using Raspberry PI devices. This allows us to develop our solution in a realistic, scalable, and cost-effective environment. The Raspberry PI cluster is created using the MPI (Messaging Passing Interface) library for communication [12]. MPI is a communication mechanism used in parallel computing environments to allow clustered nodes to interact seamlessly. The Raspberry PI devices will communicate without username or password through configured SSH [13]. The three main capabilities provided by secure SSH are secure command-shell, secure file transfer and Port forwarding. Raspberry PI cluster has a master node that has IP addresses for all cluster nodes and one or more Docker hosts which can run containers as shown in figure 5. Each Docker host is configured with our authorisation plugin as part of the daemon, which has policies for each deployed container. All containers in the system should be accessed by users through the master node.

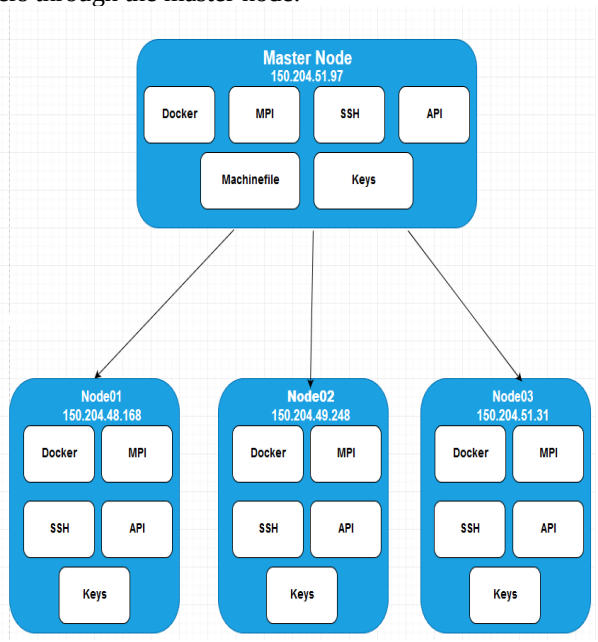

Figure 5. Trusted container PiCloud implementation

The authorization plugin is being created using the GO language because this was used by Google in the development of Docker and includes support for RBAC. GO has many libraries including one for RBAC and so we can easily extend the existing Docker plugin support framework to develop our system.

The trust plugin model is made up of the *request definition*, the *policy definition*, the *role definition*, the *policy effect* and *matchers*. As explained in the previous sections, the request definition has four factors, which are subject, object, path and action. Our implementation has three Roles (Role1, Role2 and Role3), which are related to Usergroup1, Usergroup2 and Usergroup3 respectively. The policy definitions are based on the four factors explained in the previous section, so a policy file in the authorisation plugin might typically comprise of the following policies:

A. *Usergroup1*

p, /v1.38/Role1/container/495ad09fc530//H/json, GET
p, /v1.38/ Role1/container/495ad09fc530//H/start, POST
p, /v1.38/ Role1/container/495ad09fc530//H/stop, POST
p, /v1.38/ Role1/container/495ad09fc530//H/attach, POST
g, usergroup1, Role1

B. *Usergroup2*

p, /v1.38/Role2/container/495ad09fc530//W/json, GET
p, /v1.38/ Role2/container/495ad09fc530//W/start, POST
p, /v1.38/ Role2/container/495ad09fc530//W/stop, POST
p, /v1.38/ Role2/container/495ad09fc530//W/attach, POST
g, usergroup2, Role2

 ISBN: 978-1-61208-703-0

C. *usergroup3*

```
p, /v1.38/Role3/container/495ad09fc530//F/json, GET
p, /v1.38/ Role3/container/495ad09fc530//F/start, POST
p, /v1.38/ Role3/container/495ad09fc530//F/stop, POST
p, /v1.38/ Role3/container/495ad09fc530//F/attach, POST
g,  usergroup3, Role3
```

The policy file above specifies that Usergroup1 can access all resources that start with /H, Usergroup2 can access all resources that start with /W, and usergroup3 can access all resources that start with /F within a single container that has ID 495ad09fc530. The role definition maps users to a specific usergroup to allow them to access the containers.

Finally, the matcher will compare the policy rule against the request based on the subject, object, path or action. Specifically, the matcher will compare r.sub (request definition subject) to p.sub (policy definition subject), r.obj (request definition object) to p.obj (policy definition object) and so on for the path and action. A match will be found only when there is an exact correlation between each of the request and policy parameters:

```
[matchers]
m = g(r.sub, p.sub) && r.path == p.path && r.obj == p.obj
&& r.act == p.act
```

VI. CONCLUSION

This paper has addressed the isolation issue in container-based virtualisation. We have developed a security system to enhance access control policies and provide data protection and security for users within each container. This security system can protect container guests from malicious users and improves the integrity of container data, applications and resources by adding a Role Based Access Control model.

In our system, the containers rely on the host to make the access decision through an authorisation plugin. This helps to address scalability issues because just one security model is required in the host instead of within each container. Moreover, each Docker image is defined along with a set of user groups and policies, which define how access should be granted to the resources it contains. Each time a new image is deployed in a container on the host, the authorisation plugin retrieves and applies the policy.

We are in the process of developing a proof of concept implementation of the authorisation plugin as part of our future work. Once completed, we will deploy and test it in our PiCloud CDC testbed to evaluate its suitability to provide fine-grained access control.

REFERENCES

[1] P. Sirohi and A. Agarwal, "Cloud computing data storage security framework relating to data integrity, privacy and trust," in *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, 2015, pp. 115-118.

[2] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 610-614.

[3] P. Sen, P. Saha, and S. Khatua, "A distributed approach towards trusted cloud computing platform," in *2015 Applications and Innovations in Mobile Computing (AIMoC)*, 2015, pp. 146-151.

[4] K. Hwang and D. Li, "Trusted Cloud Computing with Secure Resources and Data Coloring," *IEEE Internet Computing*, vol. 14, no. 5, pp. 14-22, 2010.

[5] Z. Shen and Q. Tong, "The security of cloud computing system enabled by trusted computing technology," in *2010 2nd International Conference on Signal Processing Systems*, 2010, vol. 2, pp. V2-11-V2-15.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171-172.

[7] J. N. Matthews *et al.*, "Quantifying the performance isolation properties of virtualization systems," in *Proceedings of the 2007 workshop on Experimental computer science*, 2007, p. 6: ACM.

[8] D. docs. (2019). *Isolate containers with a user namespace*. Available: https://docs.docker.com/engine/security/userns-remap/

[9] B. Kelley, J. J. Prevost, P. Rad, and A. Fatima, "Securing Cloud Containers Using Quantum Networking Channels," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 103-111.

[10] I. Alobaidan, M. Mackay, and P. Tso, "Build Trust in the Cloud Computing - Isolation in Container Based Virtualisation," in *2016 9th International Conference on Developments in eSystems Engineering (DeSE)*, 2016, pp. 143-148.

[11] D. docs. (2019). *Access authorization plugin*. Available: https://docs.docker.com/engine/extend/plugins_authorization/

[12] X. Wei, H. Li, and D. Li, "MPICH-G-DM: An Enhanced MPICH-G with Supporting Dynamic Job Migration," in *2009 Fourth ChinaGrid Annual Conference*, 2009, pp. 67-76.

[13] V. software. (2008). *An Overview of the Secure Shell (SSH)*. Available: https://www.vandyke.com/solutions/ssh_overview/ssh_overview.pdf

 ISBN: 978-1-61208-703-0