Déjà vu in OS Isolation

Sidhartha Agrawal @ University of British Columbia

Datacentres have two conflicting goals: packing as many client applications on as few machines as possible while maintaining the client's illusion that they are running on their separate machines. In principle, this is the same problem that virtual memory solved for limited physical memory. However, a richer vocabulary is needed for specifying isolation guarantees in a cloud deployment.

Currently, two threads in the application share everything: memory, OS state (e.g., file descriptors), hardware, etc. On the other end of the spectrum, two unrelated applications running on different machines share nothing. Between these two endpoints, we have coordinated applications with separate address spaces, but some shared information (e.g., a database), or unrelated applications running on the same hardware, but who wish complete isolation from one another (e.g., a container cloud deployment). Different isolation guarantees are achieved in a typical deployment using processes, containers, cgroups, and virtual machines. But the mechanism used also locks the application into the isolation level. For instance, a virtual machine lets the guest host do the page fault handling, but then it also virtualizes networking and file systems, which may not be what the application needs.

The recent introduction of Kata containers – a unikernel virtual machine masquerading as a container with higher safety guarantees – is a testament that we are solving one problem at a time rather than looking at the problem holistically.

Ideally, we should rethink all these solutions from the ground up. But before we get to the utopia, we need to build a *model* that identifies the resources needed by an application and how they can be used. These resources are physical memory, virtual memory, quota, bandwidth, ability to handle your exceptions, permission for a file, etc. Second, we need a *framework* to share and isolate resources amongst different protection domains by type. We present such a model and framework.

Resource	Read	Write	Exec	Use/Attach/Map
Physical Memory	X	X	X	X
Memory Range	X	X	X	X
Address Space	X	X	NA	X
Phy. Mem Quota	X	NA	NA	X
Privileged CPU state	X	X	NA	NA

Table 1: Resources

Model: A Protection Domain(PD) is a collection of resources used to run an application. We build the model by listing all resources(both physical and abstract) available to a protection domain and how they can be used.

Framework: Access to any resource in our systems is via a capability, an unforgeable token to an object, that retains meanings across address spaces. A capability has rights associated with it which dictate which operations can be done on the object.

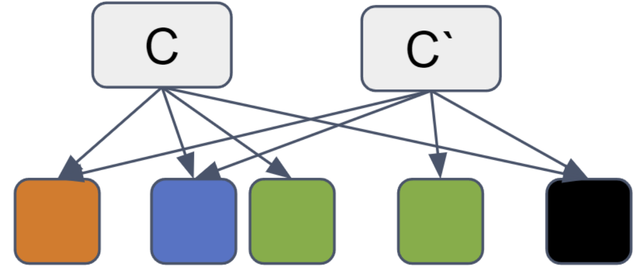

Figure 1: Sharing Capabilities

A PD with a capability to a resources can make newer capabilities to that resources with the same or fewer rights. This lets PDs share resources with varying level of access. Complex capabilities are made up of simpler capabilities. As shown in Fig. 1 the capability *C* is comprised of four low-level capabilities, and the capability *C'* shared three of these capabilities with *C*, but one(green) is different.

Resource Shared	Thread	TIS	TIF	[...]	Process	PIR
File System	Same	Same	New	??	Same	Same
CPU	New	New	New	??	New	New
Address Space	Yes	Yes - Partial	Same	??	New	New
Phy. Mem. Allocator	Same	Same	Same	??	Same	New
[...]	??	??	??	??	??	??

Table 2: Using the framework to create an arbitrary PD

Using our *framework* for selectively sharing resources, we can create arbitrary types of protection domains. Tab. 2 lists out the different classes of resources in the left column and what types of protection domains can be constructed by either using the existing PD's resources or making a new one. A center column is intentionally left blank to show that there yet exists unexplored types of protection domains between threads and processes. The empty bottom row is there to signify that this approach applies to other classes of resources as well. We show the upside of using the model by building three new PDs:

- **Threads with isolated stack(TIS)**: The threads have separate stack pages which are not accessible to other threads. This prevents cross stack corruption.
- **Threads with isolated file-handles(TIF)**: The threads have separate file systems. A thread doing stats reporting need not have handle to files with secrets.
- **Process with isolated physical resources(PIR)**: Processes have separate physical memory allocators. This prevents certain types of side channels.