Virtualization for OP-TEE

Volodymyr Babchuk
<volodymyr_babchuk@epam.com>

EPAM Systems

March 2021

Xen-troops

Bringing virtualization to automotive platforms

`https://github.com/xen-troops/`

Our accomplishments:

- Virtualization support in OP-TEE
- Multi-VM build system (`meta-xt`)
- XEN PV back-ends and front-ends for:
  - DRM (aka display)
  - Audio
  - Camera
- Support for Renesas RCAR Gen3 SoCs family in Xen
- Xen tailoring for customer needs

Agenda

Demo

Normal World $\leftrightarrow$ Secure World communication

Role of a hypervisor

Implementing VM contexts in OP-TEE

Future work

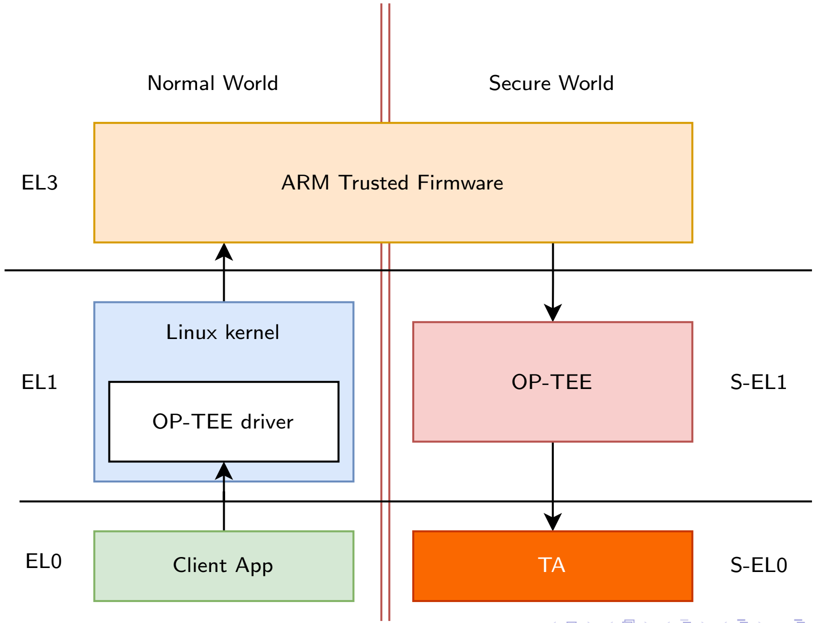
Questions

Quick Demo

Demo showing two virtual machines (VMs) that run `xtest` test package simultaneously.

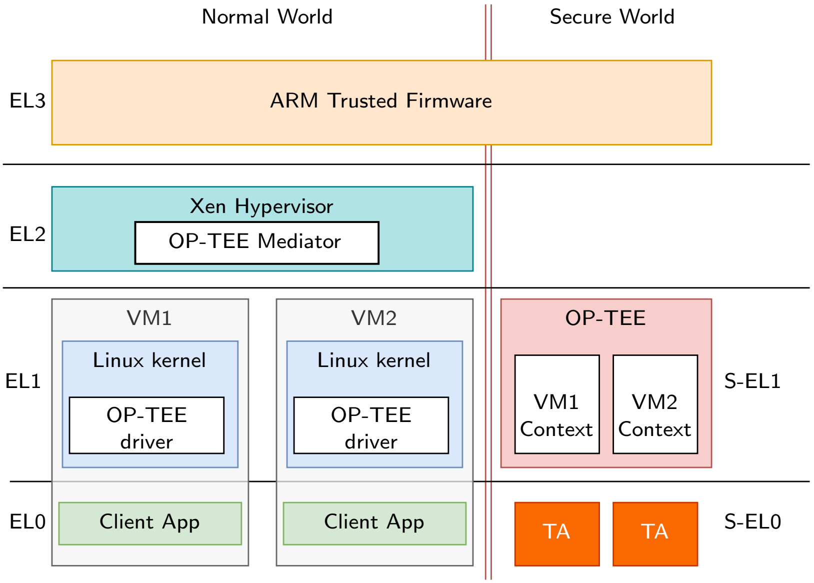
Demo is being run on Salvator-X board with Renesas RCAR H3 SoC.

Normal World ↔ Secure World communication



Normal World $\leftrightarrow$ Secure World communication

Now with virtualization



Role of a hypervisor

We can't allow VMs to call OP-TEE directly. There are multiple consideration concerning both virtualization mechanism and security:

- VM does not know physical addresses of own buffers
- OP-TEE needs to know when VM is created or destroyed
- OP-TEE needs to know which VM calls it
- We need to ensure that VM does not try to provide OP-TEE with memory reference to another VM's memory

Intermediate Physical Addresses

We all know the idea of virtual memory: Memory Management Unit (MMU) translates virtual memory addresses to real physical ones:

$$\begin{tikzcd}
\boxed{\text{VA}} \arrow[r, "\text{MMU}"] & \boxed{\text{PA}}
\end{tikzcd}$$

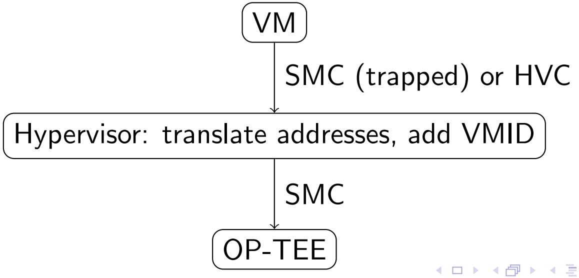
In most cases, when virtualization is used, we add a second stage of MMU translation and new type of address: Intermediate Physical Address (IPA):

$$\begin{tikzcd}
\boxed{\text{VA}} \arrow[r, "\text{MMU 1st stage}"] & \boxed{\text{IPA}} \arrow[r, "\text{MMU 2nd stage}"] & \boxed{\text{PA}}
\end{tikzcd}$$

Virtual machines manages translation tables for the first stage and believes that it is working with real physical memory. But in reality it lives in virtual address space. Hypervisor manages second stage translation which translates IPAs to PAs.
This approach allows hypervisor to manage address spaces of VMs in the same way as OS manages address spaces for processes.

Role of a hypervisor (cont.)

So, virtual machine sees only IPA and don't know real address of it's memory pages. On other hand, OP-TEE know nothing about IPAs and always expects real physical addresses to be passed from Normal World.
This is where hypervisor steps is. It traps all calls from VMs to OP-TEE and translates IPAs to PAs in all requests. In the meantime it also ensures that all memory pages in question belong to the VM. (Also, ensures that those memory pages will stay in memory for the whole duration).



Role of a hypervisor (cont.)
VM life cycle

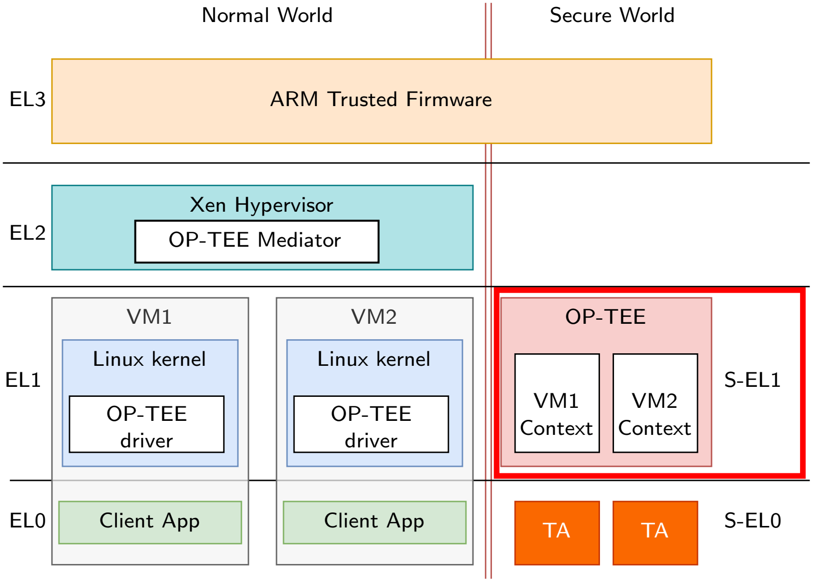
OP-TEE need to track life cycle of VMs. So it provides two special calls:

- `OPTEE_SMC_VM_CREATED(VMID)`
- `OPTEE_SMC_VM_DESTROYED(VMID)`

Hypervisor informs OP-TEE about VM creation or destruction by issuing above SMCs.

Implementing VM contexts in OP-TEE

Scope of this section



Implementing VM contexts in OP-TEE

The problem

We want to isolate state of VMs from each other. Things like shared memory buffers, TA contexts, mutexes, storage objects, . . . "belong" to a certain virtual machine.
Why? Two main considerations:

- Security (of course!)
- VM life cycle. VM can by destroyed at **any** moment. Imagine, that it is destroyed while it holds a mutex.

Ideal solution of course will be to run multiple OP-TEE instances - one per VM. But it is possible only on newest versions of ARMv8 architecture.

Implementing VM contexts in OP-TEE

Obvious approach

Obvious approach is to implement some sort of “virtual machine context”:

```
struct vm_context
{
  uint16_t vmid;
  void *mutexes;
  void *ta_sessions:
  ...
};
```

But it appears that this approach requires quite big changes in the codebase. Basically, every function, every piece of code that is used in client request handling should use this VM context. Almost all global variables should be moved into this context as well. Some sort of quota mechanism needs to be implemented: we don’t want DoS caused by a rogue VM that tricked OP-TEE to use all free resources for itself.

Implementing VM contexts in OP-TEE

Obvious approach pros and cons

Pros:

- Easy to understand
- Explicit implementation

Cons:

- All new functionality should be written with respect to this VM context
- Requires additional quota mechanism
- Complex clean-up when removing VM context: OP-TEE can be preempted while holding a mutex and never scheduled back
- Requires big changes in the codebase

Implementing VM contexts in OP-TEE

More subtle approach

Observation 1: execution state of an application is completely defined by its memory contents (and CPU registers, okay).

Observation 2: OP-TEE is quite well written and there is a distinct borderline between its housekeeping code and code that provides actual Trusted Execution Environment.

Housekeeping code (we called this part "nexus") is responsible of low-level things: device drivers, memory management, threads handling, providing synchronization primitives and so on.

"TEE" part provides actual "business value": manages TAs, sessions, secure storage, handles syscalls from TAs, this kind of things.

Implementing VM contexts in OP-TEE

More subtle approach (cont.)

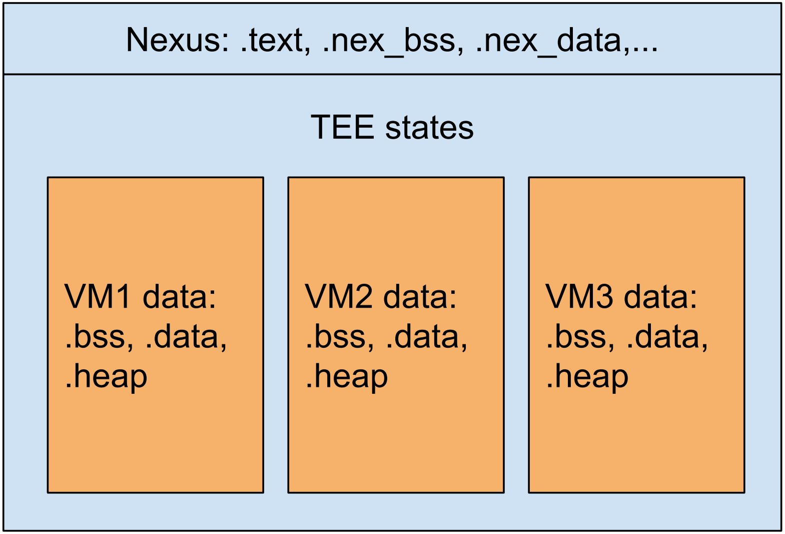
We wanted multiple OP-TEE instances. But what if we can have one “nexus” instance and multiple “TEE” instances?

We can split the whole OP-TEE memory into two parts. In the first part we will store nexus state and in the second - TEE state.

Because OP-TEE is running in virtual memory, it can map TEE memory for a required VM in a place where TEE code expects to find its data. This is somewhat similar to a memory “banking” used in old microntrollers, but done using MMU.

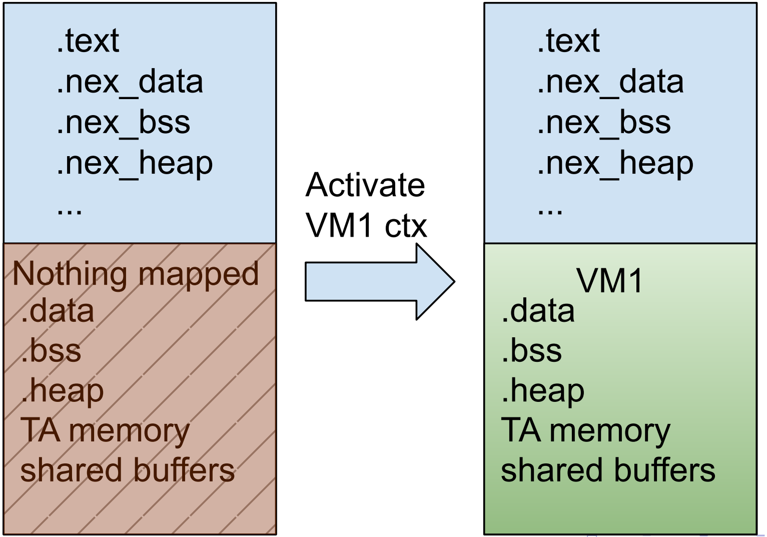
Implementing VM contexts in OP-TEE

More subtle approach (cont.)


Nexus: .text, .nex_bss, .nex_data,...
TEE states
VM1 data: .bss, .data, .heap
VM2 data: .bss, .data, .heap
VM3 data: .bss, .data, .heap

Implementing VM contexts in OP-TEE

More subtle approach (cont.)



Implementing VM contexts in OP-TEE

More subtle approach (cont.)

Pros:

- Perfect state isolation: only one VM state is accessible at a time
- Relatively small amount of changes in the code
- Easy clean-up on VM destruction: just “forget” about VM state
- Every VM gets equal share of memory

Cons:

- Predefined number of VMs (`CFG_VIRT_GUEST_COUNT` build option)
- Every VM gets equal share of memory. We can’t provide more memory to one VM and less memory to another.
- Harder to understand
- All nexus global variables should be explicitly moved to nexus memory sections

Future work

The most pressing thing:

- Hardware sharing. Things like HW accelerators, RPMB partitions and so on. (Actually, RPMB works just fine if every VM can provide own RPMB device)

Nice to have features:

- Dynamic number of VMs
- Non-equal VM resources allocation
- Support on more platform (currently tested only on QEMU, RCAR Gen3 and IMX8)

Thank you!