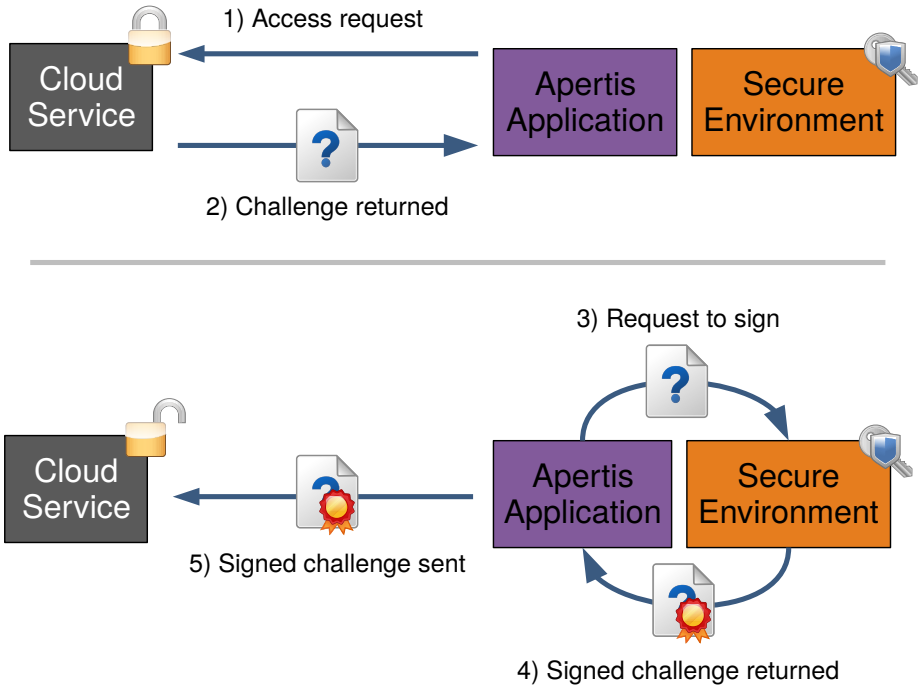
Integration of OP-TEE in Apertis

Contents

Some projects that wish to use Apertis have a requirement for strong security measures to be available in order to implement key system level functionality. A typical use case is enabling the decryption of protected content in such a way that doesn't allow the owner of the device doing the decryption to access the decryption keys. Another use for strong security is the protection of authentication keys. By shielding such keys within these strong security measures, it becomes much harder for the keys to be stolen and be used to impersonate the legitimate user.

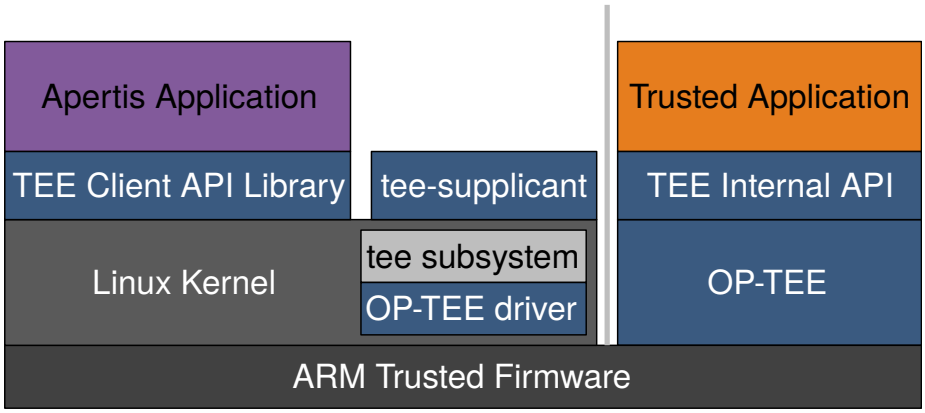


In the above example, when requesting access to the cloud service, the service returns a challenge response, which needs to be signed using asymmetric cryptography[1]. The Apertis application requests that functionality in the secure environment sign the challenge using a private key that it stores securely. The signed challenge is then returned to the cloud service, which checks the validity of the signature using the public key that it holds to authenticate the user. Such systems may additionally require the state of the system to be verified (typically by building a chain of trust[2]) before use of the secure keys is allowed, thus ensuring the device hasn't been altered in ways which may compromise protection of the keys.

Whilst a system could be architected to utilize a separate processor to perform such tasks, this significantly drives up system complexity and cost. Some platforms provide a mechanism to enable a secure, trusted environment or "Trusted Execution Environment[3]" (TEE) to be setup. A TEE runs on the application processor, but with mechanisms in place to isolate the code or data of the two running systems (the TEE and the main OS) from each other. ARM provides an implementation of such security mechanisms, known as ARM TrustZone[4], mainly on Cortex-A processors.

[1] https://en.wikipedia.org/wiki/Public-key_cryptography
[2] https://en.wikipedia.org/wiki/Chain_of_trust
[3] https://en.wikipedia.org/wiki/Trusted_execution_environment
[4] https://developer.arm.com/ip-products/security-ip/trustzone

System Architecture



A TEE exists as a separate environment running in parallel with the main operating system. At boot, both of these environments need to be loaded and initialized, this is achieved by running special boot firmware which enables the TrustZone security features and loads the required software elements. When enabled, a "secure monitor" runs in the highest privilege level provided by the processor. The secure monitor supports switching between the trusted and untrusted environments and enabling messages to be passed from one environment to the other. ARM provide a reference secure monitor as part of the ARM Trusted Firmware[5] (ATF) project. The ATF secure monitor provides an API to enable the development of trusted operating systems to run within the trusted environment, one such trusted OS is the open source OP-TEE[6]. OP-TEE provides a trusted environment which can run Trusted Applications (TAs), which are written against the TEE internal API.

As well as starting up a trusted OS in the trusted environment, ATF typically starts a standard OS such as Linux on the untrusted side, known as the rich operating system or "Rich Execution Environment" (REE), by running the firmware normally used for this OS. It is necessary for the OS to have drivers capable of interfacing with the secure monitor and that understands how to format messages for the trusted OS used on the trusted side. Linux contains a TEE subsystem[7] which provides a standardized way to communicate with TEE environments. The OP-TEE project have upstreamed a driver[8] to this subsystem to enable communications with the OP-TEE trusted environment.

OP-TEE relies on the REE to provide a number of remote services, such as file system access, as it does not have drivers for this functionality itself. The OP-

[5] https://github.com/ARM-software/arm-trusted-firmware
[6] https://www.op-tee.org/
[7] https://www.kernel.org/doc/Documentation/tee.txt
[8] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/tee/optee

TEE project provides a Linux user space supplicant daemon[9] which supplies the services required by the trusted environment. A library is also provided which implements a standardized mechanism, documented in the GlobalPlatform TEE Client API Specification v1.0[10], for communicating with the TEE. It is expected for this library to be used by applications needing to communicate with the TAs.

Boot Process

From a high level, the basic change required to the boot process is that the TEE needs to be setup before the REE. The factor missing from this description is security. In order for the TEE to be able to achieve it's stated goal, providing a secure environment, it is necessary for the boot process to be able to guarantee that at least the setup of the TEE has not been tampered with. Such guarantees are provided by enabling secure boot for the relevant platform.

The process used to perform a secure boot is dependent on the mechanisms provided by the platform which vary from vendor to vendor. Typically it requires the boot process to be locked down to boot from known storage (such as a specific flash device) and for the boot binaries to be signed so that they can be verified at boot. The keys used for verification are usually read-only and held in fuses within the SoC.

The signed binaries comprise a series of bootloaders which progressively bring up the system, each being able to perform a bit more of the process utilizing support enabled by earlier bootloaders. This series of bootloaders will load the secure monitor (known as `EL3 Runtime Software` in this context), OP-TEE (the `Secure-EL1 Payload`) and finally U-Boot (the `Non-trusted Firmware`), which loads Linux.

The ARMv8 architecture provides 4 privilege levels. The lowest privilege level, PL0, is used for executing user code under an OS or hypervisor. The next level, PL1, is used for running an OS like Linux, with PL2 above it available to run a hypervisor. The highest level PL3 is used for the secure monitor.

A more in-depth description of the boot process can be found in the OP-TEE documentation[11].

Trusted Applications

Trusted Applications (TAs) are applications that run within the trusted environment, on top of OP-TEE. Trusted Applications are used to provide the secured services and functionality that is needed in the platform. The TAs are identified by a UUID and are usually loaded from a file stored in the untrusted file system named after the UUID. In order to ensure the TAs haven't been tampered with they are signed. If the contents of the TA should remain protected, there are

[9] https://github.com/OP-TEE/optee_client
[10] https://globalplatform.org/specs-library/tee-client-api-specification/
[11] https://trustedfirmware-a.readthedocs.io/en/latest/design/firmware-design.html

options for storing it encrypted for further protection. Alternatively, if a TA is required before the tee-supplicant is running (and hence able to access the TA from the file system), TAs can also be built into the firmware as an early TA. A more in-depth description of TA implementation can be found in the OP-TEE documentation[12].

The OP-TEE project provides a number of TA examples[13].

Trusted Applications provide immense flexibility in the functionality that can be provided from the TEE environment. This flexibility is such that a proof of concept has been completed implementing a TPM 2.0 implementation[14] that can be used in OP-TEE[15].

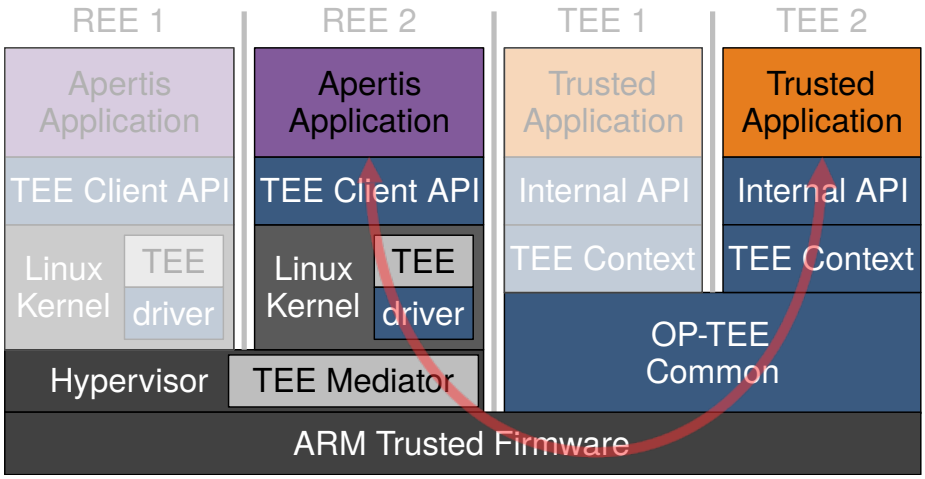
Virtualization Support

As the hypervisor and secure monitor each have a separate privilege level, it is possible for the TEE to co-exist with systems running a hypervisor. Whilst it is possible for the two to exist, a number of adaptions need to be made to allow communications to happen.

When running on a hypervisor, the guest OS uses intermediate physical addresses (IPAs) rather than physical memory addresses. These IPAs are then translated by the hypervisor to real physical addresses. The TEE concept was not developed with hypervisors in mind and the REE expects to pass the memory regions it uses for communicating with the TEE as physical addresses. However, unlike the TEE, the guest OS (acting as the REE) does not have access to the actual physical addresses, which will lead to miss-communication as to where data is stored. The hypervisor also needs to know that the contents of the used regions of physical memory can't be swapped out whilst communication between the TEE and REE is on going. Additionally something would need to keep track of which VM made the request so the response can be passed back to the right VM and handle situations such as the VM dying whilst the TEE was handling a request.

It is therefore necessary for both the TEE and hypervisor to be modified for things to function. Virtualization support has already been added to OP-TEE and experimental support[16] has been added to the Xen hypervisor running on an emulated ARMv8 system. The current approach modifies OP-TEE to provide a common TEE infrastructure with separate TEE contexts made available for each of the Virtual Machines (VMs) in which the trusted applications for each VM run.

[12] https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html
[13] https://github.com/linaro-swg/optee_examples
[14] https://github.com/Microsoft/ms-tpm-20-ref
[15] https://github.com/jbech-linaro/manifest/tree/ftpm
[16] https://optee.readthedocs.io/en/latest/architecture/virtualization.html



This works in conjunction with support from the hypervisor to provide memory mapping and to provide enumeration of the VMs so that the right context is used for each VM. The functionality added to the hypervisor is called the "TEE mediator". The advantage of this approach is that, whilst it requires changes the TEE and hypervisor, it keeps the API as seen by the guest OS and trusted applications the same, and thus existing applications and TAs do not need to be made aware of the virtualization used on the platform.

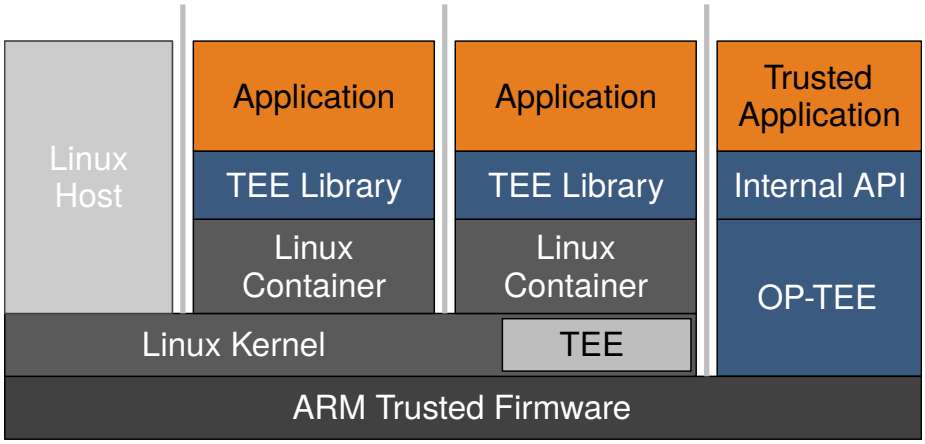
The virtualization support currently available in OP-TEE is configured at build time. A build supporting virtualization will currently not work in non-virtualized environments and vice versa. Thus separate builds will be required for each of these instances.

Access to hardware resources, such as cryptographic engines, has not been made virtualization aware and thus can't currently be handled safely between TEE contexts nor ensure that data from these resources isn't leaked between TEE contexts or to the REEs. Thus such hardware is not currently supported when virtualization is enabled[17] in OP-TEE.

Container Support

Containers, such as LXC containers, provide a significant level of isolation between the processes running in each container and the main host operating system, however each of these environments utilizes a shared kernel. Unless explicitly denied (such as by using LXC's cgroup support) each of the containers will have the ability to make calls to the TEE.

[17] https://optee.readthedocs.io/en/latest/architecture/virtualization.html?highlight=virtualization#sharing-hardware-resources-and-ptas



We expect that calls between the containers and TEE would work, however unlike when virtualization is utilized, the TEE will not provide a separate context for each of the REEs and as such it is likely that, depending on the trusted applications installed, it might be possible for some data to leak between the containers via the TEE.

More investigation work is required to understand the impact of this topology on the tee-supplicant. It is currently unclear whether it would be possible to have more than one instance of the tee-supplicant. Running the tee-supplicant on the host may be an option, though this may provide a way for containers to bypass their containment, by getting the supplicant to perform operations on or from the host. Additionally, the tee-supplicant can be extended with plugins[18]. For this to function, the tee-supplicant would realistically need to be in the container and thus a tee-supplicant would probably be needed by each container that needed one.

In the event that the TEE can not be directly utilized by containers it will be possible to utilize a TEE proxy as described later.

Advanced topology support

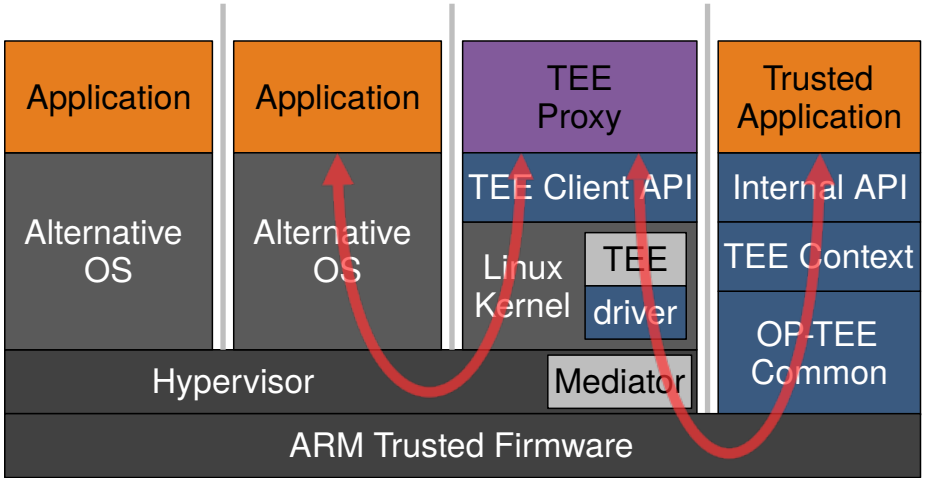
It is expected that product teams will want to utilize topologies that go beyond what is currently available in OP-TEE. Below we cover a few strategies that we envisage could be implemented.

Proxying TEE Access

It is expected that some systems will require non-Apertis guests to be run alongside Apertis on a hypervisor. These non-Apertis guests may also lack

[18] https://github.com/linaro-swg/optee_examples/commit/0607ed40746afe4cb8993149a6f275df648f7bad#diff-92cd10e6b8b068a931196d1d73a032543d5ca1a5bf445e27a1af74258254517cA

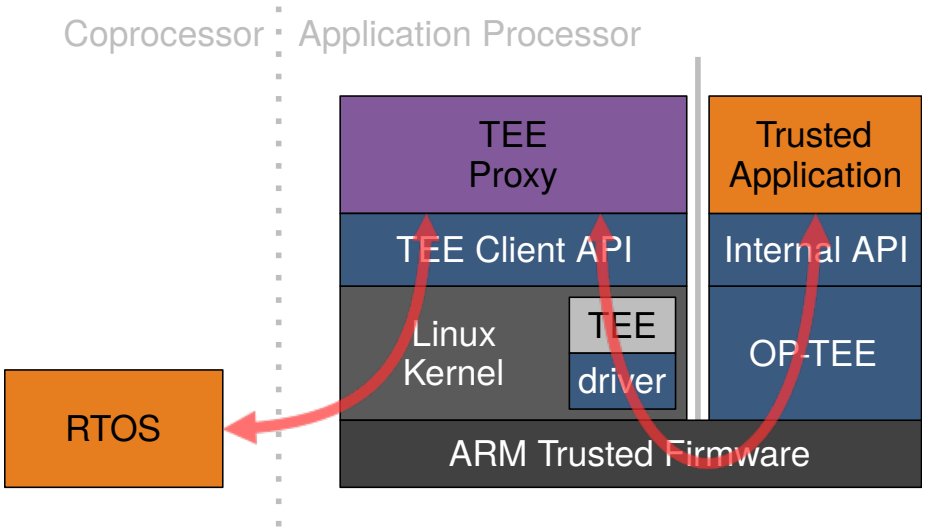
support for TEEs or there may be a desire for all the guest environments to be able to access a shared TEE environment. For these instances we would suggest the development of a "TEE proxy". The TEE proxy would be a daemon, which provides proxied access to TEE functionality.



The other virtualized environments would utilize a standard mechanism to communicate with the TEE proxy. In order for such a topology to be viable care would need to be taken to ensure that communications with the TEE proxy could be authenticated and secured. We recommend that a TCP based network connection be used to communicate with the TEE proxy. This would allow standard security measures that are widely implemented and understood (such as SSL) to be used for communications. This could be implemented over a virtual network link provided by the hypervisor. Additionally it would be necessary for the guest environment hosting the TEE proxy to be trusted as it will have access to the communications between the other VMs and the TEE.

It exact serialisation used over this network link could be D-Bus, though gRPC or other RPC mechanism may be a better fit.

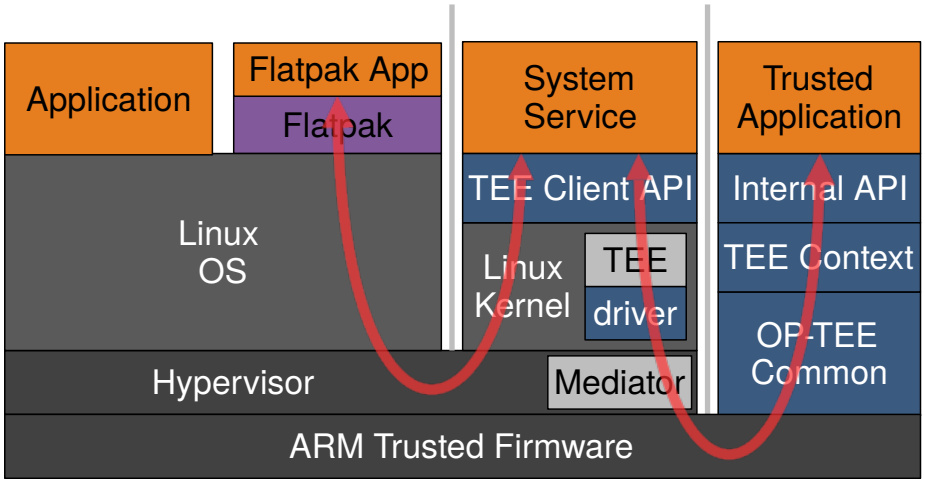
Such a TEE proxy could also be used in other instances, such as when access to TEE functionality is required by processes running on a separate core, such as a coprocessor, which will not have any access to the TEE environment.



As before, such a topology would require the TEE proxy to authenticate requests from the coprocessor, the communications mechanism used would need to be secure and the TEE proxy trusted. As the interface between co-processors and the rest of the system varies wildly, it is hard to predict exactly how communications would be structured. If the co-processor is connected via a serial link, then the Point-to-Point Protocol[19] (PPP) could be used to enable network connectivity. In other cases the TEE proxy may need to be customized to support communications with the co-processor.

A similar but slightly different use case would be where containerized applications are used, such as Flatpaks. In these instances we suggest that a system service exposing higher level functionality than that expected by a TEE proxy would be more appropriate.

[19] https://en.wikipedia.org/wiki/Point-to-Point_Protocol

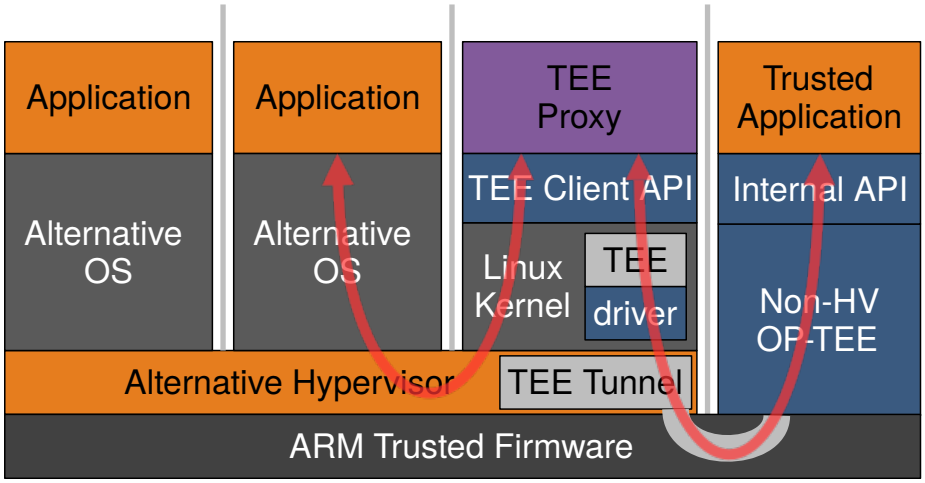


For example, rather than each Flatpak downloading encrypted downloads and passing the content of these to a TA to decrypt via a TEE proxy, a system service could be provided to download, validate and decrypt files for the Flatpak apps. This would be advantageous as it would reduce the number of hops that data would need to take from being downloaded to being available in the container decrypted, improving device efficency.

As with the previous topologies it would still require the system service to be trusted, the communication method used between the containerized application and the system service, such as a virtual network link, would need to be secure and communications with the system service authenticated. We expect such a component to be quite user case specific. As a result we do not expect the Apertis project to provide an example system service, though expect that the TEE proxy would serve as a good reference for the implementation of one.

Low impact hypervisor support for TEE

It is likely that system developers will want to utilize different hypervisor implementations that the one that gets integrated in Apertis. It is believed that implementing a TEE mediator will be a relatively complex task and system developers may wish to minimize the changes that are required in the hypervisor they are using. In instances where a single VM is expected to access the TEE, we envisage that cut down support could be added to the hypervisor, though more work is required to prove the accuracy of this plan and may be heavily effected by the exact choice of hypervisor.



This is expected to require the hypervisor to:

- Only allow one specific guest to call to OP-TEE via SMC and blocking any other guest from making SMC calls.
- Either translate the kernel physical addresses to *real* physical addresses *or* ensure the Linux kernel can do the translation itself. A minimal implementation of this could be an identity mapping in the translation tables of the guest so that no actual translation is needed (a flat mapping between the intermediate and real physical addresses).
- Ensure memory areas used for OP-TEE communications are pinned. A minimal implementation of this could be for the hypervisor to carve out a specific region of physical memory for the guest that it will not touch leaving the guest kernel in full control of said memory (allowing it to do the pinning).

One optional, but probably desirable, requirement for the hypervisor would be for it to validate the memory regions passed to OP-TEE. This would be required to prevent the guest using OP-TEE to access the memory of other guests (or even the hypervisor) via OP-TEE.

Enabling TEE in Apertis

Apertis does not currently provide the majority of the functionality needed to implement a TEE. A number of steps need to be taken in order to enable TEE support in Apertis.

It is expected that the above OP-TEE support would be integrated into Apertis in a number of phases:

Phase	Description
0	Suitable reference platform selection

Phase	Description
1	Integration of core components and basic operation
2	Addition of hypervisor support using upstream supported hypervisor (Xen)
3	Creation of TEE Proxy
4	Further investigation of TEE tunnel and documenting of the process

Reference Platform Selection

A critical part of integrating the OP-TEE functionality into Apertis is providing a working implementation on a reference platform to validate the integrated components and serve as an example for Apertis users. This enables them to experiment with and learn about the capabilities, so that they can implement OP-TEE successfully in their own systems.

To do this Apertis needs a reference platform that provides existing open source support, or one which would require minimal work to implement the required support. In this instance we need a platform that supports OP-TEE and the chosen hypervisor. In order for the TEE environment to be truly useful, it is necessary that guarantees can be made that the boot process hasn't been tampered with. As a result, a platform where we can also integrate secure boot effectively will be valuable.

The existing Apertis reference platforms do not fully meet these requirements. The OP-TEE project is specifically targeted towards the ARM ecosystem and particularly those that provide ARM TrustZone. ARM TrustZone has been improved in later iterations of the technology and standardized with a reference implementation, available as part of the ATF project, for using TEEs. We recommend that a platform that is capable of utilizing ATF is chosen for this reference. An advantage of implementing the TEE using ATF is that this provides a standardized interface for the trusted OS and thus allows Apertis to potentially be used with alternative trusted OS implementations. Whilst the Renesas R-Car platform, an existing reference platform, appears to have OP-TEE support[20], it is not openly available and therefore not viable for Apertis to use as a reference for this functionality.

There are a limited number of ARM based processors, and thus development boards, that are listed as having Xen support[21]. Two platforms that stand out as potential options at this point are the 96 Boards HiKey960[22] and a board based on the Rockchip RK3399.

[20] https://optee.readthedocs.io/en/latest/general/platforms.html

[21] https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions#Hardware

[22] https://www.96boards.org/product/hikey960/

Core components

The following core components would need integrating or work in order to provide basic operation of OP-TEE.

Secure Boot

Secure boot provides an initial important step in initialization of the TEE by ensuring that the initialization process is able to proceed without interference. Unfortunately this fundamental step is very platform dependent and can not be solved as a general case. Apertis has already taken steps to document and demonstrate secure boot[23]. At the moment, Apertis only ships some support for secure on the SABRE Lite platform. This provides a good reference for the overall process but, unfortunately, the SABRE Lite is not a good choice as a technology demonstrator for TEE due to its age.

We advise the implementation of a TEE demonstrator on a more modern platform, utilizing ATF, to take advantage of the more advanced functionality found in such platforms.

In addition to the board verifying the initial binaries that are executed, it is important that the verification of binaries continues through the boot process in order to build a chain of trust[24] so that later stages can determine whether boot was carried out appropriately.

ARM Trusted Firmware

The current ARM Trusted Firmware package in Debian does not build for any platforms currently supported in Apertis. The package will need to be tweaked to sign the ATF binaries using an Apertis key. In order to support ATF in Apertis, one of the following options will need to be taken:

- Adopt a platform already supported by the build as an additional platform in Apertis
- Enable support for a platform supported by ATF but not currently built by the deb packaging
- Add support for a preferred platform to ATF and enable it in the packaging

From the perspective of enabling ATF, these are broadly in order of effort, though clearly adding an additional platform to Apertis increases the effort for ongoing baseline maintenance.

Requirements

[23] https://em.pages.apertis.org/apertis-website/architecture/secure-boot/
[24] https://en.wikipedia.org/wiki/Chain_of_trust

In order to implement Trusted Board Boot[25] it will be necessary to upgrade `mbedtls`. This functionality is likely to be considered critical by project developers.

OP-TEE OS

The OP-TEE project provides the OP-TEE OS[26] as the trusted OS that runs in the TEE. This is not currently packaged for Debian and it would need to be to incorporated into Apertis. Like ATF, an Apertis key will need to be used to sign the binaries intended for the TEE to ensure the chain of trust. Currently when OP-TEE is built, it embeds the public key that will be used for verifying TAs. As with the key/keys used in other steps of this process, in order to ensure that products are properly secured, it would be necessary for product teams to at a minimum replace the key used with a product specific one. A product team may wish to modify OP-TEE to support alternative key management solutions, this is expected by the OP-TEE developers[27].

In addition to the trusted OS, the build of the OP-TEE OS source also builds the TA-devkit. The TA-devkit provides the resources necessary to both build and sign TAs. The TA-devkit will need to be packaged so that it can be provided as a build dependency for any TAs.

Linux Kernel

Debian (and thus the Apertis configuration) does already enable the TEE subsystem on arm64 where ATF can be used. It is understood that this should be sufficient and thus no extra modifications to the kernel will be required.

OP-TEE Supplicant and User Space Libraries

In addition to the trusted OS, the OP-TEE project provides the OP-TEE supplicant and TEE Client API[28]. The supplicant provides services to OP-TEE that it does not directly provide itself and the TEE Client API provides a user space API in the REE to communicate with the TEE. As with the OP-TEE OS, these components are currently not packaged for Debian and would need to be. As these components run in the REE they don't need to be signed.

Sample TAs

The example TAs[29] should be packaged so that they can be easily installed on an Apertis. This will enable early investigation of TEEs on Apertis, enabling developers to gain experience with using the TEE. It will provide an example of how to best package TAs for use on Apertis based systems. Depending on the

[25] https://trustedfirmware-a.readthedocs.io/en/latest/design/trusted-board-boot.html
[26] https://github.com/OP-TEE/optee_os
[27] https://github.com/OP-TEE/optee_os/issues/2233#issuecomment-379253182
[28] https://github.com/OP-TEE/optee_client
[29] https://github.com/linaro-swg/optee_examples

task that developers wish to solve using the TEE, these examples may either fulfill or provide a framework for development of the TEE requirements. For example, the examples show how to implement secure storage, implement tee-supplicant plugins and use a TA to perform encryption and decryption.

The sample TAs will be signed with the key provided by the Apertis TA-devkit package (which will be a build dependency) and thus will be usable with the OP-TEE OS built for Apertis.

Debos Scripting

Once components are added to the Apertis project, we need a way to combine them into an image that can be booted on the target platform. In Apertis this is performed by Debos using configuration files to determine exactly what packages are added to each image. This also allows for the images to be built automatically and regularly using the latest versions of packages. A special image to automate configuration of the boot process can also be generated like the one provided to update the U-Boot bootloader for the i.MX6 SABRE Lite board[30].

Hypervisor Integration

Hypervisors come in a number of different types, the main 2 classifications of hypervisor are called "type 1" and "type 2". Type 1 hypervisors run on bare metal, where as type 2 run on top of a host operating system.

Apertis supports the use of VirtualBox[31] for running the Apertis SDK, however this is not intended (nor possible in many instances) to be run on target hardware and is very much a type 2 hypervisor. Apertis also has the Linux KVM enabled which turns the Linux kernel into a hypervisor. KVM is not a clear cut type 1 or type 2 hypervisor.

We'd expect a product to utilize a type 1 hypervisor such as the open source Xen hypervisor, which contains the experimental TEE mediator. We therefore suggest utilizing this in Apertis to avoid needing to implement TEE mediator support in a different hypervisor at this point.

Xen Hypervisor

The Xen hypervisor is a GPL-2 licensed type 1 hypervisor. It is supported on Intel and ARM architectures and has the experimental TEE mediator. Xen is packaged for Debian on the `amd64`, `arm64` and `armhf` architectures which will ease adding support to Apertis, though some work is expected to ensure that the version of Xen and the version of the kernel used in Apertis are compatible.

[30] https://gitlab.apertis.org/infrastructure/apertis-image-recipes/-/blob/apertis/v2021dev2/mx6qsabrelite-uboot-installer.yaml

[31] https://em.pages.apertis.org/apertis-website/guides/virtualbox/

As previously mentioned, the TEE mediator support is still considered experimental. It is understood that this functionality has been tested on the Renesas R-Car H3 platform. Unfortunately this platform requires components that aren't openly available and thus does not present a good reference platform for Apertis. Some effort may be required to port this functionality to another platform.

If there is a wish to support hardware access, significant effort may be required to move the state of art forward.

Linux Kernel

It will be necessary to ensure that the required support is enabled in the Apertis kernel builds, however this should be a relatively straight forward task as the required configuration options are documented on the Xen website[32].

Dual configuration build of OP-TEE

Enabling the virtualization support in OP-TEE results in a version of OP-TEE that only works in virtualized environments. In order to continue to support simple and more complex configurations using a hypervisor it will be necessary to build and package at least two versions of OP-TEE, one supporting virtualization and one that doesn't.

TEE Proxy

The concept of a TEE Proxy appears to be new and no existing such projects have yet been found. The implementation of the TEE Proxy will need careful consideration of the potential security implications. The environment in which the TEE Proxy executes will also need to be trusted and will need to be carefully secured to minimize the risk that third parties could gain access to the communications between the TEE Proxy and it's client or the TEE Proxy and the TEE it's self. The TEE Proxy will also need to be protected against direct attacks against the proxy it's self.

Low Impact Hypervisor Support

The main deliverable from the completion of the low impact hypervisor support effort should be documentation describing in as generic a way as possible the minimal support required from a hypervisor to enable TEE support. This will require a good understanding of the underlying mechanisms used by hypervisors as well as communication between the TEE and REE. We would expect this to be carried out after basic and Xen hypervisor support had been added to Apertis, providing the Apertis team with some of the required experience to formulate this documentation.

[32] https://wiki.xenproject.org/wiki/Mainline_Linux_Kernel_Configs

In order to ensure that this document is accurate and the requirements well under understood, it will be necessary to implement the low impact hypervisor support in a hypervisor. This could be achieved by using Xen or by implementing such support for the KVM hypervisor.

Test Integration

The availability of a test suite[33] test suite provides some coverage of the OP-TEE functionality with minimal effort as this should be usable from automated testing. The test suite should also enable developers to easily gain some confidence that OP-TEE was installed and initialized correctly. Whilst not something that the Apertis project will utilize (due to it being proprietary), the availability of an extended test suite[34], which can be purchased from GlobalPlatform may be something that users of Apertis may wish to consider to provide extended testing.

In addition to the test suite, OP-TEE provides a benchmark framework[35], which may be beneficial to product teams with a desire or need to track execution performance to ensure that product requirements are being met.

Whilst the test suite will test operation of OP-TEE itself, an important part of initializing a TEE is the platform specific secure boot. Unless using a platform very closely aligned with an Apertis reference platform, this step will be the responsibility of the product team.

To ensure that this is properly implemented, tests could be developed that attempt to utilize incorrectly signed binaries at the different stages of the boot process to ensure that each step is properly validated, providing a reference for how to test secure boot.

Experience with the SABRE Lite has shown that whilst devices may be set up to emulate a secured configuration, their behavior differs from the behavior of devices locked via its embedded fuses. Since boards locked in a secure boot configuration no longer allow some operations, they become less useful for general development. For this reason, a dedicated set of boards locked via fuses may be required to fully test that secure boot restrictions are being enforced.

[33] https://github.com/OP-TEE/optee_test

[34] https://optee.readthedocs.io/en/latest/building/gits/optee_test.html#extended-test-globalplatform-tests

[35] https://optee.readthedocs.io/en/latest/building/gits/optee_benchmark.html