OP-TEE Documentation

TrustedFirmware.org

Aug 24, 2020

Contents

1 Getting started **3**
- 1.1 About OP-TEE . . . 3
- 1.2 Coding standards . . . 4
- 1.3 Contribute . . . 5
- 1.4 Contact . . . 8
- 1.5 License headers . . . 9
- 1.6 Platforms supported . . . 10
- 1.7 Presentations . . . 11
- 1.8 Releases . . . 12

2 Architecture **17**
- 2.1 Core . . . 17
- 2.2 Cryptographic implementation . . . 37
- 2.3 Device Tree . . . 40
- 2.4 File structure . . . 41
- 2.5 GlobalPlatform API . . . 43
- 2.6 Libraries . . . 52
- 2.7 Porting guidelines . . . 52
- 2.8 Secure boot . . . 58
- 2.9 Secure storage . . . 59
- 2.10 Trusted Applications . . . 66
- 2.11 Virtualization . . . 76

3 Build and run **79**
- 3.1 AOSP . . . 79
- 3.2 Device specific information . . . 82
- 3.3 Linux kernel TEE framework . . . 103
- 3.4 OP-TEE gits . . . 103
- 3.5 Prerequisites . . . 132
- 3.6 Toolchains . . . 133
- 3.7 Trusted Applications . . . 134

4 Debugging techniques **143**
- 4.1 Abort dumps / call stack . . . 143
- 4.2 Benchmark framework . . . 145
- 4.3 Ftrace . . . 148
- 4.4 Gprof . . . 149

5 Frequently Asked Questions **151**
5.1 Abbreviations . . . 153
5.2 Architecture . . . 153
5.3 Board support . . . 155
5.4 Building . . . 155
5.5 Certification and security reviews . . . 156
5.6 Contribution . . . 157
5.7 Interfaces . . . 158
5.8 Hardware and peripherals . . . 158
5.9 License . . . 158
5.10 Promotion . . . 159
5.11 Security vulnerabilities . . . 159
5.12 Source code . . . 159
5.13 Testing . . . 160
5.14 Trusted Applications . . . 160

This is the official location for OP-TEE documentation.

CHAPTER 1

Getting started

This contains general information about OP-TEE, how to get in contact, how to contribute, how to report security issues etc. It is intended for people who are new to OP-TEE.

1.1 About OP-TEE

OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel running on Arm; Cortex-A cores using the TrustZone technology. OP-TEE implements *TEE Internal Core API* v1.1.x which is the API exposed to Trusted Applications and the *TEE Client API* v1.0, which is the API describing how to communicate with a TEE. Those APIs are defined in the *GlobalPlatform API* specifications.

The non-secure OS is referred to as the Rich Execution Environment (REE) in TEE specifications. It is typically a Linux OS flavor as a GNU/Linux distribution or the AOSP.

OP-TEE is designed primarily to rely on the Arm TrustZone technology as the underlying hardware isolation mechanism. However, it has been structured to be compatible with any isolation technology suitable for the TEE concept and goals, such as running as a virtual machine or on a dedicated CPU.

The main design goals for OP-TEE are:

- **Isolation** - the TEE provides isolation from the non-secure OS and protects the loaded Trusted Applications (TAs) from each other using underlying hardware support,
- **Small footprint** - the TEE should remain small enough to reside in a reasonable amount of on-chip memory as found on Arm based systems,
- **Portability** - the TEE aims at being easily pluggable to different architectures and available HW and has to support various setups such as multiple client OSes or multiple TEEs.

1.1.1 OP-TEE components

OP-TEE is divided in various components:

- A secure privileged layer, executing at Arm secure PL-1 (v7-A) or EL-1 (v8-A) level.

- A set of secure user space libraries designed for Trusted Applications needs.
- A Linux kernel TEE framework and driver (merged to the official tree in v4.12).
- A Linux user space library designed upon the GlobalPlatform *TEE Client API* specifications.
- A Linux user space supplicant daemon (tee-supplicant) responsible for remote services expected by the TEE OS.
- A test suite (xtest), for doing regression testing and testing the consistency of the API implementations.
- An example git containing a couple of simple host- and TA-examples.
- And some build scripts, debugging tools to ease its integration and the development of Trusted Applications and secure services.

These components are available from several git repositories. The main ones are *build*, *optee_os*, *optee_client*, *optee_test*, *optee_examples* and the *Linux kernel TEE framework*.

1.1.2 History

OP-TEE was initially developed by ST-Ericsson (and later on by STMicroelectronics), but this was before OP-TEE got the name "OP-TEE" and was turned into an open source project. Back then it was a closed source and a proprietary TEE project. In 2013, ST-Ericsson obtained GlobalPlatform's compliance qualification with this implementation, proving that the APIs were behaving as expected according to the GlobalPlatform specifications.

Later on the same year (2013) Linaro was about to form Security Working Group (SWG) and one of the initial key tasks for SWG was to work on an open source TEE project. After talking to various TEE vendors Linaro ended up working with STMicroelectronics TEE project. But before being able to open source it there was a need to replace some proprietary components with open source components. For a couple of months Linaro/SWG together with engineers from STMicroelectronics re-wrote major parts (crypto library, secure monitor, build system etc), cleaned up the project by enforcing *Coding standards*, running checkpatch etc.

June 12 2014 was the day when OP-TEE was "born" as an open source project. At that day the OP-TEE team pushed the first commit to GitHub. A bit after this Linaro also made a press release about this. That press release contains a bit more information. At the first year as an open source project it was owned by STMicroelectronics but maintained by Linaro and STMicroelectronics. In 2015 there was an ownership transfer of OP-TEE from STMicroelectronics to Linaro. In September 2019, ownership was transferred from Linaro to the TrustedFirmware.org project (see _blogpost for more information). Maintenance is a shared responsibility between the members for TrustedFirmware.org and some community maintainers representing other companies who are using OP-TEE.

1.2 Coding standards

In this project we are trying to adhere to the same coding convention as used in the Linux kernel (see CodingStyle). We achieve this by running checkpatch from Linux kernel. However there are a few exceptions that we had to make since the code also follows GlobalPlatform standards. The exceptions are as follows:

1. **CamelCase** for GlobalPlatform types is allowed.
2. We **do not** run checkpatch on third party code that we might use in this project, such as LibTomCrypt, MPA, newlib etc. The reason for that and not doing checkpatch fixes for third party code is because we would probably deviate too much from upstream and therefore it would be hard to rebase against those projects later on and we don't expect that it is easy to convince other software projects to change coding style.
3. **All** variables **shall be** initialized to a well known value in one or another way. The reason for that is that we have had potential security issues in the past that originated from not having variables initialized with a well defined value. We have also investigate various toolchain flags that are supposed to help out finding uninitialized

variables. Unfortunately our conclusion is that you cannot trust the compilers here, since there are corner cases where compilers cannot reliably give a warning.

Variables are initialized according to these general guidelines:

- Scalars (and types like `time_t` which are standardized as scalars) are initialized with `0`, unless another value makes more sense.
- For optee_client we need maximum portability. So only initialize struct types (and `pthread_t`) with `memset()` unless there is a good reason not to do so.
- For the rest of the gits we assume that a recent version of GCC or Clang is used so we initialize structs with `{ }` in order to avoid the more clumsy `memset()` procedure. Types like `pthread_t` which can be a scalar or a composite type are initialized with `memset()` in order to minimize the amount of future headache.

Regarding the checkpatch tool, it is not included directly into this project. Please use checkpatch.pl from the Linux kernel git in combination with the local checkpatch script.

There are also targets for common use cases in the Makefiles:

```
make checkpatch         #check staging and working area
make checkpatch-staging #check staging area (added, but not committed files)
make checkpatch-working #check working area (modified, but not added files)
```

1.3 Contribute

Contributions to OP-TEE are managed by the OP-TEE *Core Team* and anyone can contribute to OP-TEE as long as it is understood that it will require a *Signed-off-by* tag from the one submitting the patch(es). The Signed-off-by tag is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open source patch (see below). You thereby assure that you have read and are following the rules stated in the `Developer Certificate of Origin` as stated below.

1.3.1 Developer Certificate of Origin

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
660 York Street, Suite 102,
San Francisco, CA 94110 USA

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
```

(continues on next page)

(continued from previous page)

```
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

We have borrowed this procedure from the Linux kernel project to improve tracking of who did what, and for legal reasons.

To sign-off a patch, just add a line in the commit message saying:

```
Signed-off-by: Random J Developer <random@developer.example.org>
```

Use your real name or on some rare cases a company email address, but we disallow pseudonyms or anonymous contributions.

1.3.2 GitHub

This section describes how to use GitHub for OP-TEE development and contributions.

Setting up an account

You do not need to own a GitHub account in order to clone a repository. But if you want to contribute, you need to create an account at GitHub. Note that a free plan is sufficient to collaborate.

SSH is recommended to access your GitHub repositories securely and without supplying your username and password each time you pull or push something. To configure SSH for GitHub, please refer to Connecting to GitHub with SSH.

Forking

Only owners of the OP-TEE projects have write permissions to the git repositories of those projects. Contributors **should fork** `OP-TEE/*.git` and/or `linaro-swg/*.git` into their own account, then work on this forked repository. The complete documentation about **forking** can be found at fork a repo.

Creating pull requests

When you want to submit a patch to the OP-TEE project, you are supposed to create a pull request to the git where you forked your git from. How that is done using GitHub is explained at the GitHub pull request page.

Commit messages

- The **subject line** should explain **what** the patch does as precisely as possible. It is usually prefixed with keywords indicating which part of the code is affected, but not always. Avoid lines longer than 80 characters.
- The **commit description** should give more details on **what** is changed, and explain **why** it is done. Indication on how to enable and use some particular feature can be useful, too. Try to limit line length to 72 characters, except when pasting some error message (compiler diagnostic etc.). Long lines are allowed to accommodate URLs, too (preferably use URLs in a Fixes: or Link: tag).
- The commit message **must** end with a blank line followed by some tags, including your `Signed-off-by:` tag. By applying such a tag to your commit, you are effectively declaring that your contribution follows the terms stated by *Developer Certificate of Origin* (in the DCO section there is also a complete example).
- Other tags may be used, such as:
  - `Tested-by: Test R <test@r.com>`
  - `Acked-by: Acke R <acke@r.com>`
  - `Suggested-by: Suggeste R <suggeste@r.com>`
  - `Reported-by: Reporte R <reporte@r.com>`
- When citing a previous commit, whether it is in the text body or in a Fixes: tag, always use the format shown above (12 hexadecimal digits prefix of the commit `SHA1`, followed by the commit subject in double quotes and parentheses).

Review feedback

It is very likely that you will get review comments from other OP-TEE users asking you to fix certain things etc. When fixing review comments, do:

- **Add** *fixup* patches on **top** of your existing branch. **Do not** squash and force push while fixing review comments.
- When all comments have been addressed, just write a simple messages in the comments field saying something like "All comments have been addressed". By doing so you will notify the maintainers that the fix might be ready for review again.

Finalizing your contribution

Once you and reviewers have agreed on the patch set, which is when all the people who have commented on the pull request have given their `Acked-by:` or `Reviewed-by:`, you need to consolidate your commits:

Use `git rebase -i` to squash the fixup commits (if any) into the initial ones. For instance, suppose the `git log --oneline` for you contribution looks as follows when the review process ends:

```
<sha1-commit4> [Review] Do something
<sha1-commit3> [Review] Do something
<sha1-commit2> Do something else
<sha1-commit1> Do something
```

Then you would do:

```
$ git rebase -i <sha1-commit1>^
```

Edit the commit script so it looks like so:

```
pick <sha1-commit1> Do something
squash <sha1-commit3> [Review] Do something
squash <sha1-commit4> [Review] Do something
pick <sha1-commit2> Do something else
```

Add the proper tags (`Acked-by: ...`, `Reviewed-by: ...`, `Tested-by: ...`) to the commit message(s) for each and every commit as provided by the people who reviewed and/or tested the patches.

Hint: `git commit --fixup=<sha1-of-commit-to-fix>` and later on `git rebase -i --autosquash <sha1-of-first-commit-in-patch-serie>^1` is pretty convenient to use when adding review patches and doing the final squash operation.

Once `rebase -i` is done, you need to force-push (`-f`) to your GitHub branch in order to update the pull request page.

```
$ git push -f <my-remote> <my-branch>
```

After completing this it is the project maintainers job to apply your patches to the master branch.

1.4 Contact

1.4.1 GitHub

Our preference is to use GitHub for communication. The reason for that is that it is an open source project, so there should be no real reason to hide discussions from other people. GitHub also makes it possible for anyone to chime in into discussion etc. So besides sending patches as pull requests on GitHub we also encourage people to use the "issues" to report bugs, give suggestions, ask questions etc.

Please try to use the "issues" in the relevant git. I.e., if you want to discuss something related to optee_client, then use "issues" at *optee_client* and so on. If you have a general question etc about OP-TEE that doesn't really belong to a specific git, then please use issues at optee_os in that case.

1.4.2 Email

You can reach the *Core Team* by sending an email to `op-tee[at]lists.trustedfirmware.org`. However note that it's a public mailinglist and **not** just TrustedFirmware engineers behind that email address.

For pure Linux kernel patches, please use the appropriate Linux kernel mailinglist, basically run the `get_maintainer.pl` script in the Linux kernel tree to figure out where to send your patches.

```
$ cd <linux-kernel>
$ ./scripts/get_maintainer.pl drivers/tee/
```

1.4.3 IRC

Some of the OP-TEE developers can be reached at Freenode (`chat.freenode.net`) at channel `#linaro-security`. Having that said, the activity there is a bit limited, so it is probably **not** the best place to discuss OP-TEE.

1.4.4 Vulnerability reporting

The OP-TEE project as part of the TrustedFirmware.org organization is using the security incident process as described at the TrustedFirmware.org security incident page. To report an issue, please follow the process as specified here. The email address to use can be found at the Mailing Aliases page.

1.4.5 Core Team

The core team consists of engineers from TrustedFirmware.org. Related, see the core team at GitHub.

1.5 License headers

This document defines the format of the copyright and license headers in OP-TEE source files. Such headers shall comply with the rules described here, which are compatible with the rules adopted by the Linux kernel community.

1.5.1 New source files

- **Rule 1.1** Shall contain exactly one SPDX license identifier, which can express a single or multiple licenses (refer to SPDX for syntax details).
- **Rule 1.2** The SPDX license identifier shall be added as a comment line. It shall be the first possible line in the file which can contain a comment. The comment style shall depend on the file type:
  - **Rule 1.2.1** C source: `// SPDX-License-Identifier: <expression>`
  - **Rule 1.2.2** C header: `/* SPDX-License-Identifier: <expression> */`
  - **Rule 1.2.3** Assembly: `/* SPDX-License-Identifier: <expression> */`
  - **Rule 1.2.4** Python, shell: `# SPDX-License-Identifier: <expression>`
- **Rule 1.3** Shall contain at least one copyright line
- **Rule 1.4** Shall not contain the mention 'All rights reserved'
- **Rule 1.5** Shall not contain any license notice other than the SPDX license identifier

Note that files imported from external projects are not new files. The rules for pre-existing files (below) apply.

1.5.2 Pre-existing or imported files

- **Rule 2.1** SPDX license identifiers shall be added according to the license notice(s) in the file and the rules above (1.1 and 1.2*)
- **Rule 2.2** It is recommended that license notices be removed once the corresponding identifier has been added. Note however that this may only be done by the copyright holder(s) of the file.
- **Rule 2.3** Similar to 2.2, and subject to the same conditions, the text: "All rights reserved" shall be removed also.

1.6 Platforms supported

Several platforms are supported. In order to manage slight differences between platforms, a `PLATFORM_FLAVOR` flag has been introduced. The `PLATFORM` and `PLATFORM_FLAVOR` flags define the whole configuration for a chip the where the Trusted OS runs. Note that there is also a composite form which makes it possible to append `PLATFORM_FLAVOR` directly, by adding a dash in-between the names. The composite form is shown below for the different boards. For more specific details about build flags etc, please read *Configuration and flags*. Some platforms have different sub-maintainers, please refer to the file MAINTAINERS for contact details for various platforms.

Table 1: Platforms officially supported in OP-TEE

Platform	Composite PLATFORM flag	Publicly available
ARM Juno Board	`PLATFORM=vexpress-juno`	Yes
Atmel ATSAMA5D2-XULT Board	`PLATFORM=sam`	Yes
Broadcom ns3	`PLATFORM=bcm-ns3`	No
DeveloperBox (Socionext Synquacer SC2A11)	`PLATFORM=synquacer`	Yes
FSL ls1021a	`PLATFORM=ls-ls1021atwr`	Yes
NXP ls1043ardb	`PLATFORM=ls-ls1043ardb`	Yes
NXP ls1046ardb	`PLATFORM=ls-ls1046ardb`	Yes
NXP ls1012ardb	`PLATFORM=ls-ls1012ardb`	Yes
NXP ls1028ardb	`PLATFORM=ls-ls1028ardb`	Yes
NXP ls1088ardb	`PLATFORM=ls-ls1088ardb`	Yes
NXP ls2088ardb	`PLATFORM=ls-ls2088ardb`	Yes
NXP ls1012afrwy	`PLATFORM=ls-ls1012afrwy`	Yes
FSL i.MX6 Quad SABRE Lite Board	`PLATFORM=imx-mx6qsabrelite`	Yes
FSL i.MX6 Quad SABRE SD Board	`PLATFORM=imx-mx6qsabresd`	Yes
SolidRun i.MX6 Quad Hummingboard Edge	`PLATFORM=imx-mx6qhmbedge`	Yes
SolidRun i.MX6 Dual Hummingboard Edge	`PLATFORM=imx-mx6dhmbedge`	Yes
SolidRun i.MX6 Dual Lite Hummingboard Edge	`PLATFORM=imx-mx6dlhmbedge`	Yes
SolidRun i.MX6 Solo Hummingboard Edge	`PLATFORM=imx-mx6shmbedge`	Yes
FSL i.MX6 UltraLite EVK Board	`PLATFORM=imx-mx6ulevk`	Yes
NXP i.MX7Dual SabreSD Board	`PLATFORM=imx-mx7dsabresd`	Yes
NXP i.MX7Solo WaRP7 Board	`PLATFORM=imx-mx7swarp7`	Yes
NXP i.MX8MQEVK Board	`PLATFORM=imx-imx8mqevk`	Yes
NXP i.MX8MMEVK Board	`PLATFORM=imx-imx8mmevk`	Yes
ARM Foundation FVP	`PLATFORM=vexpress-fvp`	Yes
HiSilicon D02	`PLATFORM=d02`	No
HiSilicon Hi3519AV100 Demo Board	`PLATFORM=hisilicon-hi3519av100_demo`	No
HiKey Board (HiSilicon Kirin 620)	`PLATFORM=hikey` or `PLATFORM=hikey-hikey`	Yes
HiKey960 Board (HiSilicon Kirin 960)	`PLATFORM=hikey-hikey960`	Yes
Marvell ARMADA 7K Family	`PLATFORM=marvell-armada7k8k`	Yes
Marvell ARMADA 8K Family	`PLATFORM=marvell-armada7k8k`	Yes
Marvell ARMADA 3700 Family	`PLATFORM=marvell-armada3700`	Yes
MediaTek MT8173 EVB Board	`PLATFORM=mediatek-mt8173`	No
Poplar Board (HiSilicon Hi3798C V200)	`PLATFORM=poplar`	Yes
QEMU	`PLATFORM=vexpress-qemu_virt`	Yes
QEMUv8	`PLATFORM=vexpress-qemu_armv8a`	Yes
Raspberry Pi 3	`PLATFORM=rpi3`	Yes
Renesas RCAR	`PLATFORM=rcar`	No
Rockchip PX30	`PLATFORM=rockchip-px30`	No
Rockchip RK322X	`PLATFORM=rockchip-rk322x`	No

Continu

Table 1 – continued from previous page

Platform	Composite PLATFORM flag	Publicly available
Rockchip RK3399	`PLATFORM=rockchip-rk3399`	Yes
STMicroelectronics b2260 - h410 (96boards fmt)	`PLATFORM=stm-b2260`	No
STMicroelectronics b2120 - h310 / h410	`PLATFORM=stm-cannes`	No
STMicroelectronics STM32MP1 series	`PLATFORM=stm32mp1`	Yes
Allwinner A64 Pine64 Board	`PLATFORM=sunxi-sun50i_a64`	Yes
Texas Instruments AM65x	`PLATFORM=k3-am65x`	Yes
Texas Instruments DRA7xx	`PLATFORM=ti-dra7xx`	Yes
Texas Instruments AM57xx	`PLATFORM=ti-am57xx`	Yes
Texas Instruments AM43xx	`PLATFORM=ti-am43xx`	Yes
Xilinx Zynq 7000 ZC702	`PLATFORM=zynq7k-zc702`	Yes
Xilinx Zynq UltraScale+ MPSOC	`PLATFORM=zynqmp-zcu102`	Yes
Spreadtrum SC9860	`PLATFORM=sprd-sc9860`	No

1.7 Presentations

Below are presentations coming from engineers working with OP-TEE in one or another way. Note that the older they are, the less relevant is the information in them. So do not trust blindly what was said back in the days, cross check with latest version to understand whether things have changed or not.

The links are sorted in chronological order, newest first and oldest at the end.

- **BKK19**
  - BKK19-419 - Debugging with OP-TEE (slides, video)
  - BKK19-415 - OP-TEE: Shared memory between TAs (slides, video)
  - BKK19-403 - Using DTB overlays in OP-TEE (slides, video)
  - BKK19-215 - TPM in TEE (slides, video)
  - BKK19-117 - Security WG Lightning talks (slides, video)
- **YVR18**
  - YVR18-414 - Keymaster and Gatekeeper (slides, video)
  - YVR18-117 - SWG updates since HKG18 (slides, video)
- **HKG18**
  - HKG18-402 - Build secure key management services in OP-TEE (slides, video)
- **SFO17**
  - SFO17-309 - Secure storage updates (slides, video)
- **Webinar**
  - TEE Linux kernel support and open source security (slides, video)
- **BUD17**
  - BUD17-416 - Benchmark and profiling in OP TEE (slides, video)
  - BUD17-400 - Secure Data Path with OPTEE (slides, video)
- **LAS16**
  - LAS16-504 - Secure Storage updates in OP-TEE (slides, video)

  - LAS16-406 - Android Widevine on OP-TEE (slides, video)
  - LAS16-111 - Easing Access to ARM TrustZone OP TEE and Raspberry Pi 3 (slides, video)
- **BKK16**
  - BKK16-201 - PlayReady OP-TEE Integration with Secure Video Path (slides, video)
  - BKK16-110 - A Gentle Introduction to Trusted Execution and OP-TEE (slides)
- **SFO15**
  - SFO15-503 - Secure storage in OP-TEE (slides, video)
  - SFO15-205 - OP-TEE Content Decryption with Microsoft PlayReady on ARM TrustZone (slides, video)
  - SFO15-200 - TEE kernel driver (slides, video)
- **HKG15**
  - HKG15-311 - OP-TEE for Beginners and Porting Review (slides, video)
  - HKG15-307 - OP-TEE pager (slides, video)
- **LCU14**
  - LCU14-306 - OP-TEE Future Enhancements (slides)
  - LCU14-302 - How to port OP-TEE to another platform (slides, video)
  - LCU14-107 - OP-TEE on ARMv8-A (slides, video)
  - LCU14-103 - How to create and run Trusted Applications on OP-TEE (slides, video)
- **LCA14**
  - LCA14-502 - The way to a generic TrustZone solution (slides)
  - LCA14-418 - Testing a secure framework (slides)

1.8 Releases

1.8.1 Cadence

New versions of OP-TEE are released four times a year, i.e., quarterly releases.

Release dates

Starting from version 3.10.0 we track the old and also show the releases being planned for the future in the table below. The dates will tell whether it is an old, upcoming or future release.

Version	Release date
OP-TEE 3.13.0	16/Apr/21
OP-TEE 3.12.0	15/Jan/21
OP-TEE 3.11.0	16/Oct/20
OP-TEE 3.10.0	21/Aug/20

1.8.2 Changelog

The changelog is stored in the *optee_os* git (CHANGELOG.md). There you can see what has been done between the different releases in terms of commits as well as pull requests.

1.8.3 Versioning schema

OP-TEE follows Semantic Versioning 2.0.0. What that means in practice is well described at the link just shown.

1.8.4 Release procedure

There are certain steps that needs to be done when making a release. This checklist here serves as guidance to the one in charge of making a new release. Roughly start with this 2-3 weeks before the targeted release date.

tl;dr

Table 2: Short version of the OP-TEE release procedure

When (Tminus)	Action	Example
3w	Create release pull request	PR#3099
3w	Inform maintainers about upcoming release	
1w	Increment the revision number in mk/config.mk	`CFG_OPTEE_REV ? = 3` `CFG_OPTEE_REV ? = x`
1w	Create release candidate tag in optee_* + build.git	`git tag -a 3.x.y-rc1 -m "3.x.y-rc1"`
1w	Let maintainers know about the release candidate tag	
1w	Test platform builds / devices	
Release day	Update CHANGELOG.md	changelog example
Release day	Collect/merge `Tested-By` tags	commit example
Release day	Create release tag in optee_* + build.git + linux.git	`git tag -a 3.x.y -m "3.x.y"` `git tag -a optee-3.x.y -m "optee-3.x.y"` `# (Linux)`

Long version

1. Create a "release pull request" at GitHub ought to collect `Tested-By` tags from various maintainers. As an example, see PR#3099.
2. Send email to all maintainers to let them know about the upcoming release. The addresses to the maintainers can be found in the MAINTAINERS file.

Hint: With this command you will get all email addresses

```
$ scripts/get_maintainer.py --release-to
```

---

3. Increment the revision number in *mk/config.mk*: `CFG_OPTEE_REVISION_MAJOR` and `CFG_OPTEE_REVISION_MINOR`. These values are made available to TAs and to the Normal World driver at boot time.
4. Create a release candidate (RC) tag (annotated tag, i.e., `git tag -a 3.x.y-rc1 -m "3.x.y-rc1"`) in the following gits `optee_*` and `build.git`. One way to do it is like this

```
$ export VER=3.x.y-rc1
$ for d in optee* build; do ( cd $d; git tag -a $VER -m $VER ); done
$ for d in optee* build; do ( cd $d; git push origin $VER ); done
```

5. Send a follow up email to all maintainers to let them know that there is a release tag ready to be tested on their devices for the platforms that they are maintaining.
6. In case major regressions are found, then fix those and create a another release candidate tag (i.e., repeat step 3 and 4 until there are no remaining issues left).
7. On release day: Update CHANGELOG.md see this changelog example to see how that should look like.
8. Collect all tags (`Tested-By` etc) from maintainers and use those in the commit message, for an example see this commit example.
9. Create a release tag (annotated tag, i.e., `git tag -a 3.x.y -m "3.x.y"`) in the following gits `optee_*` and `build.git`. Tag the tip of the `optee` branch in `linux.git`, the name of the tag has to be prefixed with `optee-` to avoid confusions. For instance: `git tag -a optee-3.x.y -m "optee-3.x.y"`.

Hint: You can use the same steps as in step 4, when creating the tags.

---

10. Create a new branch in *manifest* from `master` where the name corresponds to the release you are preparing. I.e., `git checkout -b 3.x.y origin/master`.
11. Update all *manifest* XML-files in the *manifest* git, so they refer to the tag in the release we are working with (see 3.6.0 stable commit as an example). This can be done with the make_stable.sh script. Now it is also time to push the new branch and tag it. Example:

```
$ export VER=3.x.y
$ cd manifest
$ ./make_stable.sh -o -r $VER
$ git diff  # make sure everything looks good
$ git commit -a -m "OP-TEE $VER stable"
$ git remote add upstream git@github.com:OP-TEE/manifest
$ git push upstream
```

(continues on next page)

(continued from previous page)

```
$ git tag -a $VER -m $VER
$ git push upstream tag $VER
```

12. Send a last email to maintainers and other stakeholders telling that the release has been completed.

CHAPTER 2

Architecture

2.1 Core

2.1.1 Interrupt handling

This section describes how *optee_os* handles switches of world execution context based on *SMC* exceptions and interrupt notifications. Interrupt notifications are IRQ/FIQ exceptions which may also imply switching of world execution context: normal world to secure world, or secure world to normal world.

Use cases of world context switch

This section lists all the cases where optee_os is involved in world context switches. Optee_os executes in the secure world. World switch is done by the core's secure monitor level/mode, referred below as the Monitor.

When the normal world invokes the secure world, the normal world executes a SMC instruction. The SMC exception is always trapped by the Monitor. If the related service targets the trusted OS, the Monitor will switch to optee_os world execution. When the secure world returns to the normal world, optee_os executes a SMC that is caught by the Monitor which switches back to the normal world.

When a secure interrupt is signaled by the Arm GIC, it shall reach the optee_os interrupt exception vector. If the secure world is executing, optee_os will handle straight the interrupt from its exception vector. If the normal world is executing when the secure interrupt raises, the Monitor vector must handle the exception and invoke the optee_os to serve the interrupt.

When a non-secure interrupt is signaled by the Arm GIC, it shall reach the normal world interrupt exception vector. If the normal world is executing, it will handle straight the exception from its exception vector. If the secure world is executing when the non-secure interrupt raises, optee_os will temporarily return back to normal world via the Monitor to let normal world serve the interrupt.

Core exception vectors

Monitor vector is `VBAR_EL3` in AArch64 and `MVBAR` in Armv7-A/AArch32. Monitor can be reached while normal world or secure world is executing. The executing secure state is known to the Monitor through the `SCR_NS`.

Monitor can be reached from a SMC exception, an IRQ or FIQ exception (so-called interrupts) and from asynchronous aborts. Obviously monitor aborts (data, prefetch, undef) are local to the Monitor execution.

The Monitor can be external to optee_os (case `CFG_WITH_ARM_TRUSTED_FW=y`). If not, provides a local secure monitor `core/arch/arm/sm`. Armv7-A platforms should use the optee_os secure monitor. Armv8-A platforms are likely to rely on an Trusted Firmware A.

When executing outside the Monitor, the system is executing either in the normal world (`SCR_NS=1`) or in the secure world (`SCR_NS=0`). Each world owns its own exception vector table (state vector):

- `VBAR_EL2` or `VBAR_EL1` non-secure or `VBAR_EL1` secure for AArch64.
- `HVBAR` or `VBAR` non-secure or `VBAR` secure for Armv7-A and AArch32.

All SMC exceptions are trapped in the Monitor vector. IRQ/FIQ exceptions can be trapped either in the Monitor vector or in the state vector of the executing world.

When the normal world is executing, the system is configured to route:

- secure interrupts to the Monitor that will forward to optee_os
- non-secure interrupts to the executing world exception vector.

When the secure world is executing, the system is configured to route:

- secure and non-secure interrupts to the executing optee_os exception vector. optee_os shall forward the non-secure interrupts to the normal world.

Optee_os non-secure interrupts are always trapped in the state vector of the executing world. This is reflected by a static value of `SCR_(IRQ|FIQ)`.

Native and foreign interrupts

Two types of interrupt are defined in optee_os:

- **Native interrupt** - The interrupt handled by optee_os (for example: secure interrupt)
- **Foreign interrupt** - The interrupt not handled by optee_os (for example: non-secure interrupt which is handled by normal world)

For Arm **GICv2** mode, native interrupt is sent as FIQ and foreign interrupt is sent as IRQ. For Arm **GICv3** mode, foreign interrupt is sent as FIQ which could be handled by either secure world (aarch32 Monitor mode or aarch64 EL3) or normal world. Arm GICv3 mode can be enabled by setting `CFG_ARM_GICV3=y`. For clarity, this document mainly chooses the GICv2 convention and refers the IRQ as optee_os foreign interrupts, and FIQ as optee_os native interrupts. Native interrupts must be securely routed to optee_os. Foreign interrupts, when trapped during secure world execution might need to be efficiently routed to the normal world.

Normal World invokes optee_os using SMC

Entering the Secure Monitor

The monitor manages all entries and exits of secure world. To enter secure world from normal world the monitor saves the state of normal world (general purpose registers and system registers which are not banked) and restores the previous state of secure world. Then a return from exception is performed and the restored secure state is resumed. Exit from secure world to normal world is the reverse.

Some general purpose registers are not saved and restored on entry and exit, those are used to pass parameters between secure and normal world (see ARM_DEN0028A_SMC_Calling_Convention for details).

Entry and exit of Trusted OS

On entry and exit of Trusted OS each CPU is uses a separate entry stack and runs with IRQ and FIQ blocked. SMCs are categorised in two flavors: **fast** and **standard**.

- For **fast** SMCs, optee_os will execute on the entry stack with IRQ/FIQ blocked until the execution returns to normal world.
- For **standard** SMCs, optee_os will at some point execute the requested service with interrupts unblocked. In order to handle interrupts, mainly forwarding of foreign interrupts, optee_os assigns a trusted thread (core/arch/arm/kernel/thread.c) to the SMC request. The trusted thread stores the execution context of the requested service. This context can be suspended and resumed as the requested service executes and is interrupted. The trusted thread is released only once the service execution returns with a completion status.

  For **standard** SMCs, optee_os allocates or resumes a trusted thread then unblock the IRQ/FIQ lines. When the optee_os needs to invoke the normal world from a foreign interrupt or a remote service call, optee_os blocks IRQ/FIQ and suspends the trusted thread. When suspending, optee_os gets back to the entry stack.
- **Both** fast and standard SMC end on the entry stack with IRQ/FIQ blocked and optee_os invokes the Monitor through a SMC to return to the normal world.

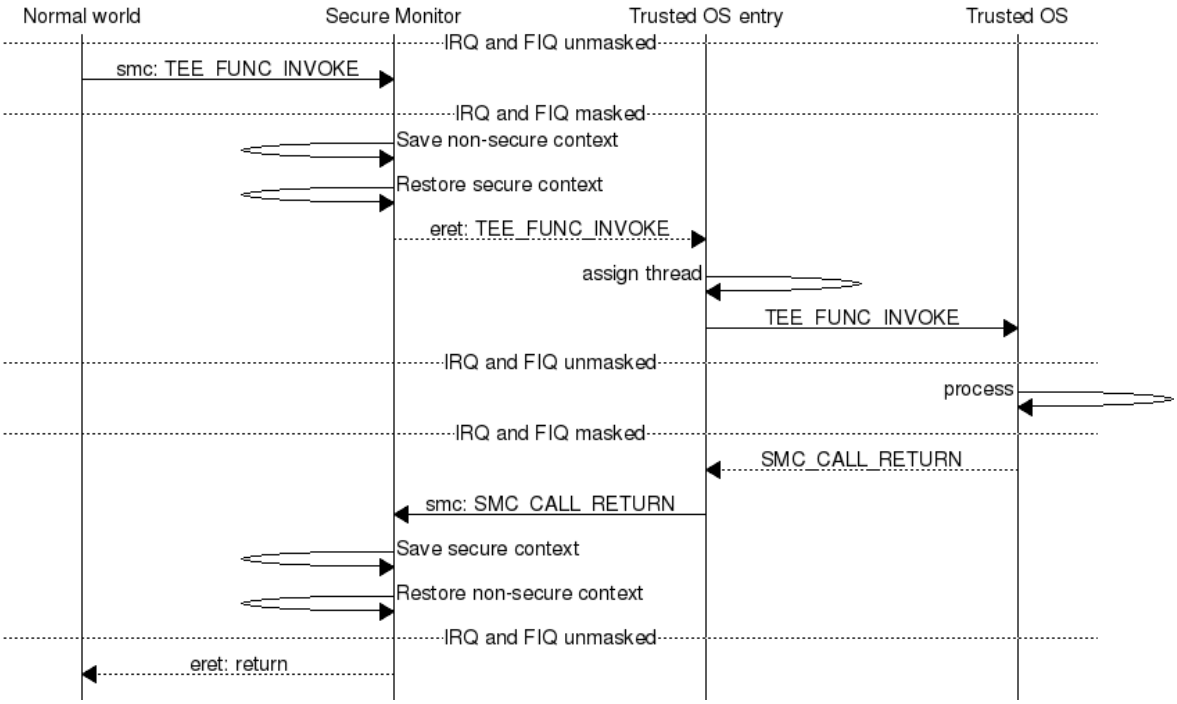


Fig. 1: SMC entry to secure world

Deliver non-secure interrupts to Normal World

This section uses the Arm GICv1/v2 conventions: IRQ signals non-secure interrupts while FIQ signals secure interrupts. On a GICv3 configuration, one should exchange IRQ and FIQ in this section.

Forward a Foreign Interrupt from Secure World to Normal World

When an IRQ is received in secure world as an IRQ exception then secure world:

1. Saves trusted thread context (entire state of all processor modes for Armv7-A)
2. Blocks (masks) all interrupts (IRQ and FIQ)
3. Switches to entry stack
4. Issues an SMC with a value to indicates to normal world that an IRQ has been delivered and last SMC call should be continued

The monitor restores normal world context with a return code indicating that an IRQ is about to be delivered. Normal world issues a new SMC indicating that it should continue last SMC.

The monitor restores secure world context which locates the previously saved context and checks that it is a return from IRQ that is requested before restoring the context and lets the secure world IRQ handler return from exception where the execution would be resumed.

Note that the monitor itself does not know/care that it has just forwarded an IRQ to normal world. The bookkeeping is done in the trusted thread handling in Trusted OS. Normal world is responsible to decide when the secure world thread should resume execution (for details, see *Thread handling*).

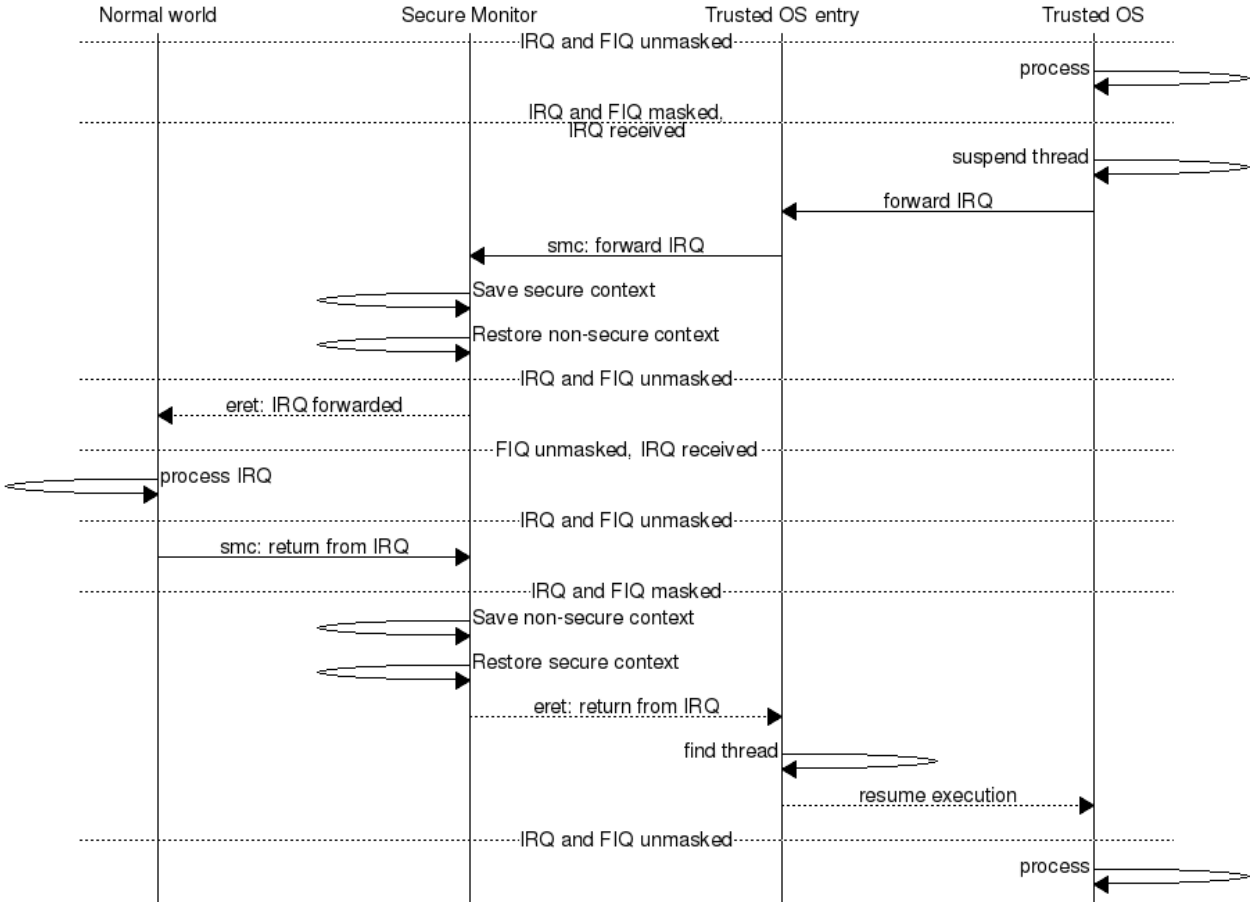


Fig. 2: IRQ received in secure world and forwarded to normal world

Deliver a non-secure interrupt to normal world when ‘‘SCR_NS‘‘ is set

Since `SCR_IRQ` is cleared, an IRQ will be delivered using the state vector (`VBAR`) in the normal world. The IRQ is received as any other exception by normal world, the monitor and the Trusted OS are not involved at all.

Deliver secure interrupts to Secure World

This section uses the Arm GICv1/v2 conventions: FIQ signals secure interrupts while IRQ signals non-secure interrupts. On a GICv3 configuration, one should exchange IRQ and FIQ in this section. A FIQ can be received during two different states, either in normal world (`SCR_NS` is set) or in secure world (`SCR_NS` is cleared). When the secure monitor is active (Armv8-A EL3 or Armv7-A Monitor mode) FIQ is masked. FIQ reception in the two different states is described below.

Deliver FIQ to secure world when SCR_NS is set

When the monitor gets an FIQ exception it:

1. Saves normal world context and restores secure world context from last secure world exit (which will have IRQ and FIQ blocked)
2. Clears `SCR_FIQ` when clearing `SCR_NS`
3. Sets "FIQ" as parameter to secure world entry
4. Does a return from exception into secure context
5. Secure world unmasks FIQs because of the "FIQ" parameter
6. FIQ is received as in exception using the state vector
7. The state vector handle returns from exception in secure world
8. Secure world issues an SMC to return to normal world
9. Monitor saves secure world context and restores normal world context
10. Does a return from exception into restored context

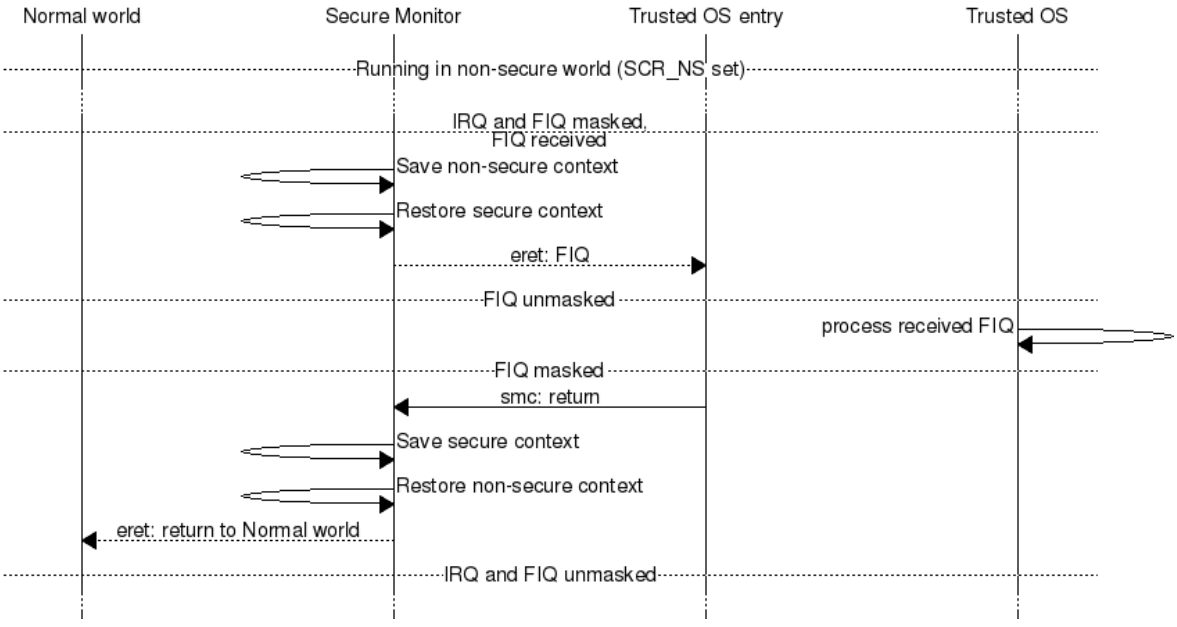


Fig. 3: FIQ received when SCR_NS is set

Deliver FIQ to secure world when SCR_NS is cleared

Since `SCR_FIQ` is cleared when `SCR_NS` is cleared a FIQ will be delivered using the state vector (`VBAR`) in secure world. The FIQ is received as any other exception by Trusted OS, the monitor is not involved at all.

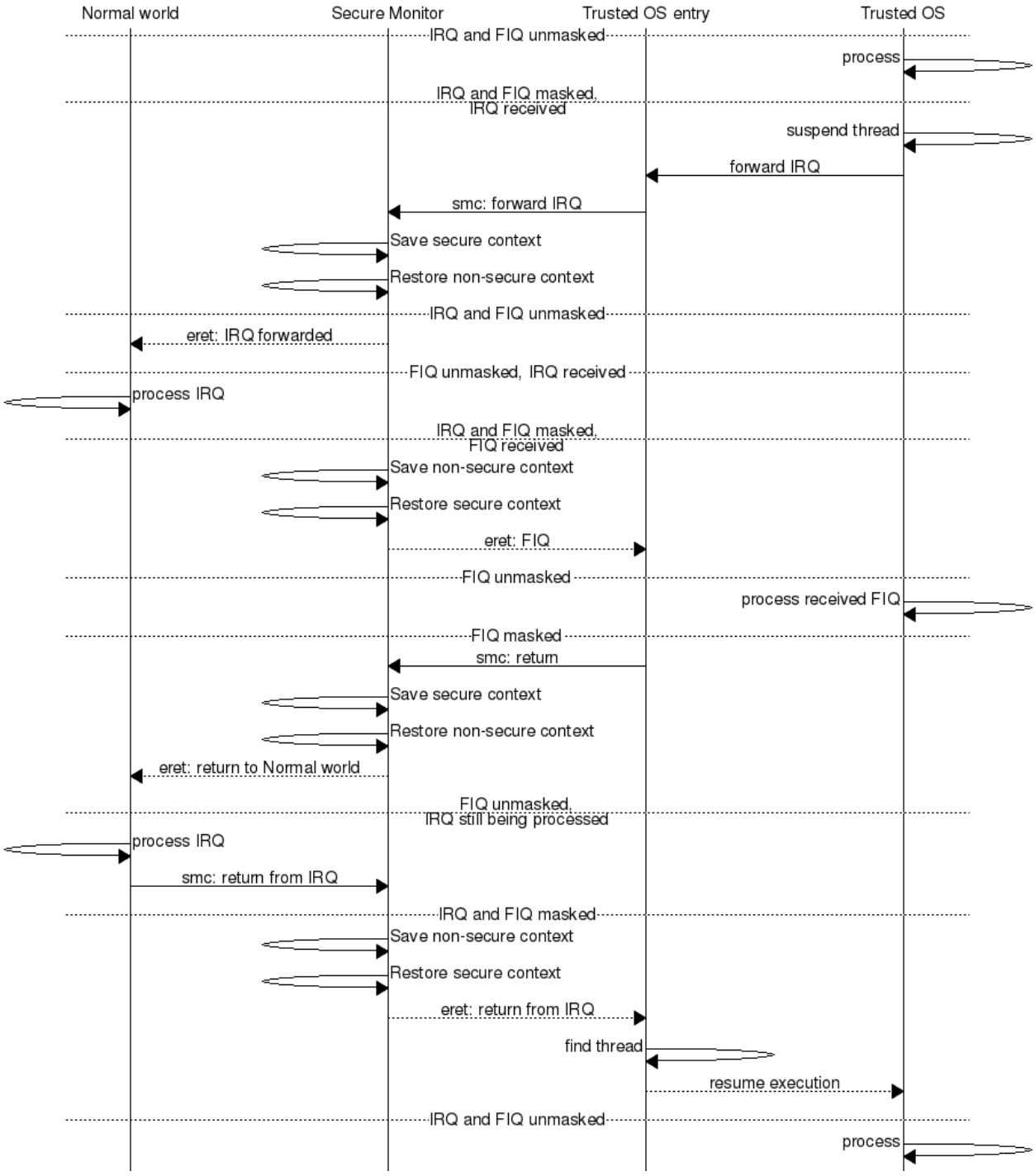


Fig. 4: FIQ received while processing an IRQ forwarded from secure world

Trusted thread scheduling

Trusted thread for standard services

OP-TEE standard services are carried through standard SMC. Execution of these services can be interrupted by foreign interrupts. To suspend and restore the service execution, optee_os assigns a trusted thread at standard SMCs entry.

The trusted thread terminates when optee_os returns to the normal world with a service completion status.

A trusted thread execution can be interrupted by a native interrupt. In this case the native interrupt is handled by the interrupt exception handlers and once served, optee_os returns to the execution trusted thread.

A trusted thread execution can be interrupted by a foreign interrupt. In this case, optee_os suspends the trusted thread and invokes the normal world through the Monitor (optee_os so-called RPC services). The trusted threads will resume only once normal world invokes the optee_os with the RPC service status.

A trusted thread execution can lead optee_os to invoke a service in normal world: access a file, get the REE current time, etc. The trusted thread is suspended/resumed during remote service execution.

Scheduling considerations

When a trusted thread is interrupted by a foreign interrupt and when optee_os invokes a normal world service, the normal world gets the opportunity to reschedule the running applications. The trusted thread will resume only once the client application is scheduled back. Thus, a trusted thread execution follows the scheduling of the normal world caller context.

Optee_os does not implement any thread scheduling. Each trusted thread is expected to track a service that is invoked from the normal world and should return to it with an execution status.

The OP-TEE Linux driver (as implemented in drivers/tee/optee since Linux kernel 4.12) is designed so that the Linux thread invoking OP-TEE gets assigned a trusted thread on TEE side. The execution of the trusted thread is tied to the execution of the caller Linux thread which is under the Linux kernel scheduling decision. This means trusted threads are scheduled by the Linux kernel.

Trusted thread constraints

TEE core handles a static number of trusted threads, see `CFG_NUM_THREADS`.

Trusted threads are only expensive on memory constrained system, mainly regarding the execution stack size.

On SMP systems, optee_os can execute several trusted threads in parallel if the normal world supports scheduling of processes. Even on UP systems, supporting several trusted threads in optee_os helps normal world scheduler to be efficient.

2.1.2 Memory objects

A memory object, **MOBJ**, describes a piece of memory. The interface provided is mostly abstract when it comes to using the MOBJ to populate translation tables etc. There are different kinds of MOBJs describing:

- **Physically contiguous memory**
  - created with `mobj_phys_alloc(...)`.
- **Virtual memory**
  - one instance with the name `mobj_virt` available.
  - spans the entire virtual address space.
- **Physically contiguous memory allocated from a `tee_mm_pool_t *`**

  - created with `mobj_mm_alloc(...)`.

- **Paged memory**
  - created with `mobj_paged_alloc(...)`.
  - only contains the supplied size and makes `mobj_is_paged(...)` return true if supplied as argument.

- **Secure copy paged shared memory**
  - created with `mobj_seccpy_shm_alloc(...)`.
  - makes `mobj_is_paged(...)` and `mobj_is_secure(...)` return true if supplied as argument.

2.1.3 MMU

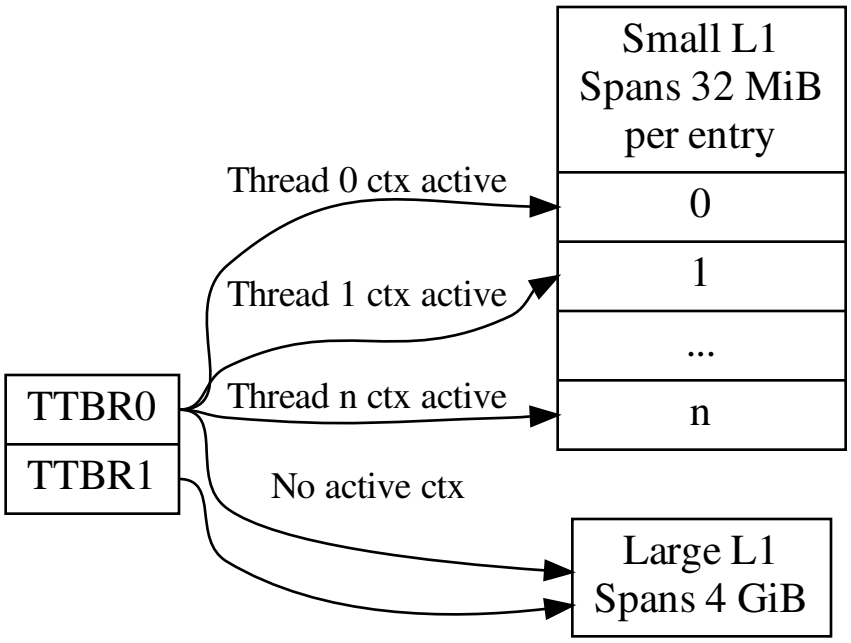
Translation tables

OP-TEE uses several L1 translation tables, one large spanning 4 GiB and two or more small tables spanning 32 MiB. The large translation table handles kernel mode mapping and matches all addresses not covered by the small translation tables. The small translation tables are assigned per thread and covers the mapping of the virtual memory space for one TA context.

Memory space between small and large translation table is configured by TTBRC. TTBR1 always points to the large translation table. TTBR0 points to the a small translation table when user mapping is active and to the large translation table when no user mapping is currently active. For details about registers etc, please refer to a Technical Reference Manual for your architecture, for example Cortex-A53 TRM.

The translation tables has certain alignment constraints, the alignment (of the physical address) has to be the same as the size of the translation table. The translation tables are statically allocated to avoid fragmentation of memory due to the alignment constraints.

Each thread has one small L1 translation table of its own. Each TA context has a compact representation of its L1 translation table. The compact representation is used to initialize the thread specific L1 translation table when the TA context is activated.



Page table cache

Page tables used to map TAs are managed with the page table cache. When the context of a TA is unmapped, all its page tables are released with a call to `pgt_free()`. All page tables needed when mapping a TA are allocated using `pgt_alloc()`.

A fixed maximum number of translation tables are available in a pool. One thread may execute a TA which needs all or almost all tables. This can block TAs from being executed by other threads. To ensure that all TAs eventually will be permitted to execute `pgt_alloc()` temporarily frees eventual tables allocated before waiting for tables to become available.

The page table cache behaves differently depending on configuration options.

Without paging (`CFG_WITH_PAGER=n`)

This is the easiest configuration. All page tables are statically allocated in the `.nozi.pgt_cache` section. `pgt_alloc()` allocates tables from the free-list and `pgt_free()` returns the tables directly to the free-list.

With paging enabled (`CFG_WITH_PAGER=y`)

Page tables are allocated as zero initialized locked pages during boot using `tee_pager_alloc()`. Locked pages are populated with physical pages on demand from the pager. The physical page can be released when not needed any longer with `tee_pager_release_phys()`.

With `CFG_WITH_LPAE=y` each translation table has the same size as a physical page which makes it easy to release the physical page when the translation table isn't needed any longer. With the short-descriptor table format (`CFG_WITH_LPAE=n`) it becomes more complicated as four translation tables are stored in each page. Additional bookkeeping is used to tell when the page for used by four separate translation tables can be released.

With paging of user TA enabled (`CFG_PAGED_USER_TA=y`)

With paging of user TAs enabled a cache of recently used translation tables is used. This can save us from a storm of page faults when restoring the mappings of a recently unmapped TA. Which translation tables should be cached is indicated with reference counting by the pager on used tables. When a table needs to be forcefully freed `tee_pager_pgt_save_and_release_entries()` is called to let the pager know that the table can't be used any longer.

When a mapping in a TA is removed it also needs to be purged from cached tables with `pgt_flush_ctx_range()` to prevent old mappings from being accidentally reused.

Switching to user mode

This section only applies with following configuration flags:

- `CFG_WITH_LPAE=n`
- `CFG_CORE_UNMAP_CORE_AT_EL0=y`

When switching to user mode only a minimal kernel mode mapping is kept. This is achieved by selecting a zeroed out big L1 translation in TTBR1 when transitioning to user mode. When returning back to kernel mode the original L1 translation table is restored in TTBR1.

Switching to normal world

When switching to normal world either via a foreign interrupt (see *Native and foreign interrupts* or RPC there is a chance that secure world will resume execution on a different CPU. This means that the new CPU need to be configured with the context of the currently active TA. This is solved by always setting the TA context in the CPU when resuming execution.

2.1.4 Pager

OP-TEE currently requires >256 KiB RAM for OP-TEE kernel memory. This is not a problem if OP-TEE uses TrustZone protected DDR, but for security reasons OP-TEE may need to use TrustZone protected SRAM instead. The amount of available SRAM varies between platforms, from just a few KiB up to over 512 KiB. Platforms with just a few KiB of SRAM cannot be expected to be able to run a complete TEE solution in SRAM. But those with 128 to 256 KiB of SRAM can be expected to have a capable TEE solution in SRAM. The pager provides a solution to this by demand paging parts of OP-TEE using virtual memory.

Secure memory

TrustZone protected SRAM is generally considered more secure than TrustZone protected DRAM as there is usually more attack vectors on DRAM. The attack vectors are hardware dependent and can be different for different platforms.

Backing store

TrustZone protected DRAM or in some cases non-secure DRAM is used as backing store. The data in the backing store is integrity protected with one hash (SHA-256) per page (4KiB). Readonly pages are not encrypted since the OP-TEE binary itself is not encrypted.

Partitioning of memory

The code that handles demand paging must always be available as it would otherwise lead to deadlock. The virtual memory is partitioned as:

Type	Sections
unpaged	text rodata data bss heap1 nozi heap2
init / paged	text_init rodata_init
paged	text_pageable rodata_pageable
demand alloc	

Where `nozi` stands for "not zero initialized", this section contains entry stacks (thread stack when TEE pager is not enabled) and translation tables (TEE pager cached translation table when the pager is enabled and LPAE MMU is used).

The `init` area is available when OP-TEE is initializing and contains everything that is needed to initialize the pager. After the pager has been initialized this area will be used for demand paged instead.

The `demand alloc` area is a special area where the pages are allocated and removed from the pager on demand. Those pages are returned when OP-TEE does not need them any longer. The thread stacks currently belongs this area. This means that when a stack is not used the physical pages can be used by the pager for better performance.

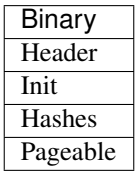
The technique to gather code in the different area is based on compiling all functions and data into separate sections. The unpaged text and rodata is then gathered by linking all object files with `--gc-sections` to eliminate sections that are outside the dependency graph of the entry functions for unpaged functions. A script analyzes this ELF file and generates the bits of the final link script. The process is repeated for init text and rodata. What is not "unpaged" or "init" becomes "paged".

Partitioning of the binary

Note: The struct definitions provided in this section are explicitly covered by the following dual license:

```
SPDX-License-Identifier: (BSD-2-Clause OR GPL-2.0)
```

The binary is partitioned into four parts as:



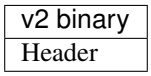
The header is defined as:

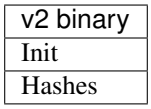
```
#define OPTEE_MAGIC             0x4554504f
#define OPTEE_VERSION           1
#define OPTEE_ARCH_ARM32        0
#define OPTEE_ARCH_ARM64        1

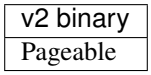
struct optee_header {
        uint32_t magic;
        uint8_t version;
        uint8_t arch;
        uint16_t flags;
        uint32_t init_size;
        uint32_t init_load_addr_hi;
        uint32_t init_load_addr_lo;
        uint32_t init_mem_usage;
        uint32_t paged_size;
};
```

The header is only used by the loader of OP-TEE, not OP-TEE itself. To initialize OP-TEE the loader loads the complete binary into memory and copies what follows the header and the following `init_size` bytes to `(init_load_addr_hi << 32 | init_load_addr_lo)`. `init_mem_usage` is used by the loader to be able to check that there is enough physical memory available for OP-TEE to be able to initialize at all. The loader supplies in `r0/x0` the address of the first byte following what was not copied and jumps to the load address to start OP-TEE.

In addition to overall binary with partitions inside described as above, three extra binaries are generated simultaneously during build process for loaders who support loading separate binaries:







In this case, loaders load header binary first to get image list and information of each image; and then load each of them into specific load address assigned in structure. These binaries are named with *v2* suffix to distinguish from the existing binaries. Header format is updated to help loaders loading binaries efficiently:

```
#define OPTEE_IMAGE_ID_PAGER    0
#define OPTEE_IMAGE_ID_PAGED    1

struct optee_image {
        uint32_t load_addr_hi;
        uint32_t load_addr_lo;
        uint32_t image_id;
        uint32_t size;
};

struct optee_header_v2 {
        uint32_t magic;
        uint8_t version;
        uint8_t arch;
        uint16_t flags;
        uint32_t nb_images;
        struct optee_image optee_image[];
};
```

Magic number and architecture are identical as original. Version is increased to two. `load_addr_hi` and `load_addr_lo` may be `0xFFFFFFFF` for pageable binary since pageable part may get loaded by loader into dynamic available position. `image_id` indicates how loader handles current binary. Loaders who don't support separate loading just ignore all v2 binaries.

Initializing the pager

The pager is initialized as early as possible during boot in order to minimize the "init" area. The global variable `tee_mm_vcore` describes the virtual memory range that is covered by the level 2 translation table supplied to `tee_pager_init(...)`.

Assign pageable areas

A virtual memory range to be handled by the pager is registered with a call to `tee_pager_add_core_area()`.

```
bool tee_pager_add_area(tee_mm_entry_t *mm,
                        uint32_t flags,
                        const void *store,
                        const void *hashes);
```

which takes a pointer to `tee_mm_entry_t` to tell the range, flags to tell how memory should be mapped (readonly, execute etc), and pointers to backing store and hashes of the pages.

Assign physical pages

Physical SRAM pages are supplied by calling `tee_pager_add_pages(...)`

```
void tee_pager_add_pages(tee_vaddr_t vaddr,
                         size_t npages,
                         bool unmap);
```

`tee_pager_add_pages(...)` takes the physical address stored in the entry mapping the virtual address `vaddr` and `npages` entries after that and uses it to map new pages when needed. The unmap parameter tells whether the pages should be unmapped immediately since they does not contain initialized data or be kept mapped until they need to be recycled. The pages in the "init" area are supplied with `unmap == false` since those page have valid content and are in use.

Invocation

The pager is invoked as part of the abort handler. A pool of physical pages are used to map different virtual addresses. When a new virtual address needs to be mapped a free physical page is mapped at the new address, if a free physical page cannot be found the oldest physical page is selected instead. When the page is mapped new data is copied from backing store and the hash of the page is verified. If it is OK the pager returns from the exception to resume the execution.

Data structures

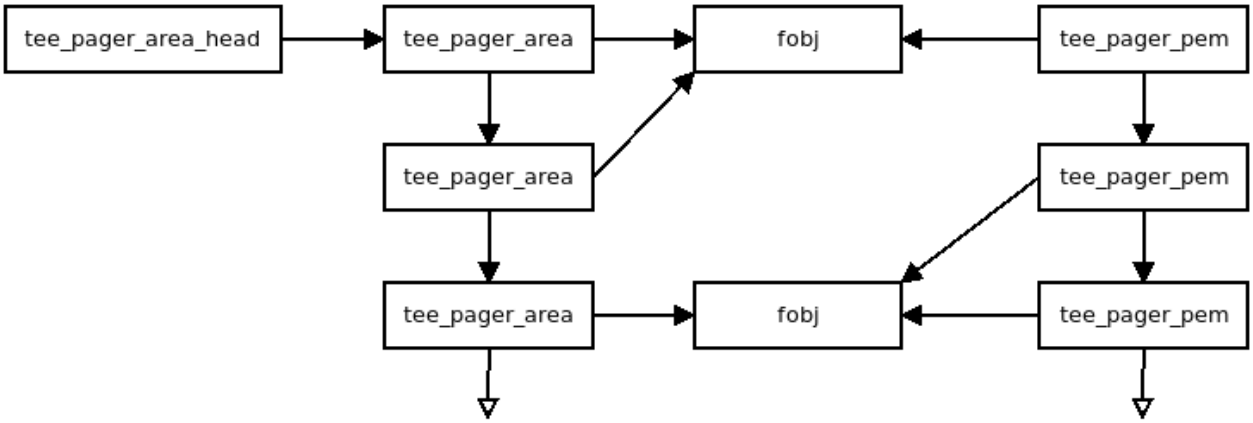


Fig. 5: How the main pager data structures relates to each other

struct tee_pager_area

This is a central data structure when handling paged memory ranges. It's defined as:

```
struct tee_pager_area {
    struct fobj *fobj;
    size_t fobj_pgoffs;
    enum tee_pager_area_type type;
    uint32_t flags;
    vaddr_t base;
    size_t size;
    struct pgt *pgt;
    TAILQ_ENTRY(tee_pager_area) link;
    TAILQ_ENTRY(tee_pager_area) fobj_link;
};
```

Where `base` and `size` tells the memory range and `fobj` and `fobj_pgoffs` holds the content. A `struct tee_pager_area` can only use `struct fobj` and one `struct pgt` (translation table) so memory ranges spanning multiple fobjs or pgts are split into multiple areas.

struct fobj

This is a polymorph object, using different implmentations depending on how it's initialized. It's defines as:

```
struct fobj_ops {
    void (*free)(struct fobj *fobj);
    TEE_Result (*load_page)(struct fobj *fobj, unsigned int page_idx,
                            void *va);
    TEE_Result (*save_page)(struct fobj *fobj, unsigned int page_idx,
                            const void *va);
};

struct fobj {
    const struct fobj_ops *ops;
    unsigned int num_pages;
    struct refcount refc;
    struct tee_pager_area_head areas;
};
```

num_pages Tells how many pages this `fobj` covers.

refc A reference counter, everyone referring to a `fobj` need to increase and decrease this as needed.

areas A list of areas using this `fobj`, traversed when making a virtual page unavailable.

struct tee_pager_pmem

This structure represents a physical page. It's defined as:

```
struct tee_pager_pmem {
    unsigned int flags;
    unsigned int fobj_pgidx;
    struct fobj *fobj;
    void *va_alias;
    TAILQ_ENTRY(tee_pager_pmem) link;
};
```

PMEM_FLAG_DIRTY Bit is set in `flags` when the page is mapped read/write at at least one location.

PMEM_FLAG_HIDDEN Bit is set in `flags` when the page is hidden, that is, not accessible anywhere.

fobj_pgidx The page at this index in the `fobj` is used in this physical page.

fobj The `fobj` backing this page.

va_alias Virtual address where this physical page is updated when loading it from backing store or when writing it back.

All `struct tee_pager_pmem` are stored either in the global list `tee_pager_pmem_head` or in `tee_pager_lock_pmem_head`. The latter is used by pages which are mapped and then locked in memory on demand. The pages are returned back to `tee_pager_pmem_head` when the pages are exlicitly released with a call to `tee_pager_release_phys()`.

A physical page can be used by more than one `tee_pager_area` simultaneously. This is also know as shared secure memory and will appear as such for both read-only and read-write mappings.

When a page is hidden it's unmapped from all translation tables and the `PMEM_FLAG_HIDDEN` bit is set, but kept in memory. When a physical page is released it's also unmapped from all translation tables and it's content is written back to storage, then the `fobj` field is set to `NULL` to note the physical page as unused.

Note that when `struct tee_pager_pmem` references a `fobj` it doesn't update the reference counter since it's already guaranteed to be available due the `struct tee_pager_area` which must reference the `fobj` too.

Paging of user TA

Paging of user TAs can optionally be enabled with `CFG_PAGED_USER_TA=y`. Paging of user TAs is analogous to paging of OP-TEE kernel parts but with a few differences:

- Read/write pages are paged in addition to read-only pages
- Page tables are managed dynamically

`tee_pager_add_uta_area(...)` is used to setup initial read/write mapping needed when populating the TA. When the TA is fully populated and relocated `tee_pager_set_uta_area_attr(...)` changes the mapping of the area to strict permissions used when the TA is running.

Paging shared secure memory

Shared secure memory is achieved by letting several `tee_pager_area` using the same backing `fobj`. When a `tee_pager_area` is allocated and assigned a `fobj` it's also added to a list for `tee_pager_areas` using this `fobj`. This helps when a physical page is released.

When a fault occurs first a matching `tee_pager_area` is located. Then `tee_pager_pmem_head` is searched to see if a physical page already holds the page of the `fobj` needed. If so the `pgt` is updated to map the physical page at the appropriate locatation. If no physical page was holding the page a new physical page is allocated, initialized and finally mapped.

In order to make as few updates to mappings as possible changes to less restricted, no access -> read-only or read-only to read-write, is done only for the virtual address was used when the page fault occurred. Changes in the other direction has to be done in all translation tables used to map the physical page.

2.1.5 Stacks

Different stacks are used during different stages. The stacks are:

- **Secure monitor stack** (128 bytes), bound to the CPU. Only available if OP-TEE is compiled with a secure monitor always the case if the target is Armv7-A but never for Armv8-A.
- **Temp stack** (small ~1KB), bound to the CPU. Used when transitioning from one state to another. Interrupts are always disabled when using this stack, aborts are fatal when using the temp stack.
- **Abort stack** (medium ~2KB), bound to the CPU. Used when trapping a data or pre-fetch abort. Aborts from user space are never fatal the TA is only killed. Aborts from kernel mode are used by the pager to do the demand paging, if pager is disabled all kernel mode aborts are fatal.
- **Thread stack** (large ~8KB), not bound to the CPU instead used by the current thread/task. Interrupts are usually enabled when using this stack.

Notes for Armv7-A/AArch32

Stack	Comment
Temp	Assigned to `SP_SVC` during entry/exit, always assigned to `SP_IRQ` and `SP_FIQ`
Abort	Always assigned to `SP_ABT`
Thread	Assigned to `SP_SVC` while a thread is active

Notes for AArch64 There are only two stack pointers, `SP_EL1` and `SP_EL0`, available for OP-TEE in AArch64. When an exception is received stack pointer is always `SP_EL1` which is used temporarily while assigning an appropriate stack pointer for `SP_EL0`. `SP_EL1` is always assigned the value of `thread_core_local[cpu_id]`. This structure has some spare space for temporary storage of registers and also keeps the relevant stack pointers. In general when we talk about assigning a stack pointer to the CPU below we mean `SP_EL0`.

Boot

During early boot the CPU is configured with the temp stack which is used until OP-TEE exits to normal world the first time.

Notes for AArch64 `SPSEL` is always `0` on entry/exit to have `SP_EL0` acting as stack pointer.

Normal entry

Each time OP-TEE is entered from normal world the temp stack is used as the initial stack. For fast calls, this is the only stack used. For normal calls an empty thread slot is selected and the CPU switches to that stack.

Normal exit

Normal exit occurs when a thread has finished its task and the thread is freed. When the main thread function, `tee_entry_std(...)`, returns interrupts are disabled and the CPU switches to the temp stack instead. The thread is freed and OP-TEE exits to normal world.

RPC exit

RPC exit occurs when OP-TEE need some service from normal world. RPC can currently only be performed with a thread is in running state. RPC is initiated with a call to `thread_rpc(...)` which saves the state in a way that when the thread is restored it will continue at the next instruction as if this function did a normal return. CPU switches to use the temp stack before returning to normal world.

Foreign interrupt exit

Foreign interrupt exit occurs when OP-TEE receives a foreign interrupt. For Arm GICv2 mode, foreign interrupt is sent as IRQ which is always handled in normal world. Foreign interrupt exit is similar to RPC exit but it is `thread_irq_handler(...)` and `elx_irq(...)` (respectively for Armv7-A/Aarch32 and for Aarch64) that saves the thread state instead. The thread is resumed in the same way though. For Arm GICv3 mode, foreign interrupt is sent as FIQ which could be handled by either secure world (EL3 in AArch64) or normal world. This mode is not supported yet.

Notes for Armv7-A/AArch32 SP_IRQ is initialized to temp stack instead of a separate stack. Prior to exiting to normal world CPU state is changed to SVC and temp stack is selected.

Notes for AArch64 `SP_EL0` is assigned temp stack and is selected during IRQ processing. The original `SP_EL0` is saved in the thread context to be restored when resuming.

Resume entry

OP-TEE is entered using the temp stack in the same way as for normal entry. The thread to resume is looked up and the state is restored to resume execution. The procedure to resume from an RPC exit or an foreign interrupt exit is exactly the same.

Syscall

Syscall's are executed using the thread stack.

Notes for Armv7-A/AArch32 Nothing special `SP_SVC` is already set with thread stack.

Notes for syscall AArch64 Early in the exception processing the original `SP_EL0` is saved in `struct thread_svc_regs` in case the TA is executed in AArch64. Current thread stack is assigned to `SP_EL0` which is then selected. When returning `SP_EL0` is assigned what is in `struct thread_svc_regs`. This allows `tee_svc_sys_return_helper(...)` having the syscall exception handler return directly to `thread_unwind_user_mode(...)`.

2.1.6 Shared Memory

Shared Memory is a block of memory that is shared between the non-secure and the secure world. It is used to transfer data between both worlds.

The shared memory is allocated and managed by the non-secure world, i.e. the Linux OP-TEE driver. Secure world only considers the individual shared buffers, not their pool. Each shared memory is referenced with associated attributes:

- Buffer start address and byte size,
- Cache attributes of the shared memory buffer,
- List of chunks if mapped from noncontiguous pages.

Shared memory buffer references manipulated must fit inside one of the shared memory areas known from the OP-TEE core. OP-TEE supports two kinds of shared memory areas: an area for contiguous buffers and an area for noncontiguous buffers. At least one has to be enabled.

Contiguous shared buffers

Configuration directives `CFG_SHMEM_START` and `CFG_SHMEM_SIZE` define a share memory area where shared memory buffers are contiguous. Generic memory layout registers it as the `MEM_AREA_NSEC_SHM` memory area.

The non-secure world issues `OPTEE_SMC_GET_SHM_CONFIG` to retrieve contiguous shared memory area configuration:

- Physical address of the start of the pool
- Size of the pool
- Whether or not the memory is cached

Contiguous shared memory (also known as static or reserved shared memory) is enabled with the configuration flag `CFG_CORE_RESERVED_SHM=y`.

Noncontiguous shared buffers

To benefit from noncontiguous shared memory buffers, secure world register dynamic shared memory areas and non-secure world must register noncontiguous buffers prior to referring to them using the OP-TEE API.

The OP-TEE core generic boot sequence discovers dynamic shared areas from the device tree and/or areas explicitly registered by the platform.

Non-secure side needs to register buffers as 4kByte chunks lists into OP-TEE core using the `OPTEE_MSG_CMD_REGISTER_SHM` API prior referencing to them using the OP-TEE invocation API.

Noncontiguous shared memory (also known as dynamic shared memory) is enabled with the configuration flag `CFG_CORE_DYN_SHM=y`.

For performance reasons, the TEE Client Library (`libteec`) uses noncontiguous shared memory when available since it avoids copies in some situations.

Shared Memory Chunk Allocation

It is the Linux kernel driver for OP-TEE that is responsible for allocating chunks of shared memory. OP-TEE linux kernel driver relies on linux kernel generic allocation support (`CONFIG_GENERIC_ALLOCATION`) to allocation/release of shared memory physical chunks. OP-TEE linux kernel driver relies on linux kernel dma-buf support (`CONFIG_DMA_SHARED_BUFFER`) to track shared memory buffers references.

Using shared memory

From the Client Application The client application can ask for shared memory allocation using the GlobalPlatform Client API function `TEEC_AllocateSharedMemory(...)`. The client application can also register a memory through the GlobalPlatform Client API function `TEEC_RegisterSharedMemory(...)`. The shared memory reference can then be used as parameter when invoking a trusted application.

From the Linux Driver Occasionally the Linux kernel driver needs to allocate shared memory for the communication with secure world, for example when using buffers of type `TEEC_TempMemoryReference`.

From OP-TEE core In case OP-TEE core needs information from TEE supplicant (dynamic TA loading, REE time request,...), shared memory must be allocated. Allocation depends on the use case. OP-TEE core asks for the following shared memory allocation:

- `optee_msg_arg` structure, used to pass the arguments to the non-secure world, where the allocation will be done by sending a `OPTEE_SMC_RPC_FUNC_ALLOC` message.
- In some cases, a payload might be needed for storing the result from TEE supplicant, for example when loading a Trusted Application. This type of allocation will be done by sending the message `OPTEE_MSG_RPC_CMD_SHM_ALLOC(OPTEE_MSG_RPC_SHM_TYPE_APPL,...)`, which then will return:
  - the physical address of the shared memory
  - a handle to the memory, that later on will be used later on when freeing this memory.

From TEE Supplicant TEE supplicant is also working with shared memory, used to exchange data between normal and secure worlds. TEE supplicant receives a memory address from the OP-TEE core, used to store the data. This is for example the case when a Trusted Application is loaded. In this case, TEE supplicant must register the provided shared memory in the same way a client application would do, involving the Linux driver.

2.1.7 SMC

SMC Interface

OP-TEE's SMC interface is defined in two levels using optee_smc.h and optee_msg.h. The former file defines SMC identifiers and what is passed in the registers for each SMC. The latter file defines the OP-TEE Message protocol which is not restricted to only SMC even if that currently is the only option available.

SMC communication

The main structure used for the SMC communication is defined in `struct optee_msg_arg` (in optee_msg.h). If we are looking into the source code, we could see that communication mainly is achieved using `optee_msg_arg` and `thread_smc_args` (in thread.h), where `optee_msg_arg` could be seen as the main structure. What will happen is that the *Linux kernel TEE framework* driver will get the parameters either from *optee_client* or directly from an internal service in Linux kernel. The TEE driver will populate the struct `optee_msg_arg` with the parameters plus some additional bookkeeping information. Parameters for the SMC are passed in registers 1 to 7, register 0 holds the SMC id which among other things tells whether it is a standard or a fast call.

2.1.8 Thread handling

OP-TEE core uses a couple of threads to be able to support running jobs in parallel (not fully enabled!). There are handlers for different purposes. In thread.c you will find a function called `thread_init_primary(...)` which assigns `init_handlers` (functions) that should be called when OP-TEE core receives standard or fast calls, FIQ and PSCI calls. There are default handlers for these services, but the platform can decide if they want to implement their own platform specific handlers instead.

Synchronization primitives

OP-TEE has three primitives for synchronization of threads and CPUs: *spin-lock*, *mutex*, and *condvar*.

Spin-lock A spin-lock is represented as an `unsigned int`. This is the most primitive lock. Interrupts should be disabled before attempting to take a spin-lock and should remain disabled until the lock is released. A spin-lock is initialized with `SPINLOCK_UNLOCK`.

Table 1: Spin lock functions

Function	Purpose
`cpu_spin_lock(..)`	Locks a spin-lock
`cpu_spin_trylock(..)`	Locks a spin-lock if unlocked and returns `0` else the spin-lock is unchanged and the function returns `!0`
`cpu_spin_unlock(..)`	Unlocks a spin-lock

Mutex A mutex is represented by `struct mutex`. A mutex can be locked and unlocked with interrupts enabled or disabled, but only from a normal thread. A mutex cannot be used in an interrupt handler, abort handler or before a thread has been selected for the CPU. A mutex is initialized with either `MUTEX_INITIALIZER` or `mutex_init(...)`.

Table 2: Mutex functions

Function	Purpose
`mutex_lock(..)`	Locks a mutex. If the mutex is unlocked this is a fast operation, else the function issues an RPC to wait in normal world.
`mutex_unlock(..)`	Unlocks a mutex. If there is no waiters this is a fast operation, else the function issues an RPC to wake up a waiter in normal world.
`mutex_trylock(..)`	Locks a mutex if unlocked and returns `true` else the mutex is unchanged and the function returns `false`.
`mutex_destroy(..)`	Asserts that the mutex is unlocked and there is no waiters, after this the memory used by the mutex can be freed.

When a mutex is locked it is owned by the thread calling `mutex_lock(...)` or `mutex_trylock(...)`, the mutex may only be unlocked by the thread owning the mutex. A thread should not exit to TA user space when holding a mutex.

Condvar A condvar is represented by `struct condvar`. A condvar is similar to a `pthread_condvar_t` in the pthreads standard, only less advanced. Condition variables are used to wait for some condition to be fulfilled and are always used together a mutex. Once a condition variable has been used together with a certain mutex, it must only be used with that mutex until destroyed. A condvar is initialized with `CONDVAR_INITIALIZER` or `condvar_init(...)`.

Table 3: Condvar functions

Function	Purpose
`condvar_wait(...)`	Atomically unlocks the supplied mutex and waits in normal world via an RPC for the condition variable to be signaled, when the function returns the mutex is locked again.
`condvar_signal(...)`	Wakes up one waiter of the condition variable (waiting in `condvar_wait(...)`).
`condvar_broadcast(...)`	Wake up all waiters of the condition variable.

The caller of `condvar_signal(...)` or `condvar_broadcast(...)` should hold the mutex associated with the condition variable to guarantee that a waiter does not miss the signal.

2.2 Cryptographic implementation

This document describes how the TEE Cryptographic Operations API is implemented, how the default crypto provider may be configured at compile time, and how it may be replaced by another implementation.

2.2.1 Overview

There are several layers from the Trusted Application to the actual crypto algorithms. Most of the crypto code runs in kernel mode inside the TEE core. Here is a schematic view of a typical call to the crypto API. The numbers in square brackets ([1], [2]...) refer to the sections below.

```
-   some_function()                                 (Trusted App) -
[1]   TEE_*()                       User space      (libutee.a)
------- utee_*() ---------------------------------------------
[2]       tee_svc_*()               Kernel space
[3]         crypto_*()                              (libtomcrypt.a and crypto.c)
[4]           /* LibTomCrypt */                     (libtomcrypt.a)
```

2.2.2 [1] The TEE Cryptographic Operations API

OP-TEE implements the Cryptographic Operations API defined by the GlobalPlatform association in the *TEE Internal Core API*. This includes cryptographic functions that span various cryptographic needs: message digests, symmetric ciphers, message authentication codes (MAC), authenticated encryption, asymmetric operations (encryption/decryption or signing/verifying), key derivation, and random data generation. These functions make up the TEE Cryptographic Operations API.

The Internal API is implemented in tee_api_operations.c, which is compiled into a static library: `${O}/ta_arm{32,64}-lib/libutee/libutee.a`.

Most API functions perform some parameter checking and manipulations, then invoke some *utee_** function to switch to kernel mode and perform the low-level work.

The *utee_** functions are declared in utee_syscalls.h and implemented in utee_syscalls_asm.S They are simple system call wrappers which use the *SVC* instruction to switch to the appropriate system service in the OP-TEE kernel.

2.2.3 [2] The crypto services

All cryptography-related system calls are declared in tee_svc_cryp.h and implemented in tee_svc_cryp.c. In addition to dealing with the usual work required at the user/kernel interface (checking parameters and copying memory buffers between user and kernel space), the system calls invoke a private abstraction layer: the **Crypto API**, which is declared in crypto.h. It serves two main purposes:

1. Allow for alternative implementations, such as hardware-accelerated versions.
2. Provide an easy way to disable some families of algorithms at compile-time to save space. See *LibTomCrypt* below.

2.2.4 [3] crypto_*()

The `crypto_*()` functions implement the actual algorithms and helper functions. TEE Core has one global active implementation of this interface. The default implementation, mostly based on LibTomCrypt, is as follows:

Listing 1: File: core/crypto/crypto.c

```
/*
 * Default implementation for all functions in crypto.h
 */

#if !defined(_CFG_CRYPTO_WITH_HASH)
TEE_Result crypto_hash_get_ctx_size(uint32_t algo __unused,
                                    size_t *size __unused)
{
        return TEE_ERROR_NOT_IMPLEMENTED;
}
...
#endif /*_CFG_CRYPTO_WITH_HASH*/
```

Listing 2: File: core/lib/libtomcrypt/tee_ltc_provider.c

```
#if defined(_CFG_CRYPTO_WITH_HASH)
TEE_Result crypto_hash_get_ctx_size(uint32_t algo, size_t *size)
{
    /* ... */
    return TEE_SUCCESS;
}

#endif /*_CFG_CRYPTO_WITH_HASH*/
```

As shown above, families of algorithms can be disabled and crypto.c will provide default null implementations that will return `TEE_ERROR_NOT_IMPLEMENTED`.

2.2.5 Public/private key format

crypto.h uses implementation-specific types to hold key data for asymmetric algorithms. For instance, here is how a

public RSA key is represented:

Listing 3: File: core/include/crypto/crypto.h

```
struct rsa_public_key {
    struct bignum *e;        /* Public exponent */
    struct bignum *n;        /* Modulus */
};
```

This is also how such keys are stored inside the TEE object attributes (`TEE_ATTR_RSA_PUBLIC_KEY` in this case). `struct bignum` is an opaque type, known to the underlying implementation only. `struct bignum_ops` provides functions so that the system services can manipulate data of this type. This includes allocation/deallocation, copy, and conversion to or from the big endian binary format.

Listing 4: File: core/include/crypto/crypto.h

```
struct bignum *crypto_bignum_allocate(size_t size_bits);

TEE_Result crypto_bignum_bin2bn(const uint8_t *from, size_t fromsize,
                struct bignum *to);

void crypto_bignum_bn2bin(const struct bignum *from, uint8_t *to);
/*...*/
```

2.2.6 [4] LibTomCrypt

Some algorithms may be disabled at compile time if they are not needed, in order to reduce the size of the OP-TEE image and reduces its memory usage. This is done by setting the appropriate configuration variable. For example:

```
$ make CFG_CRYPTO_AES=n               # disable AES only
$ make CFG_CRYPTO_{AES,DES}=n         # disable symmetric ciphers
$ make CFG_CRYPTO_{DSA,RSA,DH,ECC}=n # disable public key algorithms
$ make CFG_CRYPTO=n                   # disable all algorithms
```

Please refer to core/lib/libtomcrypt/sub.mk for the list of all supported variables.

Note that the application interface is **not** modified when algorithms are disabled. This means, for instance, that the functions `TEE_CipherInit()`, `TEE_CipherUpdate()` and `TEE_CipherFinal()` would remain present in `libutee.a` even if all symmetric ciphers are disabled (they would simply return `TEE_ERROR_NOT_IMPLEMENTED`).

2.2.7 Add a new crypto implementation

To add a new implementation, the default one in core/lib/libtomcrypt in combination with what is in core/crypto should be used as a reference. Here are the main things to consider when adding a new crypto provider:

- Put all the new code in its own directory under `core/lib` unless it is code that will be used regardless of which crypto provider is in use. How we are dealing with AES-GCM in core/crypto could serve as an example.
- Avoid modifying tee_svc_cryp.c. It should not be needed.
- Although not all crypto families need to be defined, all are required for compliance to the GlobalPlatform specification.
- If you intend to make some algorithms optional, please try to re-use the same names for configuration variables as the default implementation.

2.3 Device Tree

OP-TEE core can use the device tree format to inject platform configuration information during platform initialization and possibly some run time contexts.

Device Tree technology allows to describe platforms from ASCII source files so-called DTS files. These can be used to generate a platform description binary image, so-called DTB, embedded in the platform boot media for applying expected configuration settings during the platform initializations.

This scheme relaxes design constraints on the OP-TEE core implementation as most of the platform specific hardware can be tuned without modifying C source files or adding configuration directives in the build environments.

2.3.1 Secure and Non-Secure Device Trees

There can be several device trees embedded in the target system and some can be shared across the boot stages.

- Boot loader stages may load a device tree structure in memory for all boot stage to get platform configuration from. If such device tree data are to be accessed by the non-secure world, they shall be located in non-secure memory. Secure world may use its content during OP-TEE core initialization.
- Boot loader stages may load a device tree structure in secure memory for the benefit of the secure world only. Such device tree blob shall be located in secure memory. Secure world could use its content but this is currently not implemented in the latest OP-TEE release.
- OP-TEE core can also embedded a device tree structure to describe the platform.
- Non-secure world can embed its own device tree structure(s) and/or rely on a device tree structure loaded by the secure world during its initialization which happen before non-secure world is booted.

Obviously the non-secure world will not be able to access a device tree image located in a secure memory which non-secure world has no access to.

When OP-TEE core is built with `CFG_DT=y`, non-secure and secure device trees can be accessed by OP-TEE core to get some platform configuration information.

2.3.2 Generic boot and DTBs

Generic boot sequence gets discovers main memory address ranges from preferrably embedded DTB (section *Embedded Secure Device Tree*), defaulting to early boot external DTB (section *Early boot external device tree*).

Generic boot uses early boot external DTB (section *Early boot external device tree*) to share platform configuration information with the non-secure world.

Plaform and drivers can call OP-TEE DT API (`core/include/kernel/dt.h`) to access embedded and/or external DTBs.

2.3.3 Early boot external device tree

The bootloader provides arguments to OP-TEE core when it boots it. Among those, the physical memory base address of a non-secure device tree image accessible to OP-TEE core, or a null address value in absence of such DTB.

Platform configuration may statically define such DTB location using the build configuration directive `CFG_DT_ADDR`.

When an external DTB is referred, OP-TEE core gets the console configuration if the platform has registered a compatible driver by adding attribute `__dt_driver` to a defined `const struct dt_driver` instance.

When an external DTB is referred, OP-TEE core adds into this DTB the description of some OP-TEE resources. These information can be used by the non-secure world to properly communicate with OP-TEE. This scheme assumes the image is located in non-secure memory.

Modifications made by OP-TEE core on the non-secure device tree image provided by early boot and passed to non-secure world are the following:

- Add an OP-TEE node if none found with the related invocation parameters.
- Add a reserved memory node for the few memory areas that shall be reserved to the secure world and non accessed by the non-secure world.
- Add a PSCI description node if none found.

Early boot DTB can be accessed by OP-TEE core only during its initialization, before non-secure world boots as it is expected the DTB memory location has likely been replaced with runtime contexts content.

Assuming there is no embedded DTB (section *Embedded Secure Device Tree*) OP-TEE core discovers the main memory address ranges from the non-secure DTB.

2.3.4 Early boot device tree overlay

There are two possibilities for OP-TEE core to provide a device tree overlay to the non-secure world.

- Append OP-TEE nodes to an existing DTB overlay located in early boot DTB. (`CFG_DT_ADDR` or boot argument register `R2`/`X2`).
- Generate a new DTB overlay image at location defined by `CFG_DT_ADDR`.

In the later case, memory referred by configuration directive `CFG_DT_ADDR` shall not contain a valid DTB image when OP-TEE core is booted. A subsequent non-secure boot stage should merge the OP-TEE DTB overlay image into another DTB.

A typical bootflow for this would be Trusted Firmware-A -> OP-TEE -> U-Boot with U-Boot in charge of merging OP-TEE DTB overlay located at `CFG_DT_ADDR` into a DTB U-Boot has loaded from elsewhere.

This functionality is enabled when `CFG_EXTERNAL_DTB_OVERLAY=y`.

2.3.5 Embedded Secure Device Tree

When OP-TEE core is built with configuration directive `CFG_EMBED_DTB=y`, directive `CFG_EMBED_DTB_SOURCE_FILE` shall provide the relative path of the DTS file inside directory `core/arch/$(ARCH)/dts` from which a DTB is generated and embedded in a read-only section of OP-TEE core.

Refer to `core/include/kernel/dt.h` for API to access embedded DTB.

Section *Generic boot and DTBs* documents the generic boot sequence against embedded DTB.

2.4 File structure

This page describes what different folders in *optee_os* contains.

2.4.1 Top level directories

Table 4: Top level directories

Directory	Description
/core	Files that are only used building TEE Core
/lib	Files that are used both when building TEE Core and TAs
/ta	Files that are only used when building TAs
/mk	Makefiles supporting the build system
/tmp-stuff	Temporary stuff that will be removed before the final commit is made
/scripts	Helper scripts for miscellaneous tasks
/out	Created when building unless a different out directory is specified with `O=...` on the command line

2.4.2 /core

Table 5: Structure of /core

Directory	Description
/arch	Architecture and platform specific files
/include	Header files of resources exported by the core
/lib	Generic libraries that are likely to be replaced in a final product
/mm	Generic memory management, currently empty
/tee	Generic TEE files

2.4.3 /core/arch

Table 6: Structure of /core/arch

Directory	Description
/arm	ARMv7 and Aarch32 specific architecture and platform specific files

2.4.4 /core/arch/arm

Table 7: Structure of /core/arch/arm

Directory	Description
/dts	Device tree source files
/include	Include files used in rest of TEE core but not in any supporting libraries
/kern	Low level and core parts of TEE Core
/mm	Memory management
/tee	TEE files
/sm	Secure Monitor
/plat-foo	Specific files for the `foo` platform

2.4.5 /core/arch/arm/include

Table 8: Structure of /core/arch/arm/include

Directory	Description
/kern	Include files exposing API for /core/arch/arm/kern files
/kta	Include files exposing the KTA API that is mainly used by kernel TAs
/mm	Include files exposing API for /core/arch/arm/mm files
/rom	Old ROM files that should be removed before going public
/sm	Include files exposing API for Secure Monitor

2.4.6 /core/include

Table 9: Structure of /core/include

Directory	Description
/drivers	Include files exposing API for /core/drivers files
/dt-bindings	Include files for the device tree bindings

2.4.7 /core/lib/lib{crypto,sla}

Table 10: Structure of /core/lib/lib{crypto,sla}

Directory	Description
/	Source files for the library
/include	Include files exposing the API of the library

2.4.8 /lib/libutils

Table 11: Structure of /lib/libutils

Directory	Description
/	Source file for the library
/arch	Architecture specific source files
/arch/arm	ARMv7 and Aarch32 specific source files
/arch/arm/include	ARMv7 and Aarch32 specific include files
/include	Include files exposing the API of the library

2.5 GlobalPlatform API

2.5.1 Introduction

GlobalPlatform works across industries to identify, develop and publish specifications which facilitate the secure and interoperable deployment and management of multiple embedded applications on secure chip technology. OP-TEE has support for GlobalPlatform TEE Client API Specification v1.0 (GPD_SPE_007) and TEE Internal Core API Specification v1.1.2 (GPD_SPE_010).

2.5.2 TEE Client API

The TEE Client API describes and defines how a client running in a rich operating environment (REE) should communicate with the TEE. To identify a Trusted Application (TA) to be used, the client provides an UUID. All TA's exposes one or several functions. Those functions corresponds to a so called `commandID` which also is sent by the client.

TEE Contexts

The TEE Context is used for creating a logical connection between the client and the TEE. The context must be initialized before the TEE Session can be created. When the client has completed a job running in secure world, it should finalize the context and thereby also release resources.

TEE Sessions

Sessions are used to create logical connections between a client and a specific Trusted Application. When the session has been established the client has opened up the communication channel towards the specified Trusted Application identified by the `UUID`. At this stage the client and the Trusted Application can start to exchange data.

TEE Client API example / usage

Below you will find the main functions as defined by GlobalPlatform and are used in the communication between the client and the TEE.

```
TEEC_Result TEEC_InitializeContext(
    const char* name,
    TEEC_Context* context)

void TEEC_FinalizeContext(
    TEEC_Context* context)

TEEC_Result TEEC_OpenSession (
    TEEC_Context* context,
    TEEC_Session* session,
    const TEEC_UUID* destination,
    uint32_t connectionMethod,
    const void* connectionData,
    TEEC_Operation* operation,
    uint32_t* returnOrigin)

void TEEC_CloseSession (
    TEEC_Session* session)

TEEC_Result TEEC_InvokeCommand(
    TEEC_Session* session,
    uint32_t commandID,
    TEEC_Operation* operation,
    uint32_t* returnOrigin)
```

In principle the commands are called in this order:

```
TEEC_InitializeContext(...)
TEEC_OpenSession(...)
TEEC_InvokeCommand(...)
```

(continues on next page)

(continued from previous page)

```
TEEC_CloseSession(...)
TEEC_FinalizeContext(...)
```

It is not uncommon that `TEEC_InvokeCommand(...)` is called several times in a row when the session has been established.

For a complete example, please see chapter **5.2 Example 1: Using the TEE Client API** in the GlobalPlatform TEE Client API Specification v1.0.

2.5.3 TEE Internal Core API

The Internal Core API is the API that is exposed to the Trusted Applications running in the secure world. The TEE Internal API consists of four major parts:

1. Trusted Storage API for Data and Keys
2. Cryptographic Operations API
3. Time API
4. Arithmetical API

Examples / usage

Calling the Internal Core API is done in the same way as described above using Client API. The best place to find information how this should be done is in the TEE Internal Core API Specification v1.1.2 which contains many examples of how to call the various APIs. One can also have a look at the examples in the optee_examples git.

2.5.4 Extensions

In addition to what is stated in *TEE Internal Core API*, there are some non-official extensions in OP-TEE.

Trusted Applications should include header file `tee_api_defines_extensions.h` to import the definitions of the extensions. For each extension, a configuration directive prefixed `CFG_` allows one to disable support for the extension when building the OP-TEE packages.

Cache Maintenance Support

Following functions have been introduced in order to allow Trusted Applications to operate with the data cache:

```
TEE_Result TEE_CacheClean(char *buf, size_t len);
TEE_Result TEE_CacheFlush(char *buf, size_t len);
TEE_Result TEE_CacheInvalidate(char *buf, size_t len);
```

These functions are available to any Trusted Application defined with the flag `TA_FLAG_CACHE_MAINTENANCE` sets on, see *Cache maintenance Flag*. When not set, each function returns the error code `TEE_ERROR_NOT_SUPPORTED`. Within these extensions, a Trusted Application is able to operate on the data cache, with the following specification:

Function	Description
TEE_CacheClean()	Write back to memory any dirty data cache lines. The line is marked as not dirty. The valid bit is unchanged.
TEE_CacheFlush()	Purge any valid data cache lines. Any dirty cache lines are first written back to memory, then the cache line is invalidated.
TEE_CacheInvalidate()	Invalidate any valid data cache lines. Any dirty line are not written back to memory.

In the following two cases, the error code TEE_ERROR_ACCESS_DENIED is returned:

- The memory range has not the write access, that is TEE_MEMORY_ACCESS_WRITE is not set.
- The memory is **not** user space memory.

You may disable this extension by setting the following configuration variable in conf.mk:

```
CFG_CACHE_API := n
```

PKCS#1 v1.5 RSASSA without hash OID

This extension adds identifer‘‘TEE_ALG_RSASSA_PKCS1_V1_5‘‘ to allow signing and verifying messages with RSASSA-PKCS1-v1_5, in RFC 3447, without including the OID of the hash in the signature. You may disable this extension by setting the following configuration variable in conf.mk:

```
CFG_CRYPTO_RSASSA_NA1 := n
```

The TEE Internal Core API was extended with a new algorithm descriptor.

Algorithm	Possible Modes
TEE_ALG_RSASSA_PKCS1_V1_5	TEE_MODE_SIGN / TEE_MODE_VERIFY

Algorithm	Identifier
TEE_ALG_RSASSA_PKCS1_V1_5	0x70002830

Concat KDF

Support for the Concatenation Key Derivation Function (Concat KDF) according to SP 800-56A (*Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*) can be found in OP-TEE. You may disable this extension by setting the following configuration variable in conf.mk:

```
CFG_CRYPTO_CONCAT_KDF := n
```

Implementation notes

All key and parameter sizes **must** be multiples of 8 bits. That is:

- Input parameters: the shared secret (Z) and OtherInfo.
- Output parameter: the derived key (DerivedKeyingMaterial).

In addition, the maximum size of the derived key is limited by the size of an object of type TEE_TYPE_GENERIC_SECRET (512 bytes). This implementation does **not** enforce any requirement on the content of the OtherInfo parameter. It is the application's responsibility to make sure this parameter is constructed as specified by the NIST specification if compliance is desired.

API extension

To support Concat KDF, the *TEE Internal Core API* v1.1 was extended with new algorithm descriptors, new object types, and new object attributes as described below.

p.95 Add new object type to TEE_PopulateTransientObject

The following entry shall be added to **Table 5-8**:

Object type	Parts
TEE_TYPE_CONCAT_KDF_Z	The TEE_ATTR_CONCAT_KDF_Z part (input shared secret) must be provided.

p.121 Add new algorithms for TEE_AllocateOperation

The following entry shall be added to **Table 6-3**:

Algorithm	Possible Modes
TEE_ALG_CONCAT_KDF_SHA1_DERIVE_KEY	TEE_MODE_DERIVE
TEE_ALG_CONCAT_KDF_SHA224_DERIVE_KEY	
TEE_ALG_CONCAT_KDF_SHA256_DERIVE_KEY	
TEE_ALG_CONCAT_KDF_SHA384_DERIVE_KEY	
TEE_ALG_CONCAT_KDF_SHA512_DERIVE_KEY	
TEE_ALG_CONCAT_KDF_SHA512_DERIVE_KEY	

p.126 Explain usage of HKDF algorithms in TEE_SetOperationKey

In the bullet list about operation mode, the following shall be added:

- For the Concat KDF algorithms, the only supported mode is `TEE_MODE_DERIVE`.

p.150 Define TEE_DeriveKey input attributes for new algorithms

The following sentence shall be deleted:

```
The TEE_DeriveKey function can only be used with the algorithm
TEE_ALG_DH_DERIVE_SHARED_SECRET.
```

The following entry shall be added to **Table 6-7**:

Algorithm	Possible operation parameters
TEE_ALG_CONCAT_KDF_SHA1_DERIVE_KEY TEE_ALG_CONCAT_KDF_SHA224_DERIVE_KEY TEE_ALG_CONCAT_KDF_SHA256_DERIVE_KEY TEE_ALG_CONCAT_KDF_SHA384_DERIVE_KEY TEE_ALG_CONCAT_KDF_SHA512_DERIVE_KEY TEE_ALG_CONCAT_KDF_SHA512_DERIVE_KEY	TEE_ATTR_CONCAT_KDF_DKM_LENGTH: up to 512 bytes. This parameter is mandatory: TEE_ATTR_CONCAT_KDF_OTHER_INFO

p.152 Add new algorithm identifiers

The following entries shall be added to **Table 6-8**:

Algorithm	Identifier
TEE_ALG_CONCAT_KDF_SHA1_DERIVE_KEY	0x800020C1
TEE_ALG_CONCAT_KDF_SHA224_DERIVE_KEY	0x800030C1
TEE_ALG_CONCAT_KDF_SHA256_DERIVE_KEY	0x800040C1
TEE_ALG_CONCAT_KDF_SHA384_DERIVE_KEY	0x800050C1
TEE_ALG_CONCAT_KDF_SHA512_DERIVE_KEY	0x800060C1

p.154 Define new main algorithm

In **Table 6-9** in section 6.10.1, a new value shall be added to the value column for row bits `[7:0]`:

Bits	Function	Value
Bits [7:0]	Identifiy the main underlying algorithm itself	... 0xC1: Concat KDF

The function column for `bits[15:12]` shall also be modified to read:

Bits	Function	Value
Bits [15:12]	Define the message digest for asymmetric signature algorithms or Concat KDF	

p.155 Add new object type for Concat KDF input shared secret

The following entry shall be added to **Table 6-10**:

Name	Identifier	Possible sizes
TEE_TYPE_CONCAT_KDF_Z	0xA10000C1	8 to 4096 bits (multiple of 8)

p.156 Add new operation attributes for Concat KDF

The following entries shall be added to **Table 6-11**:

Name	Value	Protection	Type	Comment
TEE_ATTR_CONCAT_KDF_Z	0xC0000121	Protected	Ref	The shared secret (`Z`)
TEE_ATTR_CONCAT_KDF_OTHER_INFO	0xD0000122	Public	Ref	`OtherInfo`
TEE_ATTR_CONCAT_KDF_DERIVED_KEY_LENGTH	0xF0000123	Public	Value	The length (in bytes) of the derived keying material to be generated, maximum 512. This is `KeyDataLen` / 8.

HKDF

OP-TEE implements the *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* as specified in RFC 5869. This file documents the extensions to the *TEE Internal Core API* v1.1 that were implemented to support this algorithm. Trusted Applications should include `<tee_api_defines_extensions.h>` to import the definitions.

Note that the implementation follows the recommendations of version 1.1 of the specification for adding new algorithms. It should make it compatible with future changes to the official specification. You can disable this extension by setting the following in `conf.mk`:

```
CFG_CRYPTO_HKDF := n
```

p.95 Add new object type to TEE_PopulateTransientObject

The following entry shall be added to **Table 5-8**:

Object type	Parts
TEE_TYPE_HKDF_IKM	The TEE_ATTR_HKDF_IKM (Input Keying Material) part must be provided.

p.121 Add new algorithms for TEE_AllocateOperation

The following entry shall be added to **Table 6-3**:

Algorithm		Possible Modes
TEE_ALG_HKDF_MD5_DERIVE_KEY	TEE_ALG_HKDF_SHA1_DERIVE_KEY	TEE_MODE_DERIVE
TEE_ALG_HKDF_SHA224_DERIVE_KEY	TEE_ALG_HKDF_SHA256_DERIVE_KEY	
TEE_ALG_HKDF_SHA384_DERIVE_KEY	TEE_ALG_HKDF_SHA512_DERIVE_KEY	
TEE_ALG_HKDF_SHA512_DERIVE_KEY		

p.126 Explain usage of HKDF algorithms in TEE_SetOperationKey

In the bullet list about operation mode, the following shall be added:

- For the HKDF algorithms, the only supported mode is TEE_MODE_DERIVE.

p.150 Define TEE_DeriveKey input attributes for new algorithms

The following sentence shall be deleted:

```
The TEE_DeriveKey function can only be used with the algorithm
TEE_ALG_DH_DERIVE_SHARED_SECRET
```

The following entry shall be added to **Table 6-7**:

Algorithm	Possible operation parameters
TEE_ALG_HKDF_MD5_DERIVE_KEY TEE_ALG_HKDF_SHA1_DERIVE_KEY TEE_ALG_HKDF_SHA224_DERIVE_KEY TEE_ALG_HKDF_SHA256_DERIVE_KEY TEE_ALG_HKDF_SHA384_DERIVE_KEY TEE_ALG_HKDF_SHA512_DERIVE_KEY TEE_ALG_HKDF_SHA512_DERIVE_KEY	TEE_ATTR_HKDF_OKM_LENGTH: Number of bytes in the Output Keying Material TEE_ATTR_HKDF_SALT (optional) Salt to be used during the extract step TEE_ATTR_HKDF_INFO (optional) Info to be used during the expand step

p.152 Add new algorithm identifiers

The following entries shall be added to **Table 6-8**:

Algorithm	Identifier
TEE_ALG_HKDF_MD5_DERIVE_KEY	0x800010C0
TEE_ALG_HKDF_SHA1_DERIVE_KEY	0x800020C0
TEE_ALG_HKDF_SHA224_DERIVE_KEY	0x800030C0
TEE_ALG_HKDF_SHA256_DERIVE_KEY	0x800040C0
TEE_ALG_HKDF_SHA384_DERIVE_KEY	0x800050C0
TEE_ALG_HKDF_SHA512_DERIVE_KEY	0x800060C0

p.154 Define new main algorithm

In **Table 6-9** in section 6.10.1, a new value shall be added to the value column for row `bits [7:0]`:

Bits	Function	Value
Bits [7:0]	Identifiy the main underlying algorithm itself	... 0xC0: HKDF

The function column for `bits[15:12]` shall also be modified to read:

Bits	Function	Value
Bits [15:12]	Define the message digest for asymmetric signature algorithms or HKDF	

p.155 Add new object type for HKDF input keying material

The following entry shall be added to **Table 6-10**:

Name	Identifier	Possible sizes
TEE_TYPE_HKDF_IKM	0xA10000C0	8 to 4096 bits (multiple of 8)

p.156 Add new operation attributes for HKDF salt and info

The following entries shall be added to **Table 6-11**:

Name	Value	Protection	Type	Comment
TEE_ATTR_HKDF_IKM	0xC0000100	Protected	Ref	
TEE_ATTR_HKDF_SALT	0xD0000200	Public	Ref	
TEE_ATTR_HKDF_INFO	0xD0000300	Public	Ref	
TEE_ATTR_HKDF_OKM_LENGTH	0xF0000400	Public	Value	

PBKDF2

This document describes the OP-TEE implementation of the key derivation function, *PBKDF2* as specified in RFC 2898 section 5.2. This RFC is a republication of PKCS #5 v2.0 from RSA Laboratories' Public-Key Cryptography Standards (PKCS) series. You may disable this extension by setting the following configuration variable in `conf.mk`:

```
CFG_CRYPTO_PBKDF2 := n
```

API extension

To support PBKDF2, the *TEE Internal Core API* v1.1 was extended with a new algorithm descriptor, new object types, and new object attributes as described below.

p.95 Add new object type to TEE_PopulateTransientObject

The following entry shall be added to **Table 5-8**:

Object type	Parts
TEE_TYPE_PBKDF2_PASSWORD	The TEE_ATTR_PBKDF2_PASSWORD part must be provided.

p.121 Add new algorithms for TEE_AllocateOperation

The following entry shall be added to **Table 6-3**:

Algorithm	Possible Modes
TEE_ALG_PBKDF2_HMAC_SHA1_DERIVE_KEY	TEE_MODE_DERIVE

p.126 Explain usage of PBKDF2 algorithm in TEE_SetOperationKey

In the bullet list about operation mode, the following shall be added:

- For the PBKDF2 algorithm, the only supported mode is TEE_MODE_DERIVE.

p.150 Define TEE_DeriveKey input attributes for new algorithms

The following sentence shall be deleted:

```
The TEE_DeriveKey function can only be used with the algorithm
TEE_ALG_DH_DERIVE_SHARED_SECRET
```

The following entry shall be added to **Table 6-7**:

Algorithm	Possible operation parameters
TEE_ALG_PBKDF2_HMAC_SHA1_DERIVE_KEY	TEE_ATTR_SECRET_VALUE: The password TEE_ATTR_PBKDF2_DKM_LENGTH: up to 512 bytes. This parameter is mandatory. TEE_ATTR_PBKDF2_SALT TEE_ATTR_PBKDF2_ITERATION_COUNT: This parameter is mandatory.

p.152 Add new algorithm identifiers

The following entries shall be added to **Table 6-8**:

Algorithm	Identifier
TEE_ALG_PBKDF2_HMAC_SHA1_DERIVE_KEY	0x800020C2

p.154 Define new main algorithm

In **Table 6-9** in section 6.10.1, a new value shall be added to the value column for row `bits [7:0]`:

Bits	Function	Value
Bits [7:0]	Identifiy the main underlying algorithm itself	... 0xC2: PBKDF2

The function column for `bits[15:12]` shall also be modified to read:

Bits	Function	Value
Bits [15:12]	Define the message digest for asymmetric signature algorithms or PBKDF2	

p.155 Add new object type for PBKDF2 password

The following entry shall be added to **Table 6-10**:

Name	Identifier	Possible sizes
TEE_TYPE_PBKDF2_PASSWORD	0xA10000C2	8 to 4096 bits (multiple of 8)

p.156 Add new operation attributes for Concat KDF

The following entries shall be added to **Table 6-11**:

Name	Value	Protection	Type	Comment
TEE_ATTR_PBKDF2_PASSWORD	0xC0001000	Protected	Ref	
TEE_ATTR_PBKDF2_SALT	0xD0002000	Public	Ref	
TEE_ATTR_PBKDF2_ITERATION_COUNT	0xF0003000	Public	Value	
TEE_ATTR_PBKDF2_DKM_LENGTH	0xF0004000	Public	Value	The length (in bytes) of the derived keying material to be generated, maximum 512.

2.6 Libraries

2.6.1 libutee

The *TEE Internal Core API* describes services that are provided to Trusted Applications. **libutee** is a library that implements this API.

libutee is a static library the Trusted Applications shall statically link against. Trusted Applications do execute in non-privileged secure userspace and libutee also aims at being executed in the non-privileged secure userspace.

Some services for this API are fully statically implemented inside the libutee library while some services for the API are implemented inside the OP-TEE core (privileged level) and libutee calls such services through system calls.

2.6.2 libmpa

Now deprecated, used to provide the BigNum library in OP-TEE.

2.7 Porting guidelines

This document serves a dual purpose:

- Serve as a base for getting OP-TEE up and running on a new device with initial xtest validation passing. This is the first part of this document (section 2).
- Highlight the missing pieces if you intend to make a real secure product, that is what the second part of this document is about.

We are trying our best to implement full end to end security in OP-TEE in a generic way, but due to the nature of devices being different, NDA etc, it is not always possible for us to do so and in those cases, we most often try to write a generic API, but we will just stub the code. This porting guideline highlights the missing pieces that must be addressed in a real secure consumer device. Hopefully we will sooner or later get access to devices where we at least can make reference implementations publicly available to everyone for the missing pieces we are talking about here.

2.7.1 Add a new platform

The first thing you need to do after you have decided to port OP-TEE to another device is to add a new platform device. That can either be adding a new platform variant (`PLATFORM_FLAVOR`) if it is a device from a family already supported, or it can be a brand new platform family (`PLATFORM`). Typically this initial setup involve configuring UART, memory addresses etc. For simplicity let us call our fictive platform for "gendev" just so we have something to refer to when writing examples further down.

core/arch/arm

In `core/arch/arm` you will find all the currently supported devices. That is where you are supposed to add a new platform or modify an existing one. Typically you will find this set of files in a specific platform folder:

```
$ ls
conf.mk  main.c  platform_config.h  sub.mk
```

So for the gendev platform it means that the files should be placed in this folder:

```
core/arch/arm/plat-gendev
```

conf.mk

This is the device specific makefile where you define configurations unique to your platform. This mainly comprises two things: - OP-TEE configuration variables (`CFG_`), which may be assigned values in two ways. `CFG_FOO ?= bar` should be used to provide a default value that may be modified at compile time. On the other hand, variables that must be set to some value and cannot be modified should be set by: `$(call force,CFG_FOO,bar)`. - Compiler flags for the TEE core, the user mode libraries and the Trusted Applications, which may be added to macros used by the build system. Please see Platform-specific configuration and flags in the build system documentation.

It is recommended to use a existing platform configuration file as a starting point. For instance, core/arch/arm/plat-hikey/conf.mk.

The platform `conf.mk` file should at least define the default platform flavor for the platform, the core configurations (architecture and number of cores), the main configuration directives (generic boot, arm trusted firmware support, generic time source, console driver, etc...) and some platform default configuration settings.

```
PLATFORM_FLAVOR ?= hikey

include core/arch/arm/cpu/cortex-armv8-0.mk

$(call force,CFG_TEE_CORE_NB_CORE,8)
$(call force,CFG_GENERIC_BOOT,y)
$(call force,CFG_PL011,y)
$(call force,CFG_PM_STUBS,y)
$(call force,CFG_SECURE_TIME_SOURCE_CNTPCT,y)
$(call force,CFG_WITH_ARM_TRUSTED_FW,y)
$(call force,CFG_WITH_LPAE,y)

ta-targets = ta_arm32
ta-targets += ta_arm64

CFG_NUM_THREADS ?= 8
CFG_CRYPTO_WITH_CE ?= y
CFG_WITH_STACK_CANARIES ?= y
CFG_CONSOLE_UART ?= 3
CFG_DRAM_SIZE_GB ?= 2
```

main.c

This platform specific file will contain power management handlers and code related to the UART. We will talk more about the information related to the handlers further down in this document. For our gendev device it could look like this (here we are excluding the necessary license header to save some space):

```
#include <console.h>
#include <drivers/serial8250_uart.h>
#include <kernel/generic_boot.h>
```

(continues on next page)

(continued from previous page)

```
#include <kernel/panic.h>
#include <kernel/pm_stubs.h>
#include <mm/core_mmu.h>
#include <platform_config.h>
#include <stdint.h>
#include <tee/entry_fast.h>
#include <tee/entry_std.h>

static void main_fiq(void)
{
    panic();
}

static const struct thread_handlers handlers = {
    .std_smc = tee_entry_std,
    .fast_smc = tee_entry_fast,
    .nintr = main_fiq,
    .cpu_on = cpu_on_handler,
    .cpu_off = pm_do_nothing,
    .cpu_suspend = pm_do_nothing,
    .cpu_resume = pm_do_nothing,
    .system_off = pm_do_nothing,
    .system_reset = pm_do_nothing,
};

const struct thread_handlers *generic_boot_get_handlers(void)
{
    return &handlers;
}

/*
 * Register the physical memory area for peripherals etc. Here we are
 * registering the UART console.
 */
register_phys_mem(MEM_AREA_IO_NSEC, CONSOLE_UART_BASE, SERIAL8250_UART_REG_SIZE);

static struct serial8250_uart_data console_data;

void console_init(void)
{
    serial8250_uart_init(&console_data, CONSOLE_UART_BASE,
                         CONSOLE_UART_CLK_IN_HZ, CONSOLE_BAUDRATE);
    register_serial_console(&console_data.chip);
}
```

platform_config.h

This is a mandatory header file for every platform, since there are several files relaying upon the existence of this particular file. This file is where you will find the major differences between different platforms, since this is where you do the memory configuration, define base addresses etc. we are going to list a few here, but it probably makes more sense to have a look at the already existing `platform_config.h` files for the other platforms. Our fictive gendev could look like this:

```
#ifndef PLATFORM_CONFIG_H
#define PLATFORM_CONFIG_H

/* Make stacks aligned to data cache line length */
```

(continues on next page)

(continued from previous page)

```
#define STACK_ALIGNMENT             64

/* 8250 UART */
#define CONSOLE_UART_BASE   0xcafebabe /* UART0 */
#define CONSOLE_BAUDRATE    115200
#define CONSOLE_UART_CLK_IN_HZ      19200000

/* Optional: when used with CFG_WITH_PAGER, defines the device SRAM */
#define TZSRAM_BASE         0x3F000000
#define TZSRAM_SIZE         (200 * 1024)

/* Mandatory main secure RAM usually DDR */
#define TZDRAM_BASE         0x60000000
#define TZDRAM_SIZE         (32 * 1024 * 1024)

/* Mandatory TEE RAM location and core load address */
#define TEE_RAM_START               TZDRAM_BASE
#define TEE_RAM_PH_SIZE             TEE_RAM_VA_SIZE
#define TEE_RAM_VA_SIZE             (4 * 1024 * 1024)
#define TEE_LOAD_ADDR               (TZDRAM_BASE + 0x20000)

/* Mandatory TA RAM (external less secure RAM) */
#define TA_RAM_START                (TZDRAM_BASE + TEE_RAM_VA_SIZE)
#define TA_RAM_SIZE         (TZDRAM_SIZE - TEE_RAM_VA_SIZE)

/* Mandatory: for static SHM, need a hardcoded physical address */
#define TEE_SHMEM_START             0x08000000
#define TEE_SHMEM_SIZE              (4 * 1024 * 1024)

#endif /* PLATFORM_CONFIG_H */
```

This is minimal amount of information in the `platform_config.h` file. I.e, the memory layout for on-chip and external RAM. Note that parts of the DDR typically will need to be shared with normal world, so there is need for some kind of memory firewall for this (more about that further down). As you can see we have also added the UART configuration here, i.e., the `DEVICE0_xyz` part.

Official board support in OP-TEE?

We do encourage everyone to submit their board support to the OP-TEE project itself, so it becomes part of the official releases and will be maintained by the OP-TEE community itself. If you intend to do so, then there are a few more things that you are supposed to do.

Update platforms supported

There is a section at the *Platforms supported* page that lists all devices officially supported in OP-TEE, that is where you also shall list your device. It should contain the name of the platform, then composite `PLATFORM` flag and whether the device is publicly available or not. If there is a product page on the internet for the device, please also create a link when writing the device name.

Update .shippable.yml

Since we are using Shippable to test pull requests etc, we would like that you also add your device to the .shippable.yml file, so that it will at least be built when someone is doing a pull request. Add a line at the end of file:

```
- _make PLATFORM=<platform-name>_
```

Maintainer

If you are submitting the board support upstream we are going to ask you to become the maintainer for the device you have added. This means that you should also update the MAINTAINERS.md file accordingly. By being a maintainer for a device you are responsible to keep it up to date and you will be asked every quarter as part of the OP-TEE release schedule to test your device running the latest OP-TEE software.

Update build.git and manifest.git

This isn't strictly necessary, but we are trying to create and maintain OP-TEE developer builds that should make it easy to setup, build and deploy OP-TEE on various devices. We encourage all maintainers to do the same for the boards they are in charge of. Therefore please consider creating a new *manifest* (and a new `*.mk` in *build*) for the device you have added to OP-TEE.

2.7.2 Hardware Unique Key

Most devices have some kind of Hardware Unique Key (HUK) that is mainly used to derive other keys. The HUK could for example be used when deriving keys used in secure storage etc. The important thing with the HUK is that it needs to be well protected and in the best case the HUK should never ever be readable directly from software, not even from the secure side. There are different solutions to this, crypto accelerator might have support for it or, it could involve another secure co-processor.

In OP-TEE the HUK **is** just **stubbed** and you will see that in the function called `tee_otp_get_hw_unique_key(...)` in core/include/kernel/tee_common_otp.h. In a real secure product you **must** replace this with something else. If your device lacks the hardware support for a HUK, then you must at least change this to something else than just zeroes. But, remember it is not good secure practice to store a key in software, especially not the key that is the root for everything else, so this is not something we recommend that you should do.

2.7.3 Secure Clock

The Time API in GlobalPlatform Internal Core API specification defines three sources of time; system time, TA persistent time and REE time. The REE time is by nature considered as an unsecure source of time, but the other two should in a fully trustable hardware make use of trustable source of time, i.e., a secure clock. Note that from GlobalPlatform point of view it is not required to make use of a secure clock, i.e., it is OK to use time from REE, but the level of trust should be reflected by the `gpd.tee.systemTime.protectionLevel` property and the `gpd.tee.TAPersistentTime.protectionLevel` property (100=REE controlled clock, 1000=TEE controlled clock). So the functions that one needs to pay attention to are `tee_time_get_sys_time(...)` and `tee_time_get_ta_time(...)`. If your hardware has a secure clock, then you probably want to change the implementation there to instead use the secure clock (and then you would also need to update the property accordingly, i.e., `tee_time_get_sys_time_protection_level()` and the variable `ta_time_prot_lvl` in `tee_svc.c`).

2.7.4 Root and Chain of Trust

To be able to assure that your devices are running the (untampered) binaries you intended to run you will need to establish some kind of trust anchor on the devices.

The most common way of doing that is to put the root public key in some read only memory on the device. Quite often SoC's/OEM's stores public key(s) directly or the hash(es) of the public key(s) in OTP. When the boot ROM (which indeed needs to be ROM) is about to load the first stage bootloader it typically reads the public key from the software binary itself, hash the key and compare it to the key in OTP. If they are matching, then the boot ROM can be sure that the first stage bootloader was indeed signed with the corresponding private key.

In OP-TEE you will not find any code at all related to this and this is a good example when it is hard for us to do this in a generic way since device manufacturers all tend to do this in their own unique way and they are not very keen on

sharing their low level boot details and security implementation with the rest of the world. This is especially true on ARMv7-A. For ARMv8-A it looks bit better, since Arm in Trusted Firmware A have implemented and defined how a abstract the chain of trust (see auth-framework.rst). We have successfully verified OP-TEE by using the authentication framework from Trusted Firmware A (see *Secure boot* for the details).

2.7.5 Hardware Crypto IP

By default OP-TEE uses a software crypto library (currently mbed TLS and LibTomCrypt) and you have the ability to enable Crypto Extensions that were introduced with ARMv8-A (if the device is capable of that). Some of the devices we have in our hands do have hardware crypto IP's, but due to NDA's etc it has not been possible to enable it. If you have a device capable of doing crypto operations on a dedicated crypto block and you prefer to use that in favor for the software implementation, then you will need to implement relevant functions defined in core/include/crypto/crypto.h, the Crypto API, and write the low level driver that communicates with the device. Our *Cryptographic implementation* page describes in detail how the Crypto API is integrated. Since the communication with crypto blocks tends to be quite different depending on what kind of crypto IP you have, we have not written how that should be done. It might be that we do that in the future when get hold of a device where we can use the crypto block.

By default OP-TEE is configured with a software PRNG. The entropy is added to software PRNG at various places, but unfortunately it is still quite easy to predict the data added as entropy. As a consequence, unless the RNG is based on hardware the generated random will be quite weak.

2.7.6 Power Management / PSCI

In the *Add a new platform* section where we talked about the file `main.c`, we added a couple of handlers related to power management, we are talking about the following lines:

```
.cpu_on = cpu_on_handler,
.cpu_off = pm_do_nothing,
.cpu_suspend = pm_do_nothing,
.cpu_resume = pm_do_nothing,
.system_off = pm_do_nothing,
.system_reset = pm_do_nothing,
```

The only function that actually does something there is the `cpu_on` function, the rest of them are stubbed. The main reason for that is because we think that how to suspend and resume is a device dependent thing. The code in OP-TEE is prepared so that callbacks etc from Trusted Firmware A will be routed to OP-TEE, but since the function(s) are just stubbed we will not do anything and just return. In a real production device, you would probably want to save and restore CPU states, secure hardware IPs' registers and TZASC and other memory firewall related setting when these callbacks are being called.

2.7.7 Memory firewalls / TZASC

Arm have defined a system IP / SoC peripheral called TrustZone Address Space Controller (TZASC, see TZC-380 and TZC-400). TZASC can be used to configure DDR memory into separate regions in the physcial address space, where each region can have an individual security level setting. After enabling TZASC, it will perform security checks on transactions to memory or peripherals. It is not always the case that TZASC is on a device, in some cases the SoC has developed something equivalent. In OP-TEE this is very well reflected, i.e., different platforms have different ways of protecting their memory. On ARMv8-A platforms we are in most of the cases using Trusted Firmware A as the boot firmware and there the secure bootloader is the one that configures secure vs non-secure memory using TZASC (see plat_arm_security_setup in TF-A). The takeaway here is that you must make sure that you have configured whatever memory firewall your device has such that it has a secure and a non-secure memory area.

2.7.8 Trusted Application private/public keypair

By default all Trusted Applications (TA's) are signed with the pre-generated 2048-bit RSA development key (private key). This key is located in the `keys` folder (in the root of optee_os.git) and is named `default_ta.pem`. This key **must** be replaced with your own key and you should **never ever** check-in this private key in the source code tree when in use in a real product. The recommended way to store private keys is to use some kind of HSM (Hardware Security Module), but an alternative would be temporary put the private key on a computer considered as secure when you are about to sign TA's intended to be used in real products. Typically it is only a few number of people having access to this type of key in company. The key handling in OP-TEE is currently a bit limited since we only support a single key which is used for all TA's. We have plans on extending this to make it a bit more flexible. Exactly when that will happen has not been decided yet.

2.8 Secure boot

2.8.1 Armv8-A - Using the authentication framework in TF-A

This section gives a brief description on how to enable the verification of OP-TEE using the authentication framework in Trusted Firmware A (TF-A), i.e., something that could be used in an Armv8-A environment.

According to user-guide.rst, there is no additional specific build options for the verification of OP-TEE. If we have enabled the authentication framework and specified the `BL32` build option when building TF-A, the BL32 related certificates will be created automatically by the cert_create tool, and then these certificates will be verified during booting up.

To enable the authentication framework, the following steps should be followed according to user-guide.rst. For more details about the authentication framework, please see auth-framework.rst and trusted-board-boot.rst.

- Check out a recent version of the mbed TLS repository and then switch to tag mbedtls-2.2.0
- Besides the normal build options, add the following build options for TF-A

```
MBEDTLS_DIR=<path of the directory containing mbed TLS sources>
TRUSTED_BOARD_BOOT=1
GENERATE_COT=1
ARM_ROTPK_LOCATION=devel_rsa
ROT_KEY=<TF-A-PATH/plat/arm/board/common/rotpk/arm_rotprivk_rsa.pem>
```

Above steps have been tested on FVP platform, all verification steps are OK and xtest runs successfully without regression.

2.8.2 Armv7-A systems

Unlike for Armv8-A systems where one can use a more standardized way of doing secure boot by leverage the authentication framework as described above, most device manufacturers have their own way of doing secure boot. Please reach out directly to the manufacturer for the device you are working with to be able to understand how to do secure boot on their devices.

2.9 Secure storage

2.9.1 Background

Secure Storage in OP-TEE is implemented according to what has been defined in GlobalPlatform's *TEE Internal Core API* (here called Trusted Storage). This specification mandates that it should be possible to store general-purpose data and key material that guarantees confidentiality and integrity of the data stored and the atomicity of the operations that modifies the storage (atomicity here means that either the entire operation completes successfully or no write is done).

There are currently two secure storage implementations in OP-TEE:

- The first one relies on the normal world (REE) file system. It is described in this document and is the default implementation. It is enabled at compile time by `CFG_REE_FS=y`.
- The second one makes use of the Replay Protected Memory Block (RPMB) partition of an eMMC device, and is enabled by setting `CFG_RPMB_FS=y`. It is described in *RPMB Secure Storage*.

It is possible to use the normal world file systems and the RPMB implementations simultaneously. For this, two OP-TEE specific storage identifiers have been defined: `TEE_STORAGE_PRIVATE_REE` and `TEE_STORAGE_PRIVATE_RPMB`. Depending on the compile-time configuration, one or several values may be used. The value `TEE_STORAGE_PRIVATE` selects the REE FS when available, otherwise the RPMB FS (in this order).

2.9.2 REE FS Secure Storage

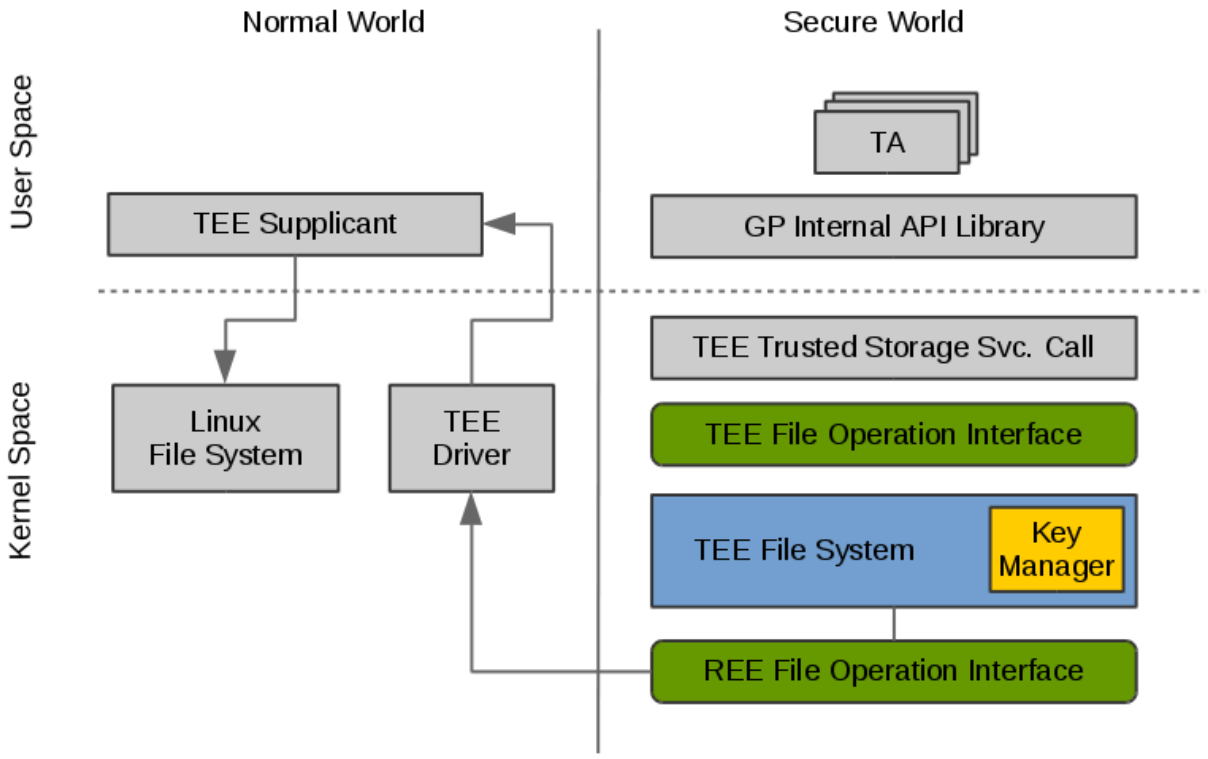


Fig. 6: Secure Storage System Architecture

Source Files in OP-TEE OS

Table 12: Secure storage files

Source file	Purpose
core/tee/tee_svc_storage.c	TEE trusted storage service calls
core/tee/tee_ree_fs.c	TEE file system & REE file operation interface
core/tee/fs_htree.c	Hash tree
core/tee/tee_fs_key_manager.c	Key manager
lib/libutee/	GlobalPlatform Internal API library

Basic File Operation Flow

When a TA is calling the write function provided by GP Trusted Storage API to write data to a persistent object, a corresponding syscall implemented in TEE Trusted Storage Service will be called, which in turn will invoke a series of TEE file operations to store the data. TEE file system will then encrypt the data and send REE file operation commands and the encrypted data to TEE supplicant by a series of RPC messages. TEE supplicant will receive the messages and store the encrypted data accordingly to the Linux file system. Reading files are handled in a similar manner.

GlobalPlatform Trusted Storage Requirement

Below is an excerpt from the specification, listing the most vital requirements:

```
1. The Trusted Storage may be backed by non-secure resources as long as
   suitable cryptographic protection is applied, which MUST be as strong as
   the means used to protect the TEE code and data itself.

2. The Trusted Storage MUST be bound to a particular device, which means
   that it MUST be accessible or modifiable only by authorized TAs
   running in the same TEE and on the same device as when the data was
   created.

3. Ability to hide sensitive key material from the TA itself.

4. Each TA has access to its own storage space that is shared among all the
   instances of that TA but separated from the other TAs.

5. The Trusted Storage must provide a minimum level of protection against
   rollback attacks. It is accepted that the actually physical storage
   may be in an insecure area and so is vulnerable to actions from
   outside of the TEE. Typically, an implementation may rely on the REE
   for that purpose (protection level 100) or on hardware assets
   controlled by the TEE (protection level 1000).

(see GP TEE Internal Core API section 2.5 and 5.2)
```

If configured with `CFG_RPMB_FS=y` the protection against rollback is controlled by the TEE and is set to 1000. If `CFG_RPMB_FS=n`, there's no protection against rollback, and the protection level is set to 0.

TEE File Structure in Linux File System

OP-TEE by default uses `/data/tee/` as the secure storage space in the Linux file system. Each persistent object is assigned an internal identifier. It is an integer which is visible in the Linux file system as `/data/tee/<file number>`.

A directory file, `/data/tee/dirf.db`, lists all the objects that are in the secure storage. All normal world files are integrity protected and encrypted, as described below.

2.9.3 Key Manager

Key manager is a component in TEE file system, and is responsible for handling data encryption and decryption and also management of the sensitive key materials. There are three types of keys used by the key manager: the Secure Storage Key (*SSK*), the TA Storage Key (*TSK*) and the File Encryption Key (*FEK*).

Secure Storage Key (SSK)

SSK is a per-device key and is generated and stored in secure memory when OP-TEE is booting. SSK is used to derive the TA Storage Key (TSK).

SSK is derived by

$$\text{SSK} = \text{HMAC}_{\text{SHA256}} (\text{HUK}, \text{Chip ID} \parallel \text{“static string”})$$

The functions to get *Hardware Unique Key* (HUK) and chip ID depends on the platform implementation. Currently, in OP-TEE OS we only have a per-device key, SSK, which is used for secure storage subsystem, but, for the future we might need to create different per-device keys for different subsystems using the same algorithm as we generate the SSK; An easy way to generate different per-device keys for different subsystems is using different static strings to generate the keys.

Trusted Application Storage Key (TSK)

The TSK is a per-Trusted Application key, which is generated from the SSK and the TA's identifier (UUID). It is used to protect the FEK, in other words, to encrypt/decrypt the FEK.

TSK is derived by:

$$\text{TSK} = \text{HMAC}_{\text{SHA256}} (\text{SSK}, \text{TA_UUID})$$

File Encryption Key (FEK)

When a new TEE file is created, key manager will generate a new FEK by PRNG (pesudo random number generator) for the TEE file and store the encrypted FEK in meta file. FEK is used for encrypting/decrypting the TEE file information stored in meta file or the data stored in block file.

2.9.4 Hash Tree

The hash tree is responsible for handling data encryption and decryption of a secure storage file. The hash tree is implemented as a binary tree where each node (`struct tee_fs_htree_node_image` below) in the tree protects its two child nodes and a data block. The meta data is stored in a header (`struct tee_fs_htree_image` below) which also protects the top node.

All fields (header, nodes, and blocks) are duplicated with two versions, 0 and 1, to ensure atomic updates. See core/tee/fs_htree.c for details.

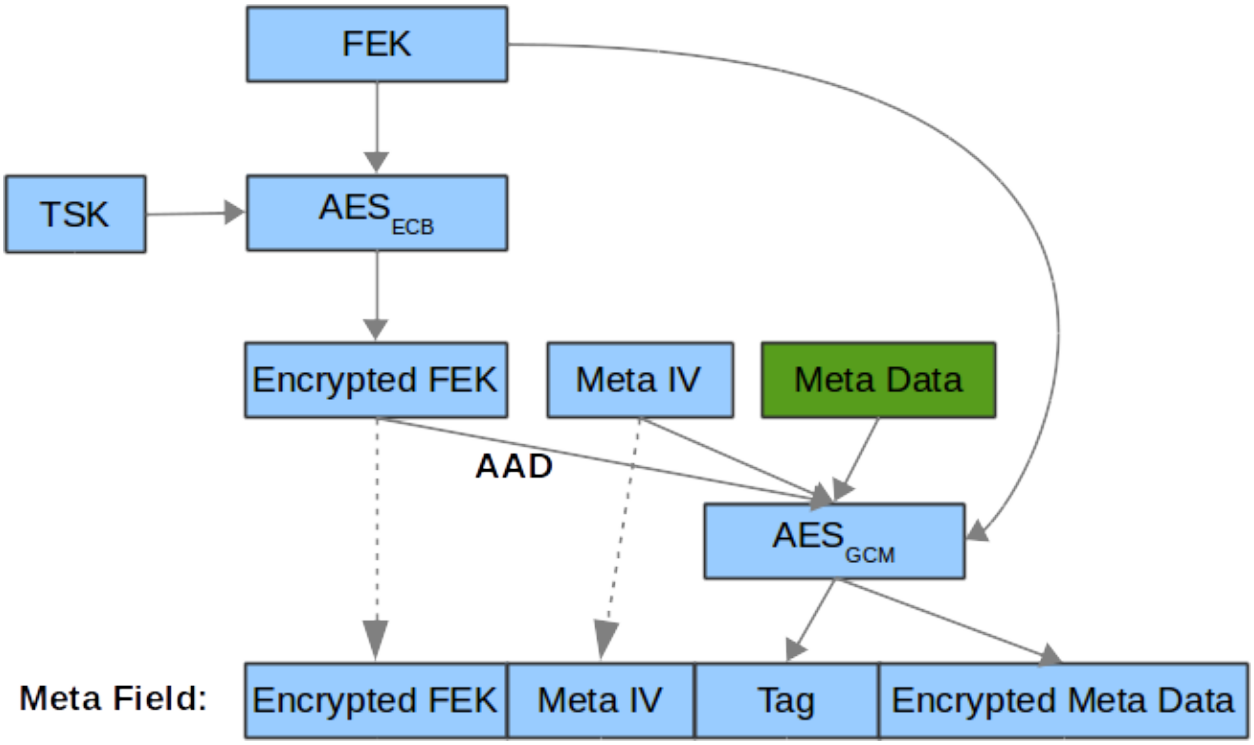


Fig. 7: Meta data encryption

Meta Data Encryption Flow

A new meta IV will be generated by PRNG when a meta data needs to be updated. The size of meta IV is defined in core/include/tee/fs_htree.h, likewise are the data structures of meta data and node data are defined in fs_htree.h as follows:

```
struct tee_fs_htree_node_image {
        uint8_t hash[TEE_FS_HTREE_HASH_SIZE];
        uint8_t iv[TEE_FS_HTREE_IV_SIZE];
        uint8_t tag[TEE_FS_HTREE_TAG_SIZE];
        uint16_t flags;
};

struct tee_fs_htree_meta {
        uint64_t length;
};

struct tee_fs_htree_imeta {
        struct tee_fs_htree_meta meta;
        uint32_t max_node_id;
};

struct tee_fs_htree_image {
        uint8_t iv[TEE_FS_HTREE_IV_SIZE];
        uint8_t tag[TEE_FS_HTREE_TAG_SIZE];
        uint8_t enc_fek[TEE_FS_HTREE_FEK_SIZE];
        uint8_t imeta[sizeof(struct tee_fs_htree_imeta)];
        uint32_t counter;
};
```

Block Data Encryption Flow

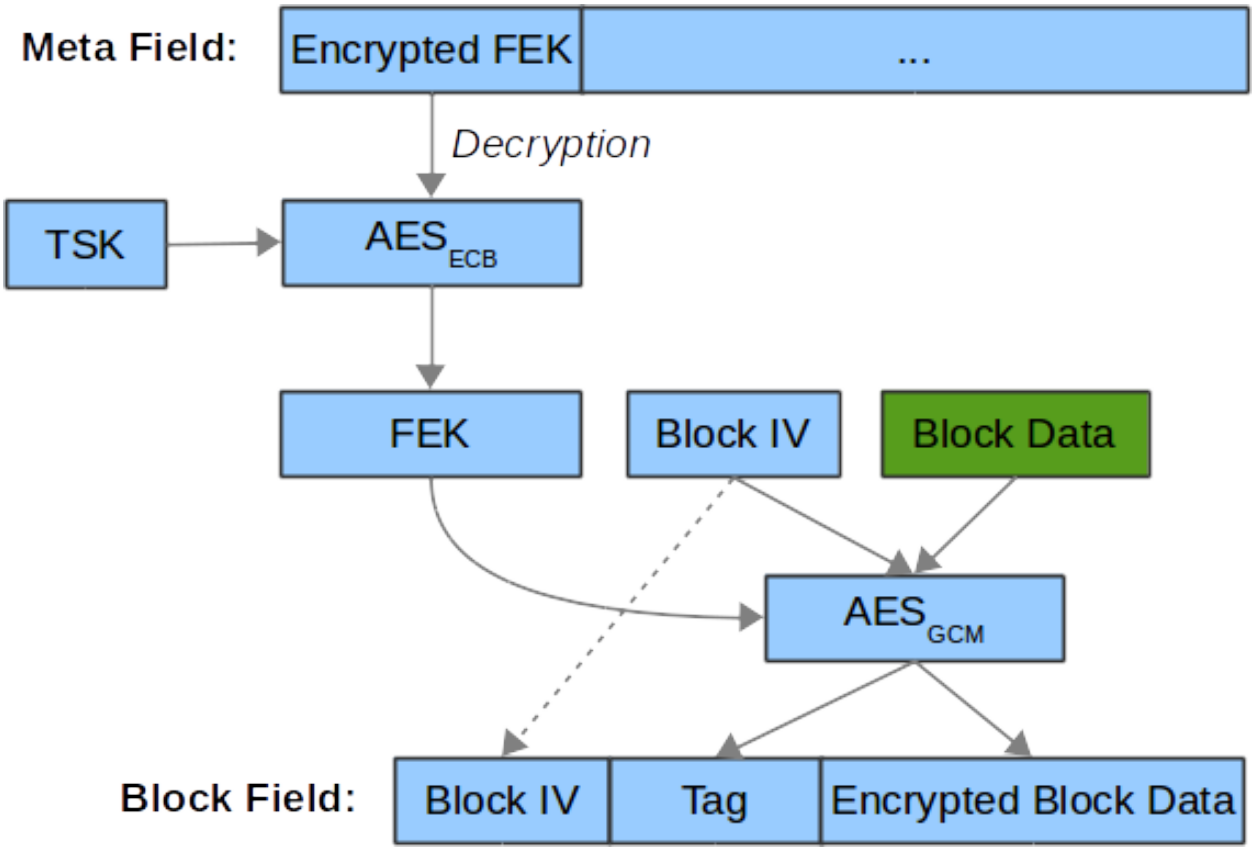


Fig. 8: Block data encryption

A new block IV will be generated by PRNG when a block data needs to be updated. The size of block IV is defined in core/include/tee/fs_htree.h.

2.9.5 Atomic Operation

According to GlobalPlatform Trusted Storage requirement of the atomicity, the following operations should support atomic update:

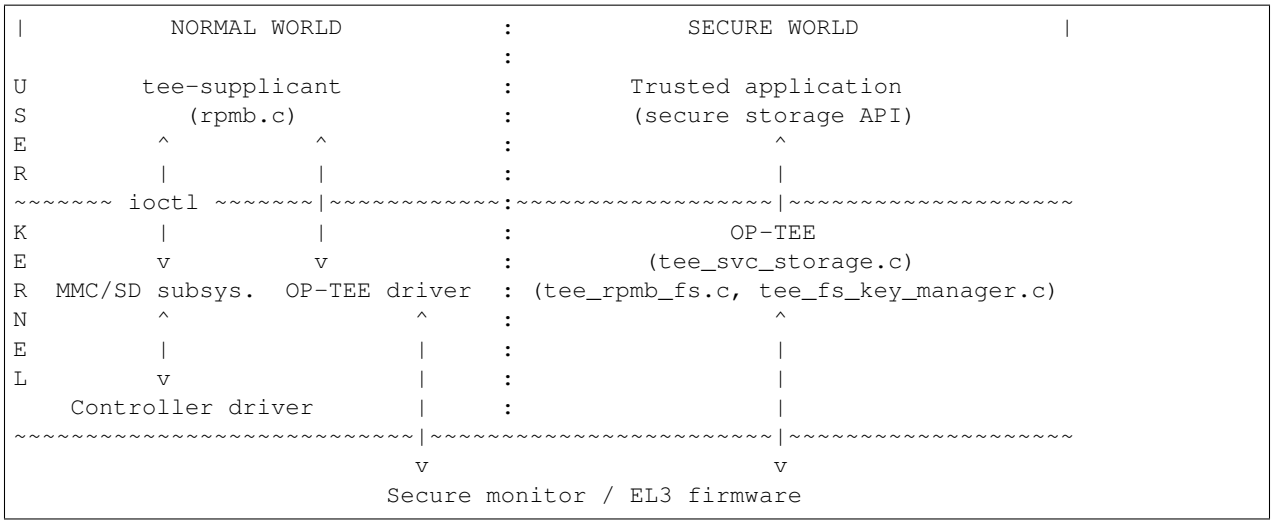
```
Write, Truncate, Rename, Create and Delete
```

The strategy used in OP-TEE secure storage to guarantee the atomicity is out-of-place update.

2.9.6 RPMB Secure Storage

This document describes the RPMB secure storage implementation in OP-TEE, which is enabled by setting `CFG_RPMB_FS=y`. Trusted Applications may use this implementation by passing a storage ID equal to `TEE_STORAGE_PRIVATE_RPMB`, or `TEE_STORAGE_PRIVATE` if `CFG_REE_FS` is disabled. For details about RPMB, please refer to the JEDEC eMMC specification (JESD84-B51).

The architecture is depicted below.



For information about the `ioctl()` interface to the MMC/SD subsystem in the Linux kernel, see the Linux core MMC header file linux/mmc/core.h and the mmc-utils repository.

The Secure Storage API

This part is common with the REE-based filesystem. The interface between the system calls in core/tee/tee_svc_storage.c and the RPMB filesystem is the *tee_file_operations*, namely `struct tee_file_ops`.

The RPMB filesystem

The FS implementation is entirely in core/tee/tee_rpmb_fs.c and the RPMB partition is divided in three parts:

- The first 128 bytes are reserved for partition data (`struct rpmb_fs_partition`).
- At offset 512 is the File Allocation Table (FAT). It is an array of `struct rpmb_fat_entry` elements, one per file. The FAT grows dynamically as files are added to the filesystem. Among other things, each entry has the start address for the file data, its size, and the filename.
- Starting from the end of the RPMB partition and extending downwards is the file data area.

Space in the partition is allocated by the general-purpose allocator functions, `tee_mm_alloc(...)` and `tee_mm_alloc2(...)`.

All file operations are atomic. This is achieved thanks to the following properties:

- Writing one single block of data to the RPMB partition is guaranteed to be atomic by the eMMC specification.
- The FAT block for the modified file is always updated last, after data have been written successfully.
- Updates to file content is done in-place only if the data do not span more than the "reliable write block count" blocks. Otherwise, or if the file needs to be extended, a new file is created.

Device access

There is no eMMC controller driver in OP-TEE. The device operations all have to go through the normal world. They are handled by the `tee-supplicant` process which further relies on the kernel's `ioctl()` interface to access the device. `tee-supplicant` also has an emulation mode which implements a virtual RPMB device for test purposes.

RPMB operations are the following:

- Reading device information (partition size, reliable write block count).
- Programming the security key. This key is used for authentication purposes. Note that it is different from the Secure Storage Key (SSK) defined below, which is used for encryption. Like the SSK however, the security key is also derived from a hardware unique key or identifier. Currently, the function `tee_otp_get_hw_unique_key()` is used to generate the RPMB security key.
- Reading the write counter value. The write counter is used in the HMAC computation during read and write requests. The value is read at initialization time, and stored in `struct tee_rpmb_ctx`, i.e., `rpmb_ctx->wr_cnt`.
- Reading or writing blocks of data.

RPMB operations are initiated on request from the FS layer. Memory buffers for requests and responses are allocated in shared memory using `thread_rpc_alloc_payload(...)`. Buffers are passed to the normal world in a `TEE_RPC_RPMB_CMD` message, thanks to the `thread_rpc_cmd()` function. Most RPMB requests and responses use the data frame format defined by the JEDEC eMMC specification. HMAC authentication is implemented here also.

Encryption

The FS encryption routines are in core/tee/tee_fs_key_manager.c. Block encryption protects file data. The algorithm is 128-bit AES in Cipher Block Chaining (CBC) mode with Encrypted Salt-Sector Initialization Vector (ESSIV), see CBC-ESSIV for details.

- During OP-TEE initialization, a 128-bit AES Secure Storage Key (SSK) is derived from a *Hardware Unique Key* (HUK). It is kept in secure memory and never written to disk. A Trusted Application Storage Key is derived from the SSK and the TA UUID.
- For each file, a 128-bit encrypted File Encryption Key (FEK) is randomly generated when the file is created, encrypted with the TSK and stored in the FAT entry for the file.
- Each 256-byte block of data is then encrypted in CBC mode. The initialization vector is obtained by the ESSIV algorithm, that is, by encrypting the block number with a hash of the FEK. This allows direct access to any block in the file, as follows:

```
FEK = AES-Decrypt(TSK, encrypted FEK);
k = SHA256(FEK);
IV = AES-Encrypt(128 bits of k, block index padded to 16 bytes)
Encrypted block = AES-CBC-Encrypt(FEK, IV, block data);
Decrypted block = AES-CBC-Decrypt(FEK, IV, encrypted block data);
```

SSK, TSK and FEK handling is common with the REE-based secure storage, while the AES CBC block encryption is used only for RPMB (the REE implementation uses GCM). The FAT is not encrypted.

REE FS hash state

If configured with both `CFG_REE_FS=y` and `CFG_RPMB_FS=y` the REE FS will create a special file, `dirfile.db.hash` in RPMB which hold a hash representing the state of REE FS.

2.9.7 Important caveats

Warning: Currently **no OP-TEE platform** is able to support retrieval of the Hardware Unique Key or Chip ID required for secure operation. For all platforms, a constant key is used, resulting in no protection against

decryption, or Secure Storage duplication to other devices. This is because information about how to retrieve key data from the SoC is considered sensitive by the vendors and it is not publicly available.

In OP-TEE, there are APIs for reading keys generically from One-Time-Programmable (OTP) memory. But there are no existing platform implementations.

To allow Secure Storage to operate securely on your platform, you must define implementations in your platform code for:

```
void tee_otp_get_hw_unique_key(struct tee_hw_unique_key *hwkey);

int tee_otp_get_die_id(uint8_t *buffer, size_t len);
```

These implementations should fetch the key data from your SoC-specific e-fuses, or crypto unit according to the method defined by your SoC vendor.

2.9.8 References

For more information about secure storage, please see SFO15-503, LAS16-504, SFO17-309 at *Presentations* and the *TEE Internal Core API* specification.

2.10 Trusted Applications

There are two ways to implement Trusted Applications (TAs), Pseudo TAs and user mode TAs. User mode TAs are full featured Trusted Applications as specified by the *GlobalPlatform API* TEE specifications, these are simply the ones people are referring to when they are saying "Trusted Applications" and in most cases this is the preferred type of TA to write and use.

2.10.1 Pseudo Trusted Applications

A Pseudo Trusted Application is not a Trusted Application. A Pseudo TA is not a specific entity. A Pseudo TA is an interface. It is an interface exposed by the OP-TEE Core to its outer world: to secure client Trusted Applications and to non-secure client entities.

These are implemented directly to the OP-TEE core tree in, e.g., `core/pta` and are built along with and statically built into the OP-TEE core blob.

The Pseudo Trusted Applications included in OP-TEE already are OP-TEE secure privileged level services hidden behind a "GlobalPlatform TA Client" API. These Pseudo TAs are used for various purposes such as specific secure services or embedded tests services.

Pseudo TAs **do not** benefit from the GlobalPlatform Core Internal API support specified by the GlobalPlatform TEE specs. These APIs are provided to TAs as a static library each TA shall link against (the "*libutee*") and that calls OP-TEE core service through system calls. As OP-TEE core does not link with *libutee*, Pseudo TAs can **only** use the OP-TEE core internal APIs and routines.

As Pseudo TAs runs at the same privileged execution level as the OP-TEE core code itself and that might or might not be desirable depending on the use case.

In most cases an unprivileged (user mode) TA is the best choice instead of adding your code directly to the OP-TEE core. However if you decide your application is best handled directly in OP-TEE core like this, you can look at `core/pta/stats.c` as a template and just add your Pseudo TA based on that to the `sub.mk` in the same directory.

2.10.2 User Mode Trusted Applications

User Mode Trusted Applications are loaded (mapped into memory) by OP-TEE core in the Secure World when something in Rich Execution Environment (REE) wants to talk to that particular application UUID. They run at a lower CPU privilege level than OP-TEE core code. In that respect, they are quite similar to regular applications running in the REE, except that they execute in Secure World.

Trusted Application benefit from the GlobalPlatform *TEE Internal Core API* as specified by the GlobalPlatform TEE specifications. There are several types of user mode TAs, which differ by the way they are stored.

2.10.3 TA locations

Plain TAs (user mode) can reside and be loaded from various places. There are three ways currently supported in OP-TEE.

Early TA

The so-called early TAs are virtually identical to the REE FS TAs, but instead of being loaded from the Normal World file system, they are linked into a special data section in the TEE core blob. Therefore, they are available even before `tee-supplicant` and the REE's filesystems have come up. Please find more details in the early TA commit.

REE filesystem TA

They consist of a ELF file, signed and optionally encrypted, named from the UUID of the TA and the suffix `.ta`. They are built separately from the OP-TEE core boot-time blob, although when they are built they use the same build system, and are signed with the key from the build of the original OP-TEE core blob.

Because the TAs are signed and optionally encrypted with `scripts/sign_encrypt.py`, they are able to be stored in the untrusted REE filesystem, and `tee-supplicant` will take care of passing them to be checked and loaded by the Secure World OP-TEE core.

REE-FS TA rollback protection

OP-TEE core maintains a `ta_ver.db` file in secure storage to check for version of REE TAs as they are loaded from REE-FS in order to prevent against any TA version downgrades. TA version can be configured via TA build option: `CFG_TA_VERSION=<unsigned integer>`.

Note: Here rollback protection is effective only when `CFG_RPMB_FS=y`.

REE-FS TA formats

REE filesystem TAs come in three formats:

1. Legacy TAs signed, not encrypted, cannot be created anymore by the build scripts since version 3.7.0.
2. Bootstrap TAs, signed with the key from the build of the original OP-TEE core blob, not encrypted.
3. Encrypted TAs, sign-then-encrypt-then-MAC, encrypted with `TA_ENC_KEY` when `CFG_ENCRYPT_TA=y`. During OP-TEE runtime, the symmetric key used to decrypt TA has to be provided in a platform specific manner via overriding API:

```
TEE_Result tee_otp_get_ta_enc_key(uint32_t key_type, uint8_t *buffer,
                                  size_t len);
```

REE-FS TA header structure

All REE filesystems TAs has common header, `struct shdr`, defined as:

```
enum shdr_img_type {
        SHDR_TA = 0,
        SHDR_BOOTSTRAP_TA = 1,
        SHDR_ENCRYPTED_TA = 2,
};

#define SHDR_MAGIC      0x4f545348

/**
 * struct shdr - signed header
 * @magic:      magic number must match SHDR_MAGIC
 * @img_type:   image type, values defined by enum shdr_img_type
 * @img_size:   image size in bytes
 * @algo:       algorithm, defined by public key algorithms TEE_ALG_*
 *              from TEE Internal API specification
 * @hash_size:  size of the signed hash
 * @sig_size:   size of the signature
 * @hash:       hash of an image
 * @sig:        signature of @hash
 */
struct shdr {
        uint32_t magic;
        uint32_t img_type;
        uint32_t img_size;
        uint32_t algo;
        uint16_t hash_size;
        uint16_t sig_size;
        /*
         * Commented out element used to visualize the layout dynamic part
         * of the struct.
         *
         * hash is accessed through the macro SHDR_GET_HASH and
         * signature is accessed through the macro SHDR_GET_SIG
         *
         * uint8_t hash[hash_size];
         * uint8_t sig[sig_size];
         */
};

#define SHDR_GET_SIZE(x)        (sizeof(struct shdr) + (x)->hash_size + \
                                 (x)->sig_size)
#define SHDR_GET_HASH(x)        (uint8_t *)(((struct shdr *)(x)) + 1)
#define SHDR_GET_SIG(x)         (SHDR_GET_HASH(x) + (x)->hash_size)
```

The field `img_type` tells the type of TA, if it's `SHDR_TA` (0), it's a legacy TA. If it's `SHDR_BOOTSTRAP_TA` (1) it's a bootstrap TA.

The field `algo` tells the algorithm used. The script used to sign TAs currently uses `TEE_ALG_RSASSA_PKCS1_V1_5_SHA256` (0x70004830). This means RSA with PKCS#1v1.5 padding

and SHA-256 hash function. OP-TEE accepts any of the `TEE_ALG_RSASSA_PKCS1_*` algorithms.

For bootstrap TAs `struct shdr` is followed by a subheader, `struct shdr_bootstrap_ta` which is defined as:

```
/**
 * struct shdr_bootstrap_ta - bootstrap TA subheader
 * @uuid:       UUID of the TA
 * @ta_version: Version of the TA
 */
struct shdr_bootstrap_ta {
        uint8_t uuid[sizeof(TEE_UUID)];
        uint32_t ta_version;
};
```

The fields `uuid` and `ta_version` allows extra checks to be performed when loading the TA. Currently only the `uuid` field is checked.

For encrypted TAs `struct shdr` is followed by a subheader, `struct shdr_bootstrap_ta` which is followed by another subheader, `struct shdr_encrypted_ta` defined as:

```
/**
 * struct shdr_encrypted_ta - encrypted TA header
 * @enc_algo:   authenticated encyption algorithm, defined by symmetric key
 *              algorithms TEE_ALG_* from TEE Internal API
 *              specification
 * @flags:      authenticated encyption flags
 * @iv_size:    size of the initialization vector
 * @tag_size:   size of the authentication tag
 * @iv:         initialization vector
 * @tag:        authentication tag
 */
struct shdr_encrypted_ta {
        uint32_t enc_algo;
        uint32_t flags;
        uint16_t iv_size;
        uint16_t tag_size;
        /*
         * Commented out element used to visualize the layout dynamic part
         * of the struct.
         *
         * iv is accessed through the macro SHDR_ENC_GET_IV and
         * tag is accessed through the macro SHDR_ENC_GET_TAG
         *
         * uint8_t iv[iv_size];
         * uint8_t tag[tag_size];
         */
};
```

The field `enc_algo` tells the algorithm used. The script used to encrypt TAs currently uses `TEE_ALG_AES_GCM` (0x40000810). OP-TEE core also accepts `TEE_ALG_AES_CCM` algorithm.

The field `flags` supports a single flag to tell encryption key type which is defined as:

```
#define SHDR_ENC_KEY_TYPE_MASK  0x1

enum shdr_enc_key_type {
        SHDR_ENC_KEY_DEV_SPECIFIC = 0,
```

(continues on next page)

(continued from previous page)

```
        SHDR_ENC_KEY_CLASS_WIDE = 1,
};
```

REE-FS TA binary formats

TA binary follows the ELF file which normally is stripped as additional symbols etc will be ignored when loading the TA.

Legacy TA binary is formatted as:

```
hash = H(<struct shdr> || <stripped ELF>)
signature = RSA-Sign(hash)
legacy_binary = <struct shdr> || <hash> || <signature> || <stripped ELF>
```

Bootstrap TA binary is formatted as:

```
hash = H(<struct shdr> || <struct shdr_bootstrap_ta> || <stripped ELF>)
signature = RSA-Sign(<hash>)
bootstrap_binary = <struct shdr> || <hash> || <signature> ||
                   <struct shdr_bootstrap_ta> || <stripped ELF>
```

Encrypted TA binary is formatted as:

```
nonce = <unique random value>
ciphertext, tag = AES_GCM(<stripped ELF>)
hash = H(<struct shdr> || <struct shdr_bootstrap_ta> ||
         <struct shdr_encrypted_ta> || <nonce> || <tag> || <stripped ELF>)
signature = RSA-Sign(<hash>)
encrypted_binary = <struct shdr> || <hash> || <signature> ||
                   <struct shdr_bootstrap_ta> ||
                   <struct shdr_encrypted_ta> || <nonce> || <tag> ||
                   <ciphertext>
```

Loading REE-FS TA

A REE TA is loaded into shared memory using a series or RPC in *Loading a REE TA into nonsecure shared memory*. The payload memory is allocated via TEE-supplicant and later freed when the TA has been loaded into secure memory in *Freeing previously allocated nonsecure shared memory*.

Secure Storage TA

These are stored in secure storage. The meta data is stored in a database of all installed TAs and the actual binary is stored encrypted and integrity protected as a separate file in the untrusted REE filesystem (flash). Before these TAs can be loaded they have to be installed first, this is something that can be done during initial deployment or at a later stage.

For test purposes the test program xtest can install a TA into secure storage with the command:

```
$ xtest --install-ta
```

TAs stored in secure storage are kept in a TA database. The TA database consists of a single file with the name `dirf.db` which is stored either in the REE filesystem based secure storage or in RPMB. The file is encrypted and integrity

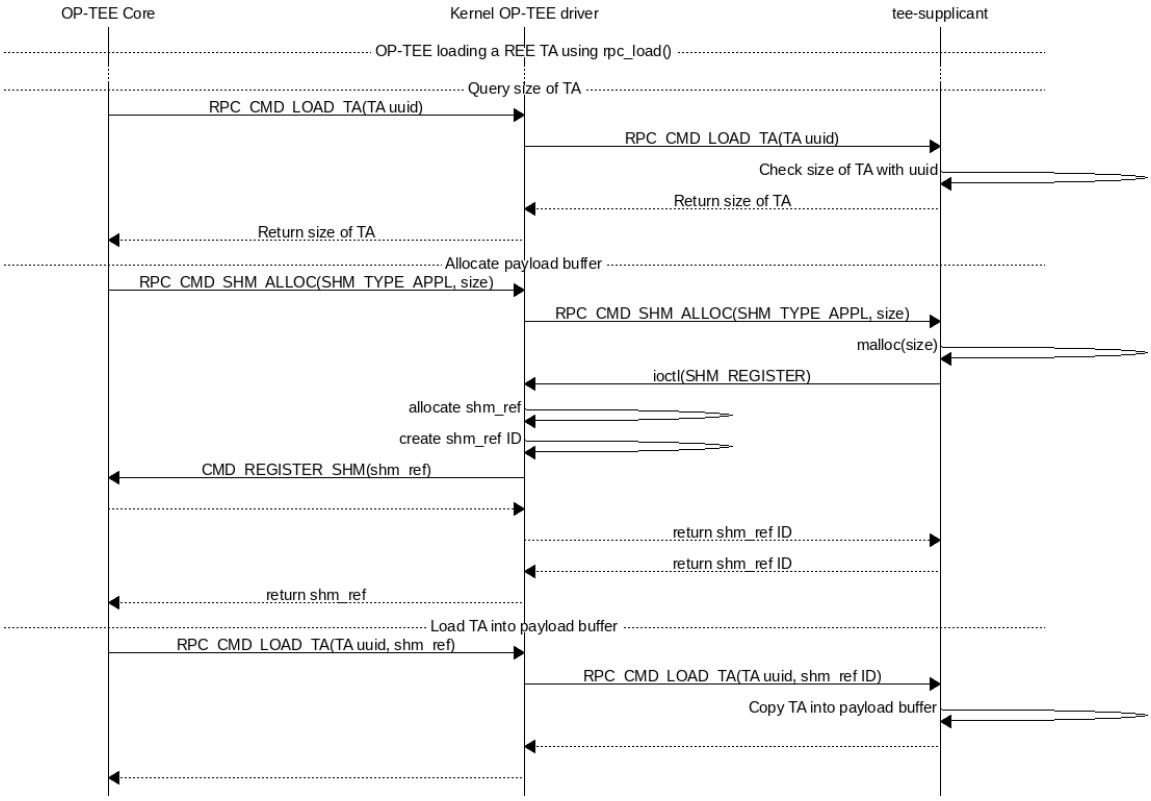


Fig. 9: Loading a REE TA into nonsecure shared memory

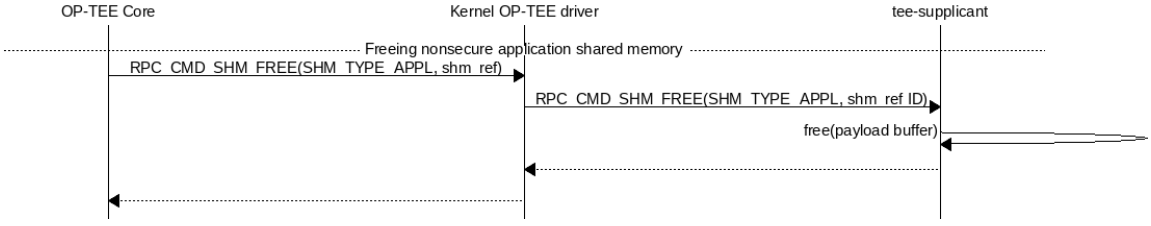


Fig. 10: Freeing previously allocated nonsecure shared memory

protected as any other object in secure storage. The TAs themselves are not stored in `dirf.db`, they are instead stored in the REE filesystem encrypted and integrity protected. One reason for this is that TAs can potentially be quite large, several megabytes, while secure storage is designed to hold only small objects counted in kilobytes.

`dirf.db` constsist of an array of `struct tadb_entry`, defined as:

```
/*
 * struct tee_tadb_property
 * @uuid:       UUID of Trusted Application (TA) or Security Domain (SD)
 * @version:    Version of TA or SD
 * @custom_size:Size of customized properties, prepended to the encrypted
 *              TA binary
 * @bin_size:   Size of the binary TA
 */
struct tee_tadb_property {
        TEE_UUID uuid;
        uint32_t version;
        uint32_t custom_size;
        uint32_t bin_size;
};

#define TADB_IV_SIZE            TEE_AES_BLOCK_SIZE
#define TADB_TAG_SIZE           TEE_AES_BLOCK_SIZE
#define TADB_KEY_SIZE           TEE_AES_MAX_KEY_SIZE

/*
 * struct tadb_entry - TA database entry
 * @prop:        properties of TA
 * @file_number: encrypted TA is stored in <file_number>.ta
 * @iv:          Initialization vector of the authentication crypto
 * @tag:         Tag used to validate the authentication encrypted TA
 * @key:         Key used to decrypt the TA
 */
struct tadb_entry {
        struct tee_tadb_property prop;
        uint32_t file_number;
        uint8_t iv[TADB_IV_SIZE];
        uint8_t tag[TADB_TAG_SIZE];
        uint8_t key[TADB_KEY_SIZE];
};
```

Entries where the `UUID` consists of zeros only are not valid and are ignored. The `file_number` field represents that name of the file stored in the REE filesystem. The filename is made from the decimal string representation of `file_number` with `.ta` appended, or if it was to be printed: `printf("%u.ta", file_number)`.

The TA is decrypted using the authentication encryption algorithm AES-GCM initialized with the `iv` and `key` fields, the `tag` field is used when finalizing the decryption

A TA is looked up in the TA database by opening `dirf.db` and scanning through the elements which are of type `struct tadb_entry` until a matching UUID is found.

2.10.4 Loading and preparing TA for execution

User mode TAs are loaded into final memory in the same way using the user mode ELF loader `ldelf`. The different TA locations has a common interface towards `ldelf` which makes the user mode operations identical regarless of how the TA is stored.

The TA is loaded into secure memory in *Preparing TA for execution*.

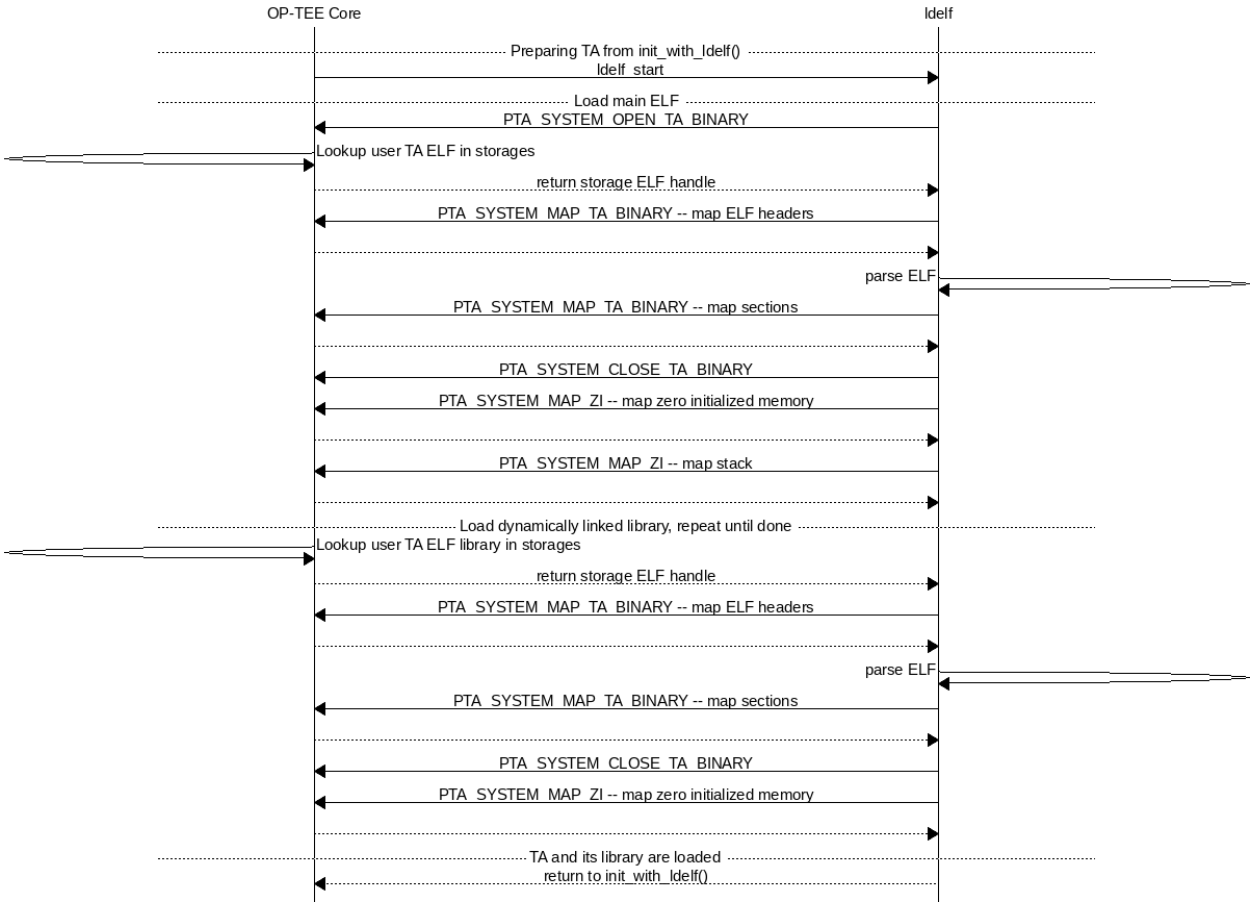


Fig. 11: Preparing TA for execution

After `ldelf` has returned with a TA prepared for execution it still remains in memory to serve the TA if dlopen() and friends are used. `ldelf` is also used to dump stack trace and detailed memory mappings if a TA is terminated via an abort.

A high level view of the entire flow from the client application in Linux user space where a session is opened to a TA is given in *Open session to a TA*.



Fig. 12: Open session to a TA

2.10.5 TA Properties

This section give a more in depth description of the TA properties (see *Trusted Applications* also).

GlobalPlatform Properties

Standard TA properties must be defined through property flag in macro `TA_FLAGS` in `user_ta_header_defines.h`

Single Instance

`"gpd.ta.singleInstance"` is a boolean property of the TA. This property defines if one instance of the TA must be created and will receive all open session request, or if a new specific TA instance must be created for each incoming open session request. OP-TEE TA flag `TA_FLAG_SINGLE_INSTANCE` sets to configuration of this property. The boolean property is set to `true` if `TA_FLAGS` sets bit `TA_FLAG_SINGLE_INSTANCE`, otherwise the boolean property is set to `false`.

Multi-session

"gpd.ta.multiSession" is a boolean property of the TA. This property defines if the TA instance can handle several sessions. If disabled, TA instance support only one session. In such case, if the TA already has a opened session, any open session request will return with a busy error status.

Note: This property is **meaningless** if TA is **NOT** SingleInstance TA.

OP-TEE TA flag TA_FLAG_MULTI_SESSION sets to configuration of this property. The boolean property is set to true if TA_FLAGS sets bit TA_FLAG_MULTI_SESSION, otherwise the boolean property is set to false.

Keep Alive

"gpd.ta.instanceKeepAlive" is a boolean property of the TA. This property defines if the TA instance created must be destroyed or not when all sessions opened towards the TA are closed. If the property is enabled, TA instance, once created (at 1st open session request), is never removed unless the TEE itself is restarted (boot/reboot).

Note: This property is **meaningless** if TA is **NOT** SingleInstance TA.

OP-TEE TA flag TA_FLAG_INSTANCE_KEEP_ALIVE sets to configuration of this property. The boolean property is set to true if TA_FLAGS sets bit TA_FLAG_INSTANCE_KEEP_ALIVE, otherwise the boolean property is set to false.

Heap Size

"gpd.ta.dataSize" is a 32bit integer property of the TA. This property defines the size in bytes of the TA allocation pool, in which TEE_Malloc() and friends allocate memory. The value of the property must be defined by the macro TA_DATA_SIZE in user_ta_header_defines.h (see *TA Properties*).

Stack Size

"gpd.ta.stackSize" is a 32bit integer property of the TA. This property defines the size in bytes of the stack used for TA execution. The value of the property must be defined by the macro TA_STACK_SIZE in user_ta_header_defines.h (see *TA Properties*).

Property Extensions

Secure Data Path Flag

TA_FLAG_SECURE_DATA_PATH is a bit flag supported by TA_FLAGS. This property flag claims the secure data support from the OP-TEE OS for the TA. Refer to the OP-TEE OS for secure data path support. TAs that do not set TA_FLAG_SECURE_DATA_PATH in the value of TA_FLAGS will **not** be able to handle memory reference invocation parameters that relate to secure data path buffers.

Cache maintenance Flag

TA_FLAG_CACHE_MAINTENANCE is a bit flag supported by TA_FLAGS. This property flag, when enabled, allows Trusted Applciation to use the cache maintenance API extension of the Internal Core API described in *Cache Maintenance Support*. TAs that do not set TA_FLAG_CACHE_MAINTENANCE in the value of their TA_FLAGS will not be able to call the cache maintenance API.

Deprecated Property Flags

Older versions of OP-TEE used to define extended property flags that are deprecated and meaningless to current OP-TEE. These are TA_FLAG_USER_MODE, TA_FLAG_EXEC_DDR and TA_FLAG_REMAP_SUPPORT.

2.11 Virtualization

OP-TEE have experimental virtualization support. This is when one OP-TEE instance can run TAs from multiple virtual machines. OP-TEE isolates all VM-related states, so one VM can't affect another in any way.

With virtualization support enabled, OP-TEE will rely on a hypervisor, because only the hypervisor knows which VM is calling OP-TEE. Also, naturally the hypervisor should inform OP-TEE about creation and destruction of VMs. Besides, in almost all cases, hypervisor enables two-stage MMU translation, so VMs does not see real physical address of memory, instead they work with intermediate physical addresses (IPAs). On other hand OP-TEE can't translate IPA to PA, so this is a hypervisor's responsibility to do this kind of translation. So, hypervisor should include a component that knows about OP-TEE protocol internals and can do this translation. We call this component "TEE mediator" and right now only XEN hypervisor have OP-TEE mediator.

2.11.1 Configuration

Virtualization support is enabled with CFG_VIRTUALIZATION configuration option. When this option is enabled, OP-TEE will **not** work without compatible a hypervisor. This is because the hypervisor should send OPTEE_SMC_VM_CREATED SMC with VM ID before any standard SMC can be received from client.

CFG_VIRT_GUEST_COUNT controls the maximum number of supported VMs. As OP-TEE have limited size of available memory, increasing this count will decrease amount of memory available to one VM. Because we want VMs to be independent, OP-TEE splits available memory in equal portions to every VM, so one VM can't consume all memory and cause DoS to other VMs.

2.11.2 Requirements for hypervisor

As said earlier, hypervisor should be aware of OP-TEE and SMCs from virtual guests to OP-TEE. This is a list of things, that compatible hypervisor should perform:

1. When new OP-TEE-capable VM is created, hypervisor should inform OP-TEE about it with SMC OPTEE_SMC_VM_CREATED. a1 parameter should contain VM id. ID 0 is defined as HYP_CLNT_ID and is reserved for hypervisor itself.
2. When OP-TEE-capable VM is being destroyed, hypervisor should stop all VCPUs (this will ensure that OP-TEE have no active threads for that VMs) and send SMC OPTEE_SMC_VM_DESTROYED with the same parameters as for OPTEE_SMC_VM_CREATED.
3. Any SMC to OP-TEE should have VM ID in a7 parameter. This is either HYP_CLNT_ID if call originates from hypervisor or VM ID that was passed in OPTEE_SMC_VM_CREATED call.

4. Hypervisor should perform IPA<->PA address translation for all SMCs. This includes both arguments in `a1-a6` registers and in in-memory command buffers.
5. Hypervisor should pin memory pages that VM shares with OP-TEE. This means, that hypervisor should ensure that pinned page will reside at the original PA as long, as it is shared with OP-TEE. Also it should still belong to the VM that shared it. For example, the hypervisor should not swap out this page, transfer ownership to another VM, unmap it from VM address space and so on.
6. Naturally, the hypervisor should correctly handle the OP-TEE protocol, so for any VM it should look like it is working with OP-TEE directly.

2.11.3 Limitations

Virtualization support is in experimental state and it have some limitations, user should be aware of.

Platforms support

Only Armv8 architecture is supported. There is no hard restriction, but currently Armv7-specific code (like MMU or thread manipulation) just know nothing about virtualization. Only one platform has been tested right now and that is QEMU-V8 (aka qemu that emulates Arm Versatile Express with Armv8 architecture). Support for Rcar Gen3 should be added soon.

Static VMs guest count and memory allocation

Currently, a user should configure maximum number of guests. OP-TEE will split memory into equal chunks, so every VM will have the same amount of memory. For example, if you have 6MB for your TAs, you can set `CFG_VIRT_GUEST_COUNT` to 3 and every VM would be able to use 2MB maximum, even if there is no other VMs running. This is okay for embedded setups when you know exact number and roles of VMs, but can be inconvenient for server applications. Also, it is impossible to configure amount of memory available for a given VM. Every VM instance will have exactly the same amount of memory.

Sharing hardware resources and PTAs

Right now only HW that can be used by multiple VMs simultaneously is serial console, used for logging. Devices like HW crypto accelerators, secure storage devices (e.g. external flash storage, accessed directly from OP-TEE) and others are not supported right now. Drivers should be made virtualization-aware before they can be used with virtualization extensions.

Every VM will have own PTA states, which is a good thing in most cases. But if one wants PTA to have some global state that is shared between VMs, he need to write PTA accordingly.

No compatibility with “normal” mode

OP-TEE built with `CFG_VIRTUALIZATION=y` will not work without a hypervisor, because before executing any standard SMC, `OPTEE_SMC_VM_CREATED` must be called. This can be inconvenient if one wants to switch between virtualized and non-virtualized environment frequently. On other hand, it is not a big deal in a production environment. Simple workaround can be made for this: if OP-TEE receives standard SMC prior to `OPTEE_SMC_VM_CREATED`, it implicitly creates VM context and uses it for all subsequent calls.

Implementation details

OP-TEE as a whole can be split into two entities. Let us call them "nexus" and TEE. Nexus is a core part of OP-TEE that takes care of low level things: SMC handling, memory management, threads creation and so on. TEE is a part that does the actual job: handles requests, loads TAs, executes them, and so on. So, it is natural to have one nexus instance and multiple instances of TEE, one TEE instance per registered VM. This can be done either explicitly or implicitly.

Explicit way is to move TEE state in some sort of structure and make all code to access fields of this structure. Something like `struct task_struct` and `current` in linux kernel. Then it is easy to allocate such structure for every VM instance. But this approach basically requires to rewrite all OP-TEE code.

Implicit way is to have banked memory sections for TEE/VM instances. So memory layout can look something like that:

```
+-----------------------------------------------+
|           Nexus: .nex_bss, .nex_data, ...     |
+-----------------------------------------------+
|                   TEE states                  |
|                                               |
| VM1 TEE state | VM 2 TEE state | VM 3 TEE state |
| .bss, .data   | .bss, .data    | .bss, .data,   |
+-----------------------------------------------+
```

This approach requires no changes in TEE code and requires some changes into nexus code. So, idea that Nexus state resides in separate sections (`.nex_data`, `.nex_bss`, `.nex_nozi`, `.nex_heap` and others) and is always mapped.

TEE state resides in standard sections (like `.data`, `.bss`, `.heap` and so on). There is a separate set of this sections for every VM registered and Nexus maps them only when it receives call from corresponding VM.

As Nexus and TEE have separate heaps, `bget` allocator was extended to work with multiple "contexts". `malloc()`, `free()` with friends work with one context. `nex_malloc()` (and other `nex_` functions) were added. They use different context, so now Nexus can use separate heap, which is always mapped into OP-TEE address space. When virtualization support is disabled, all those `nex_` functions are defined to point to standard `malloc()` counterparts.

To change memory mappings in run-time, in MMU code we have added a new entity, named "partition", which is defined by `struct mmu_partition`. It holds information about all page-tables, so the whole MMU mapping can be switched by one write to `TTBR` register.

There is the default partition, it holds MMU state when there is no VM context active, so no TEE state is mapped. When OP-TEE receives `OPTEE_SMC_VM_CREATED` call, it copies default partition into new one and then maps sections with TEE data. This is done by `prepare_memory_map()` function in `virtualization.c`.

When OP-TEE receives STD call it checks that the supplied VM ID is valid and then activates corresponding MMU partition, so TEE code can access its own data. This is basically how virtualization support is working.

CHAPTER 3

Build and run

In this part of the documentation you will find information telling you how to build OP-TEE as a whole developer setup or as individual components. Likewise it will also tell you how to run OP-TEE on various devices.

Since you pressed this, it's likely that you want to know how to build a full OP-TEE developer setup. So a first place to start looking is probably at the "*build*" page to get started.

You may also want to look at "*Q: What is the quickest and easiest way to try OP-TEE?*".

3.1 AOSP

This page contains information that tells how to get OP-TEE up and running on HiKey devices (see *HiKey 620*, *HiKey 960*) together with AOSP. The build is based on the latest OP-TEE release and updated every quarter together with the regular OP-TEE releases.

Note: We **only** use and support this static/stable configuration. If you try using it with the latest available AOSP, there is a risk that both OP-TEE and other parts are not working as expected.

3.1.1 Prerequisites

- You should already be able to build AOSP for Hikey according to the official instructions. Note that the official build is **NOT** part of the OP-TEE build. It is a separate and non-related build used only to verify and make sure that your system has everything needed to build AOSP without any issues.
- Distro should have necessary packages installed, and the repo tool should be installed. Note that AOSP is built with Java. Also make sure that the `mtools` package is installed, which is needed to make the hikey boot image.
- In addition, you will need the pre-requisites necessary to build optee-os.

After following the AOSP setup instructions, the following additional packages from main *Prerequisites* page are needed. Please install them.

3.1.2 Build instructions

```
$ git clone https://github.com/linaro-swg/optee_android_manifest [-b <release_tag>]
# release tags come in the form of X.Y.Z, e.g. 3.8.0
$ cd optee_android_manifest
```

HiKey620 - LeMaker 8GB

```
$ ./sync-p.sh
$ ./build-p.sh
```

HiKey620 - CircuitCo 4GB

```
$ ./sync-p.sh
$ ./build-p.sh -4g
```

HiKey960

```
$ ./sync-p-hikey960.sh
$ ./build-p-hikey960.sh
```

These steps **MUST** finish with **no errors**. For `sync*.sh` scripts, that means there must be no errors prior to the `Sync done!` console output. For `build*.sh` scripts, that means there must be a `#### build completed successfully (MM:SS (mm:ss)) ####` console output! If there are errors, then there is no point in trying to flash the device.

Warning:

- `--force-sync` is used which means you might **lose your work** so save often, save frequent, and save accordingly, especially before running `sync-p.sh` again!
- **Attention!** Do **NOT** use `git clean` with `-x` or `-X` or `-e` option in `optee_android_manifest/`, else risk **losing all files** in the directory!!!

Hint: You can add the `-squashfs` option to `build.sh` option to make `system.img` size smaller, but this will make `/system` read-only, so you won't be able to push files to it.

Currently, only version P is supported. Other existing files are for internal development purposes ONLY and **NOT SUPPORTED**!

3.1.3 Flashing the image

The instructions for flashing the image can be found in detail under `device/linaro/hikey/installer/hikey{960}/README` in the tree.

1. Set jumpers/switches `1-2` and `3-4`, and unset `5-6`.
2. Reset the board. After that, invoke:

HiKey620

```
$ cp -a out/target/product/hikey/*.img device/linaro/hikey/installer/hikey/
$ sudo ./device/linaro/hikey/installer/hikey/flash-all.sh /dev/ttyUSBn
```

HiKey960

```
$ cp -a out/target/product/hikey960/*.img device/linaro/hikey/installer/hikey960/
$ sudo ./device/linaro/hikey/installer/hikey960/flash-all.sh /dev/ttyUSBn
```

where the `/dev/ttyUSBn` device is the one that appears after rebooting with the 3-4 jumper set. Note that the device only remains in this recovery mode for about 90 seconds. If you take too long to run the flash commands, it will need to be reset again. After flashing, unset the 3-4 jumper again to boot normally.

3.1.4 Partial flashing

The last handful of lines in the `flash-all.sh` script flash various images. After modifying and rebuilding Android, it is only necessary to flash *boot*, *system*, *cache*, *vendor* and *userdata*. If you aren't modifying the kernel, *boot* is not necessary, either.

3.1.5 Experimental prebuilts

Available at http://snapshots.linaro.org/android under `android-hikey*` directories. Note that these images do not always work and are **NOT SUPPORTED** as well!

3.1.6 Running xtest

Do NOT try to run `tee-supplicant` as it has already been started automatically as a service! Once booted to the command prompt, `xtest` can be run immediately from the console or an `adb` shell. For more details about running OP-TEE, please see *Run xtest* at *optee_test*.

3.1.7 Running VTS Gtest unit for Gatekeeper and Keymaster (Optional)

On the device after going into the command prompt, run:

```
$ su
$ ./data/nativetest64/VtsHalGatekeeperV1_0TargetTest/VtsHalGatekeeperV1_0TargetTest
$ ./data/nativetest64/VtsHalKeymasterV3_0TargetTest/VtsHalKeymasterV3_0TargetTest
```

Note: These tests need to be run as root.

3.1.8 Enable adb over USB

Boot the device. On serial console:

```
$ su setprop sys.usb.configfs 1
$ stop adbd
$ start adbd
```

3.1.9 Known issues

- If you don't have a monitor or hdmi emulator (dummy plug) connected to the board, you'll see constant errors scrolling on the console. As a workaround, move `android.hardware.graphics.composer@2.1-service.rc` out of `/vendor/etc/init`. Move it back in when working with a monitor again.
- Adb over USB currently doesn't work on HiKey960. As a workaround, use adb over tcpip. See https://bugs.96boards.org/show_bug.cgi?id=502 for details on how to connect. There are still some limitations however. E.g. running `adb shell` or a second `adb` instance will break the current adb tcpip connection. This might be due to unstable WiFi (there are periodic error messages like `wlcore: WARNING corrupted packet in RX: status: 0x1 len: 76`) or just incompleteness of the generic HiKey960 builds under P.

3.2 Device specific information

3.2.1 DeveloperBox

The instructions here will tell how to build OP-TEE for DeveloperBox.

Build instructions

1. Follow the "*Get and build the solution*" in *build* from step 1 to step 3.
2. Initialize EDK2 submodule

```
$ cd <optee-project>/edk2
$ git submodule update --init
```

3. Follow "*Get and build the solution*" step 4 & 5
4. Stage a new OP-TEE update capsule. This updates TF-A, OP-TEE and UEFI.

```
$ fwupdate --apply {50b94ce5-8b63-4849-8af4-ea479356f0e3} \
  > <optee-project>/edk2-platforms/Build/DeveloperBox/RELEASE_GCC5/FV/\
  > SYNQUACERFIRMWAREUPDATECAPSULEFMPPKCS7.Cap
```

> **Hint:** Change `RELEASE_GCC5` to `DEBUG_GCC5` for debug build.

5. Reboot to update.
6. Follow the rest of"*Get and build the solution*" from step 7

3.2.2 FVP

The instructions here will tell how to build and run OP-TEE using Foundation Models.

Build instructions

Start out by following the "*Get and build the solution*" as described in *build*. However, stop before doing "*Step 5 - Build the solution*".

Next you should obtain the Armv8-A Foundation Platform (For Linux Hosts Only). To download FVPs you'll need to log in to Arm Self Service. That binary should be untar'ed to the root of the repo forest, i.e., like this:

<fpv-project>/Foundation_Platformpkg. In the end after cloning all source code, getting the toolchains and "installing" Foundation_Platformpkg you should have a folder structure that looks like this:

```
$ ls -al
drwxrwxr-x 15 jbech jbech 4096 Feb  5 09:10 .
drwxr-xr-x 22 jbech jbech 4096 Jan 15 12:45 ..
drwxrwxr-x 18 jbech jbech 4096 Feb  5 09:10 arm-trusted-firmware
drwxrwxr-x  9 jbech jbech 4096 Feb  5 09:10 build
drwxrwxr-x 15 jbech jbech 4096 Feb  5 09:10 buildroot
drwxrwxr-x 51 jbech jbech 4096 Feb  5 09:10 edk2
drwxrwxr-x  5 jbech jbech 4096 Feb  5 09:10 edk2-platforms
drwxrwxr-x  6 jbech jbech 4096 Mar 15  2018 Foundation_Platformpkg
drwxrwxr-x 15 jbech jbech 4096 Feb  5 09:10 grub
drwxrwxr-x 26 jbech jbech 4096 Feb  5 09:10 linux
drwxrwxr-x  6 jbech jbech 4096 Feb  5 09:10 optee_client
drwxrwxr-x 10 jbech jbech 4096 Feb  5 09:10 optee_examples
drwxrwxr-x 11 jbech jbech 4096 Feb  5 09:10 optee_os
drwxrwxr-x  8 jbech jbech 4096 Feb  5 09:10 optee_test
drwxrwxr-x  7 jbech jbech 4096 Feb  5 09:10 .repo
lrwxrwxrwx  1 jbech jbech   23 Feb  5 09:09 toolchains
```

When this pre-condition met you can simply continue with

```
$ make run
```

and then FVP should build the rootfs and then start the simulation and when you have a terminal you can log in and run xtest (as described at *Step 9 - Run xtest*).

3.2.3 HiKey 620

The instructions here will tell how to run OP-TEE on HiKey 620.

Multiple sources for HiKey and OP-TEE instructions?

First you must understand that the HiKey project as such is led by the 96Boards project. So, if you **aren't** interested in running OP-TEE on the device, then you should stop reading here and instead have a look at the official HiKey documentation.

For OP-TEE using HiKey you will still find information in more than one place. There are a couple of reasons for that.

- **96Boards**: The official 96Boards project used to host some OP-TEE instructions and they include OP-TEE in their official releases.
- **Google**: has an AOSP HiKey branch, where OP-TEE is supported to some extent.
- **Linaro-SWG**: The OP-TEE team has done some work related to AOSP (see the *AOSP* page) and there HiKey has been one of the devices in use.

If you have questions regarding the configurations above, please reach out to the people on the right forum (96Boards, Google and Linaro-SWG).

This particular guide is maintained by the OP-TEE core team and this is what we use when we are doing are stable releases for our OP-TEE developer builds. I.e, for OP-TEE this should be considered as a well maintained guide with a fully working setup.

Supported HiKey boards

There are four different versions of the HiKey board.

Name	Manufacturer	Memory	Flash	Comment
HiKey	CircuitCo	1GB	4GB	Green solder mask
HiKey	LeMaker	1GB	8GB	Black solder mask
HiKey	LeMaker	2GB	8GB	Black solder mask

All of them works, but where differences apply we have default configurations that works for the LeMaker 8GB eMMC versions.

UART adapter board

Everything is configured to use the 96Boards UART Adapter Board. The UART is by default configured to `UART3`. If you don't have any UART adapter board and instead would like to use `UART0`, then you need to change that before building. See `CFG_NW_CONSOLE_UART` and `CFG_NW_CONSOLE_UART` in hikey.mk.

Build instructions

Just follow the "*Get and build the solution*" as described in *build*. The `make flash` step will tell you how you should set the jumpers on the board.

Recovery

If you manage to corrupt the device, so that fastboot doesn't load automatically on boot, then you will need to run the recovery procedure. Basically what you will need to do is use another make target and change some jumpers. All that is described when you run the target:

```
$ make recovery
```

3.2.4 HiKey 960

The instructions here will tell how to run OP-TEE on HiKey 960.

Supported HiKey960 boards

There are two different versions of the HiKey960 board.

Name	Manufacturer	Memory	Flash	Comment
HiKey960	Archermind/LeMaker	3GB	32GB	v2 uses DIP Switches (SW2201)
HiKey960	Archermind/LeMaker	3GB	32GB	v1 uses Jumpers (J2001)

UART adapter board

Everything is configured to use the 96Boards UART Serial adapter. The UART is by default configured to UART6. If you have a v1 board and need to use UART5, then you need to change that before building. See `CFG_CONSOLE_UART` in hikey960.mk.

Build instructions

Just follow the instructions at "*Get and build the solution*". If `make flash` doesn't work, try `make recovery`.

Recovery

If you manage to corrupt the device, such that fastboot doesn't load automatically on boot, then you will need to run the recovery procedure. Basically what you will need to do is use another make target and change some jumpers. All that is described when you run the target:

External guide

https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/plat/hikey960.rst

```
$ make recovery
```

3.2.5 Juno

The instructions here will tell how to run OP-TEE on the Juno board. The instructions has been tested and verified on the `Juno r0` revision (see Juno revisions for more details).

Regular build

First step is to start out by following the instructions in the *Get and build the solution* as described in *build*.

Deploy files on the device

Enter the firmware console on the Juno board and press **enter** to stop the auto boot.

```
ARM V2M_Juno Firmware v1.3.9
Build Date: Nov 11 2015

Time :  12:50:45
Date :  29:03:2016

Press Enter to stop auto boot...
```

Enable `FTP` at the firmware prompt.

```
Cmd> ftp_on
Enabling ftp server...
 MAC address: xxxxxxxxxxxx

 IP address: 192.168.1.158

 Local host name = V2M-JUNO-A2
```

Flash the binary by running

Note: Use the **IP address** from output from previous command.

```
$ make JUNO_IP=192.168.1.158 flash
```

Once all binaries have been transferred, reboot the board:

```
Cmd> reboot
```

Update the flash layout

The flash layout for Juno may need to be updated for the flashing above to work. If flashing fails or if TF-A refuses to boot due to wrong version of the `SCP` binary, then the flash(-layout) needs to be updated. To update the flash please follow the instructions at Arm's old release notes page selecting one of the zips under "Development boards / Juno / Prebuilt configurations" and flash it as described at Run the Arm Platforms deliverables on Juno.

GlobalPlatform testsuite support

Note: Depending on the Juno pre-built configuration, the built `ramdisk.img` size with GlobalPlatform testsuite may exceed its pre-defined Juno flash memory reserved location (`image.txt` file). In that case, you will need to extend the Juno flash block size reserved location for the `ramdisk.img` in the `image.txt` file accordingly and follow the instructions under "5.7.1 Update flash and its layout".

Example

Example with `juno-latest-busybox-uboot.zip`. The current `ramdisk.img` size with GlobalPlatform testsuite is 8.6 MBytes and that is too big to fit in the default configuration, therefore we need to make adjustments to the flash layout. You will do that by making changes to `/JUNO/SITE1/HBI0262B/images.txt`. I.e., from:

```
1 NOR4UPDATE: AUTO                ;Image Update:NONE/AUTO/FORCE
2 NOR4ADDRESS: 0x01800000         ;Image Flash Address
3 NOR4FILE: \SOFTWARE\ramdisk.img ;Image File Name
4 NOR4NAME: ramdisk.img
5 NOR4LOAD: 00000000              ;Image Load Address
6 NOR4ENTRY: 00000000             ;Image Entry Point
```

to extending the *Image Flash Address* to 16MB

```
1 NOR4UPDATE: AUTO                ;Image Update:NONE/AUTO/FORCE
2 NOR4ADDRESS: 0x01000000         ;Image Flash Address
3 NOR4FILE: \SOFTWARE\ramdisk.img ;Image File Name
4 NOR4NAME: ramdisk.img
5 NOR4LOAD: 00000000              ;Image Load Address
6 NOR4ENTRY: 00000000             ;Image Entry Point
```

GCC > 5.x support

Note: In case you are using the **latest version** of the OP-TEE Arm Juno build (i.e., `juno.xml` manifest), then the `ramdisk.img` built with a GCC version newer than 5.x will be bigger than built with older GCC versions. This

means that you will need to update the sections in `image.txt` that tells where various images will start (see the `image.txt` file).

To solve this problem you will need to extend the Juno flash block size reserved location for the `ramdisk.img` and decrease the size for other images in the `image.txt` file accordingly in the same manner as described in the previous section above.

For example with `juno-latest-busybox-uboot.zip`. The current `ramdisk.img` size with GCC 5.x compiler is 29.15MB and therefore we will need to extend that size for that to 32MB. You do that by changing the highlighted ones (i.e., *Image Flash Address*) in file `/JUNO/SITE1/HBI0262B/images.txt`.

```
NOR2UPDATE: AUTO                ;Image Update:NONE/AUTO/FORCE
NOR2ADDRESS: 0x00100000         ;Image Flash Address
NOR2FILE: \SOFTWARE\Image       ;Image File Name
NOR2NAME: norkern               ;Rename kernel to norkern
NOR2LOAD: 00000000              ;Image Load Address
NOR2ENTRY: 00000000             ;Image Entry Point

NOR3UPDATE: AUTO                ;Image Update:NONE/AUTO/FORCE
NOR3ADDRESS: 0x02C00000         ;Image Flash Address
NOR3FILE: \SOFTWARE\juno.dtb    ;Image File Name
NOR3NAME: board.dtb             ;Specify target filename to preserve file extension
NOR3LOAD: 00000000              ;Image Load Address
NOR3ENTRY: 00000000             ;Image Entry Point

NOR4UPDATE: AUTO                ;Image Update:NONE/AUTO/FORCE
NOR4ADDRESS: 0x00D00000         ;Image Flash Address
NOR4FILE: \SOFTWARE\ramdisk.img ;Image File Name
NOR4NAME: ramdisk.img
NOR4LOAD: 00000000              ;Image Load Address
NOR4ENTRY: 00000000             ;Image Entry Point

NOR5UPDATE: AUTO                ;Image Update:NONE/AUTO/FORCE
NOR5ADDRESS: 0x02D00000         ;Image Flash Address
NOR5FILE: \SOFTWARE\hdlcdclk.dat ;Image File Name
NOR5LOAD: 00000000              ;Image Load Address
NOR5ENTRY: 00000000             ;Image Entry Point
```

On this page you will find device specific information for QEMU v7 (Armv7-A) and QEMU v8 (Armv8-A).

3.2.6 QEMU v7

The instructions here will tell how to run OP-TEE using QEMU for Armv7-A.

Build instructions

As long as you pick the v7 manifest, i.e., `default.xml` the "*Get and build the solution*" tells all you need to know to build and boot up QEMU v7.

Consoles

After running `make run` you will end up in the QEMU console and it will also spawn two UART consoles. One console containing the UART for secure world and one console containing the UART for normal world. You will see that it stops waiting for input on the QEMU console. To continue, do:

```
(qemu) c
```

Host-Guest folder sharing

You can use the VirtFS QEMU feature to avoid changing rootfs CPIO archive every time you need to add additional files or modify existing files. To do this, you share a folder between the guest and host operating systems. To enable and use this feature you have to provide additional arguments when running make, example:

```
$ make QEMU_VIRTFS_ENABLE=y QEMU_USERNET_ENABLE=y
```

Hint: You can also add `QEMU_VIRTFS_HOST_DIR=<share>` in case you don't want to use the default sharing location (which is the root of <qemu-v7-project>).

When QEMU with OP-TEE is up and running, you can mount the host folder in QEMU (normal world UART).

```
# mount -t 9p -o trans=virtio host <mount_point>
```

`<mount_point>` here is folder in the QEMU where you want to mount the host PC's shared folder. So if you want to mount it at `/mnt/host` you typically do this from QEMU NW/UART.

```
# mkdir -p /mnt/host
# mount -t 9p -o trans=virtio host /mnt/host
```

Networking

After booting QEMU, `eth0` will automatically receive an IP address from QEMU via DHCP using the SLiRP user networking feature. QEMU will act as a gateway to the host network SLiRP.

Please note that ICMP won't work in the guest unless additional configuration is made, so the `ping` utility won't work.

GDB - Normal world

If you need to debug a client application, using GDB in a remote debugging configuration may be useful. Remote debugging means `gdb` runs on your PC, where it can access the source code, while the program being debugged runs on the remote system (in this case, in the QEMU environment in normal world). Here is how to do that. On your PC, build with `GDBSERVER=y`:

```
$ cd <qemu-v7-project>/build
# You **only** need to rm -rf the first time you build with the new flag.
# If you omit doing so, it's likely that you will see "stamp" errors in the
# build log.
$ rm -rf <qemu-v7-project>/out-br
$ make -j8 run GDBSERVER=y
```

Boot up as usual

```
(qemu) c
```

Inside QEMU (Normal World UART), run your application with gdbserver (for example `xtest 4002`):

```
# gdbserver :12345 xtest 4002
Process xtest created; pid = 654
Listening on port 12345
```

Back on your PC, open another terminal, start GDB and connect to the target:

```
$ <qemu-v7-project>/out-br/host/bin/arm-buildroot-linux-gnueabihf-gdb
(gdb) set sysroot <qemu-v7-project>/out-br/host/arm-buildroot-linux-gnueabihf/sysroot
(gdb) target remote :12345
```

Now GDB is connected to the remote application. You may use GDB normally.

```
(gdb) b main
(gdb) c
```

GDB - Secure world

TEE core debugging

To debug TEE core running QEMU with GDB, you don't have to enable any special flags as such, but it's easier to debug if you have optimization disabled. Other than that you will have four consoles that you are working with.

- Qemu console
- NW UART console
- SW UART console
- GDB console

All of them but the GDB console are consoles you normally will see/use when running OP-TEE/xtest using QEMU. The first thing is to start QEMU, i.e.,

```
$ cd <qemu-v7-project>/build
# make run-only also works if you don't want to rebuild things
$ make run
```

Next launch another console for GDB and do this

```
$ cd <qemu-v7-project>/toolchains/aarch32/bin
$ ./arm-linux-gnueabihf-gdb -q
```

In the GDB console connect to the QEMU GDB server, like this (the output is included to show what you normally will see).

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00000000 in ?? ()
```

Still in the GDB console, load the symbols for TEE core

```
(gdb) symbol-file <qemu-v7-project>/optee_os/out/arm/core/tee.elf
Reading symbols from <qemu-v7-project>/optee_os/out/arm/core/tee.elf...done.
```

Now you can set a breakpoint for any symbol in OP-TEE, for example

```
(gdb) b tee_entry_std
Breakpoint 1 at 0xe103012: file core/arch/arm/tee/entry_std.c, line 526.
```

Last step is to initiate the boot, do that also from the GDB console

```
(gdb) c
Continuing.
```

At this point will see UART output in the Normal world console as well as the Secure world UART console. If you now for example *Run xtest*, then you will rather soon hit the breakpoint we previously set and you will see something like this in the GDB console:

```
Continuing.
[Switching to Thread 2]

Thread 2 hit Breakpoint 1, tee_entry_std (smc_args=0xe183f18
<stack_thread+8216>) at core/arch/arm/tee/entry_std.c:526
526             struct optee_msg_arg *arg = NULL;       /* fix gcc warning */
(gdb)
```

From here you can start to poke around with GDB, single step, read memory, read registers, print variables and all sorts of things that you normally do with a debugger.

Hint: Some people find it easier to also see the source code while debugging. You can enable the "TUI mode" to see the source code in GDB. To enable that, run GDB with

```
$ ./arm-linux-gnueabihf-gdb -q -tui
```

3.2.7 QEMU v8

The instructions here will tell how to run OP-TEE using QEMU for Armv8-A.

Build instructions

As long as you pick the v8 manifest, i.e., `qemu_v8.xml` the "*Get and build the solution*" tells all you need to know to build and boot up QEMU v8.

All other things (networking, GDB etc) in the v7 section above is also applicable on QEMU v8 as long as you replace `<qemu-v7-project>` with `<qemu-v8-project>` to get the correct paths relative to your QEMU v8 setup.

3.2.8 Raspberry Pi 3

Sequitur Labs did the initial OP-TEE port which at the time also came with modifications in U-Boot, Trusted Firmware A and Linux kernel. Since that initial port more and more patches have found mainline trees and today the OP-TEE setup for Raspberry Pi 3 uses only upstream tree's with the exception of Linux kernel.

Disclaimer

Warning: This port of Trusted Firmware A and OP-TEE to Raspberry Pi 3 **IS NOT SECURE!** Although the Raspberry Pi3 processor provides ARM TrustZone exception states, the mechanisms and hardware required to implement secure boot, memory, peripherals or other secure functions are not available. Use of OP-TEE or TrustZone capabilities within this package **does not result** in a secure implementation. This package is provided solely for **educational purposes** and **prototyping**.

What is expected to work?

First, note that all OP-TEE developer builds (ref, *build*) have rather simple overall goals:

- Successfully build OP-TEE for certain devices.
- Run xtest and optee_example binaries successfully with no regressions using UART(s).

I.e., it is important to understand that our "*OP-TEE developer builds*" shall not be compared with full Linux distributions which supports "everything". As a couple of examples, we don't enable any particular drivers in Linux kernel, we don't include all sorts of daemons, we do not include an X-environment etc. At the same time this doesn't mean that you cannot use OP-TEE in real environments. It is usually perfectly fine to run on all sorts of devices, environments etc. It's just that for the OP-TEE developer builds we have intentionally stripped down the environment to make it rather fast to get all the source code, build it all and run xtest.

We are highlighting this here, since over the years we have had many questions at GitHub about things that people usually find working on their Raspberry Pi devices when they are using Raspbian (which this is not). The table below describes what is *officially* supported in the Raspberry Pi 3 OP-TEE developer builds and right after that follows sections for each of giving a bit more context to it.

Name	Supported?
Buildroot	Yes
HDMI	No
NFS	Yes
Random packages	Maybe
Raspbian	No
Secure boot	Maybe
TFTP	Yes
UART	Yes
Wi-Fi	No

Buildroot

We are using Buildroot as the tool to create a stripped down filesystem for Linux where we also put OP-TEE binaries like Trusted Applications, client libraries and TEE supplicant. If a user wants to add/enable additional packages, then that is also possible by adding new lines in `common.mk` in *build* (search for `BR2_PACKAGE_` in the git to see how it's done).

HDMI

X isn't enabled and we have not built nor enabled any drivers for graphics.

NFS

Works to boot up a Linux root filesystem, more on that further down.

Random packages

See the *Buildroot* section above. You can enable packages supported by Buildroot, but as mentioned initially in this section, lack of drivers and other daemons etc might make it impossible to run.

Raspbian

We are not using it. However, people (from Sequitur Labs) have successfully been able to add OP-TEE to Raspbian builds. But since we're not using it and haven't tried, we simply don't support it.

Secure boot

First pay attention to the initial warning on this page. I.e., no matter what you are doing with Raspberry Pi and TrustZone / OP-TEE you **cannot** make it secure. But that doesn't mean that you cannot "enable" secure features as such for prototyping and to learn how to build and use those. That kind of knowledge can later on be transferred and used on other devices which have all the necessary secure capabilities needed to make a secure system. We haven't tested to enable secure boot on Raspberry Pi 3. But we believe that a good starting point would be Trusted Firmware A's documentation about the "Authentication Framework" and RPi3 in TF-A.

TFTP

When you reach U-Boot (see *Boot sequence*), then you can start using TFTP to load boot firmware etc. Note that if you overwrite `armstub8.bin` for example and that happens to be faulty, then you will need to re-mount the BOOT partition on the SD-card and put a new working version of it. Also note that changing early boot binaries (TF-A, OP-TEE core etc) will require you to reboot the device see the changes.

UART

Fully supported, for more details look at the UART section further down.

Wi-Fi

Even though Raspberry Pi 3 has a Wi-Fi chip, we do not support it in our stripped down builds.

What versions of Raspberry Pi will work?

Below is a table of supported hardware in our OP-TEE developer builds. We have only used the Raspberry Pi 3 Model B, i.e., the first RPi 3 device that was released. But we know that people have successfully been able to use it with both RPi 2's as well as the newer RPi 3 B+. But as long as we in the core team doesn't have those at hands we cannot guarantee anything, therefore we simply say "No" below.

Hardware	Supported?
Raspberry Pi 1 Model A	No
Raspberry Pi 1 Model B	No
Raspberry Pi 1+ Model A	No
Raspberry Pi 1+ Model B	No
Raspberry Pi 2 Model B	No
Raspberry Pi 2 Model B v1.2	No
Raspberry Pi 3+ Model A	No
Raspberry Pi 3 Model B	Yes
Raspberry Pi 3+ Model B	Yes
Raspberry Pi 4	No
Zero - all versions	No
Compute module - all versions	No

Boot sequence

- The **GPU** starts executing the first stage bootloader, which is stored in ROM on the SoC. The first stage bootloader reads the SD-card, and loads the second stage bootloader (`bootcode.bin`) into the L2 cache, and runs it.
- `bootcode.bin` enables SDRAM, and reads the third stage bootloader `loader.bin` from the SD-card into RAM, and runs it.
- `loader.bin` reads the GPU firmware (`start.elf`).
- `start.elf` reads `config.txt`, pre-loads `armstub8.bin` (which contains: BL1/TF-A + BL2/TF-A + BL31/TF-A + BL32/OP-TEE + BL33/U-boot) to `0x0` and jumps to the first instruction.
- A traditional boot sequence of TF-A -> OP-TEE -> U-boot is performed, i.e., BL1 loads BL2, then BL2 loads and run BL31(SM), BL32(OP-TEE), BL33(U-boot) (one after another)
- U-Boot runs `fatload/booti` sequence to load from eMMC to RAM both `zImage` and then `DTB` and boot.

Build instructions

1. Start by following the *Get and build the solution* as described in *build*, but stop at the "*Step 6 - Flash the device*" step (i.e., **don't** run the make flash command!).
2. Next step is to partition and format the memory card and to put the files onto the same. That is something we don't want to automate, since if anything goes wrong, in worst case it might wipe one of your regular hard disks. Instead what we have done, is that we have created another makefile target that will tell you exactly what to do. Run that command and follow the instructions there.

```
$ make img-help
```

> **Note:** The mention of `/dev/sdx1` and `/dev/sdx2` when running the command above are just examples. You need to figure out and replace that with the correct name(s) for your computer and SD-card (typically run `dmesg` and look for the device name matching your SD-card).

3. Put the SD-card back into the Raspberry Pi 3.
4. Plug in the UART cable and attach to the UART

```
$ picocom -b 115200 /dev/ttyUSB0
```

Note: Install picocom if not already installed `$ sudo apt-get install picocom`.

5. Power up the Raspberry Pi 3 and the system shall start booting which you will see on the UART (not *HDMI*).
6. When you have a shell, then it's simply just to follow the "*Step 9 - Run xtest*" instructions.

NFS boot

Booting via NFS is quite useful for several reasons, but the obvious reason when working with Raspberry Pi is that you don't have to move the SD-card back and forth between the host machine and the Raspberry Pi 3 itself when working with **Normal World** files, like Linux kernel and user space programs. Here we will describe how to setup NFS server, so the rootfs can be mounted via NFS.

Warning: This guide doesn't focus on any desktop security, so eventually you would need to harden your setup.

In the description below we will use the following terminology, IP addresses and paths. The reader of this guide is supposed to update this to match his own environment.

```
192.168.1.100   <--- This is your desktop computer (NFS server)
192.168.1.200   <--- This is the Raspberry Pi
/srv/nfs/rpi    <--- Location for the NFS share
```

Configure NFS

Start by installing the NFS server

```
$ sudo apt-get install nfs-kernel-server
```

Then edit the exports file,

```
$ sudo vim /etc/exports
```

In this file you shall tell where your files/folder are and the IP's allowed to access the files. The way it's written below will make it available to every machine on the same subnet (again, be careful about security here). Let's add this line to the file (it's the only line necessary in the file, but if you have several different filesystems available, then you should of course add them too, one line for each share).

```
/srv/nfs/rpi 192.168.1.0/24(rw,sync,no_root_squash,no_subtree_check)
```

Next create the folder where you are going to put the root filesystem

```
$ sudo mkdir /srv/nfs/rpi
```

After this, restart the NFS kernel server

```
$ service nfs-kernel-server restart
```

Hint: To see that your shares are correctly setup and that the NFS server is running, you can run: `$ showmount --all localhost` and you should get a list of `IP:<path>'s` based on what you have added in your exports file. If you get nothing there, then your NFS server hasn't been setup correctly.

Prepare files to be shared

We are now going to put the root filesystem on the location we prepared in the previous section.

Note: The path to the `rootfs.cpio.gz` refers to <rpi3-project>, replace this so it matches your setup.

```
$ cd /srv/nfs/rpi
$ sudo gunzip -cd <rpi3-project>/out-br/images/rootfs.cpio.gz | sudo cpio -idmv
$ sudo rm -rf /srv/nfs/rpi/boot/*
```

uboot.env configuration

The file `uboot.env` contains boot configurations that tells what binaries to load and at what addresses. When using NFS you need to tell U-Boot where the NFS server is located (IP and path). Since the exact IP and path varies for each user, we must update `uboot.env` accordingly.

There are two ways to update `uboot.env`, one is to update `uboot.env.txt` (in *build*) and the other is to update directly from the U-Boot console. Pick the one that you suits your needs. We will cover each of them separately here.

Change uboot.env.txt

In an editor open: `<rpi3-project>/build/rpi3/firmware/uboot.env.txt` and change:

- `nfsserverip` to match the IP address of your NFS server.
- `gatewayip` to the IP address of your router.
- `nfspath` to the exported filesystem in your NFS share.

As an example a section of `uboot.env.txt` could look like this:

```
# NFS/TFTP boot configuraton
gatewayip=192.168.1.1
netmask=255.255.255.0
nfsserverip=192.168.1.100
nfspath=/srv/nfs/rpi
```

Next, you need to re-generate `uboot.env`:

```
$ cd <rpi3-project>/build
$ make u-boot-env-clean
$ make u-boot-env
```

Finally, you need to copy the updated `<rpi3-project>/out/uboot.env` to the **BOOT** partition of your SD-card (mount it as described in *Build instructions* and then just overwrite (`cp`) the file on the **BOOT** partition of your SD-card).

Update u-boot.env from U-Boot console

Boot up the device until you see U-Boot running and counting down, then hit any key and will see the `U-Boot>` prompt. You can then update the `nfsserverip`, `gatewayip` and `nfspath` by writing

```
U-Boot> setenv nfsserverip '192.168.1.100'
U-Boot> setenv gatewayip '192.168.1.1'
U-Boot> setenv nfspath '/srv/nfs/rpi'
```

If you want those environment variables to persist between boots, then type.

```
U-Boot> saveenv
```

Boot up with NFS

With all preparations above done correctly, you should now be able to boot up the device and kernel, secure side OP-TEE and the entire root filesystem should be loaded from the network shares (NFS). Power up the Raspberry, halt in U-Boot and then type.

```
U-Boot> run nfsboot
```

If everything works, you can simply copy paste files like `xtest`, Trusted Applications and other things that usually resides on the host PC's filesystem, i.e., directly from your build folders to the `/srv/nfs/rpi/...` folders. By doing so you don't have to reboot the device when doing development and testing. Just rebuild and copy is sufficient.

Note: You **cannot** make symlinks in the NFS share to the built files, i.e., you must copy them!

JTAG

To enable JTAG you need to add a line saying `enable_jtag_gpio=1` in `config.txt`. There are two ways you can do this, both requires that you to mount the **BOOT** partition on the SD-card at your computer (see the `make img-help` step under *Build instructions*). **After** you have mounted the BOOT partition continue with whichever way is most suitable for you.

Change config.txt directly

With your editor, open `/media/boot/config.txt` and add a line `enable_jtag_gpio=1`, save the file, unmount the BOOT partition and you're good to go after rebooting the device.

Rebuild and untar

1. With your editor, open `<rpi3-project>/build/rpi3/firmware/config.txt` and add a line `enable_jtag_gpio=1`, save the file.
2. `$ cd <rpi3-project>/build && make`
3. `$ cd /media`
4. `$ sudo gunzip -cd <rpi3-project>/out-br/images/rootfs.cpio.gz | sudo cpio -idmv "boot/*"`

Note: You didn't forget to mount the BOOT partition before trying this step?

5. Unmount the BOOT partition and you're good to go after rebooting the device.

JTAG/RPi3 cable

We have created our own cables that consists of a standard 20-pin JTAG connector and a 22-pin connector for the Raspberry Pi 3 itself. Then using a ribbon cable we have connected the cables according to the table below (JTAG pin <-> Raspberry Pi 3 Header pin).

JTAG pin	Signal	GPIO	Mode	RPi3 Header pin
1	3v3	N/A	N/A	1
3	nTRST	GPIO22	ALT4	15
5	TDI	GPIO26	ALT4	37
7	TMS	GPIO27	ALT4	13
9	TCK	GPIO25	ALT4	22
11	RTCK	GPIO23	ALT4	16
13	TDO	GPIO24	ALT4	18
18	GND	N/A	N/A	14
20	GND	N/A	N/A	20

Warning: Be careful and cross check the wiring as incorrect wiring might **damage** your device! Also be careful to connect the cable correctly at both ends (don't flip it and don't put it at the wrong pins in the Raspberry Pi 3 side).

UART/RPi3 cable

In addition to the JTAG connections we have also wired up the RX/TX to be able to use the UART. Note, for this you don't need to do JTAG wirings, i.e., it's perfectly fine to just wire up the UART only. There are many ready made cables for this on the net (eBay) and cost almost nothing. Get one of those if you **don't** intend to use JTAG.

UART pin	Signal	GPIO	Mode	RPi3 Header pin
Black (GND)	GND	N/A	N/A	6
White (RXD)	TXD	GPIO14	ALT0	8
Green (TXD)	RXD	GPIO15	ALT0	10

Warning: Be careful and cross check the wiring as incorrect wiring might **damage** your device!

OpenOCD

Build OpenOCD

Before building OpenOCD, ensure that you have the `libusb-dev` installed.

```
$ sudo apt-get install libusb-1.0-0-dev
```

We are using the official OpenOCD release, simply clone that to your computer and then building is like a lot of other software, i.e.,

```
$ git clone http://repo.or.cz/openocd.git
$ cd openocd
$ ./bootstrap
$ ./configure
$ make
```

Note: In recent versions of OpenOCD, the legacy ft2332 support has been depracted. All these devices now uses libftdi instead. From OpenOCD release notes: *"GPL-incompatible FTDI D2XX library support dropped (Presto, OpenJTAG and USB-Blaster I are using libftdi only now)"*.

We leave it up to the reader of this guide to decide if he wants to install it properly (`make install`) or if he will just run it from the tree directly. The rest of this guide will just run it from the tree.

OpenOCD RPi3 configuration file

Unfortunately, the necessary RPi3 OpenOCD config isn't upstreamed yet into the official OpenOCD repository, so you should use the one stored here `<rpi3-project/build/rpi3/debugger/pi3.cfg`.

Running OpenOCD

Depending on the JTAG debugger you are using you'll need to find and use the interface file for that particular debugger. We've been using J-Link debuggers and Bus Blaster successfully. To start an OpenOCD session using a J-Link device you type:

```
$ cd <openocd>
$ ./src/openocd -f ./tcl/interface/jlink.cfg -f <rpi3-project>/build/rpi3/debugger/
↪pi3.cfg
```

For Bus Blaster type:

```
$ ./src/openocd -f ./tcl/interface/ftdi/dp_busblaster.cfg \ -f <rpi3_repo_dir>/build/
↪rpi3/debugger/pi3.cfg
```

To be able to write commands directly to OpenOCD, you simply open up another shell and type:

```
$ nc localhost 4444
```

From there you can set breakpoints, examine memory etc ("`> help`" will give you a list of available commands). Having that said, if you connect to OpenOCD using GDB, then there is not much incentive connecting to OpenOCD directly, since you will be able to do the same in GDB by the `monitor` command.

Use GDB

OpenOCD will by default listen to GDB connections on port `3333`. So after starting OpenOCD, make a connection to GDB.

```
# Ensure that you have "gdb" in your $PATH
$ aarch64-linux-gnu-gdb -q
(gdb) target remote localhost:3333
```

To load symbols you just use the `symbol-file <path/to/my.elf` as usual. For convenience you can create an alias in the `~/.gdbinit` file. For TEE core debugging this works:

```
define jtag_rpi3
  target remote localhost:3333
  symbol-file <rpi3-project>/optee_os/out/arm/core/tee.elf
end
```

So, when running GDB, you simply type: `(gdb) jtag_rpi3` and it will both connect and load the symbols for TEE core. For Linux kernel and other binaries you would do the same.

Debug session example

After making an initial Raspberry Pi 3 build for OP-TEE where you've enabled JTAG, installed and built OpenOCD, connected the JTAG cable, then you're ready for debugging OP-TEE using JTAG on Raspberry 3. Boot up the Raspberry Pi 3 until you are in Linux and ready to run xtest. Start a new shell (on the host machine) where you run OpenOCD:

```
$ cd <openocd>
$ ./src/openocd -f ./tcl/interface/jlink.cfg -f <rpi3-project>/build/rpi3/debugger/
↪pi3.cfg
```

Start another shell, where you run GDB

```
$ <rpi3-project>/toolchains/aarch64/bin/aarch64-linux-gnu-gdb -q
(gdb) target remote localhost:3333
(gdb) symbol-file <rpi3-project>/optee_os/out/arm/core/tee.elf
```

Next, try to set a breakpoint for the function `hmac_init`, here use **hardware** breakpoints (i.e., `hb`)!

```
(gdb) hb hmac_init
Hardware assisted breakpoint 2 at 0x1012a178: file core/lib/libtomcrypt/src/mac/hmac/
↪hmac_init.c, line 65.
(gdb) c
Continuing.
```

In the UART console (RPi3/Linux), run xtest.

```
# xtest
```

And shortly thereafter you will see GDB stops on your breakpoint and from there you can debug using normal GDB commands.

3.2.9 STM32MP1

The instructions here will tell how to run OP-TEE on one of the supported STM32MP1 boards.

Supported boards

Baord Name	Manufacturer	Boot media	Hardware Description
STM32MP157A-DK1	STMicroelectronics	SDcard	Wiki STM32MP157X-DKX
STM32MP157C-DK2	STMicroelectronics	SDcard	Wiki STM32MP157X-DKX
STM32MP157C-EV1	STMicroelectronics	SDCard	Wiki STM32MP157C-EV1

Build instructions

Follow the instructions at "*Get and build the solution*".

Configuration switch `PLATFORM` can be used to specify the target device as listed in table below:

Board Name	Build configuration directive
STM32MP157A-DK1	`PLATFORM=stm32mp1-157A_DK1`
STM32MP157C-DK2	`PLATFORM=stm32mp1-157C_DK2`
STM32MP157C-EV1	`PLATFORM=stm32mp1-157C_EV1`

When the build completes, generated image file sdcard.img can be found in the generated binary images directory `../out/bin/` from build root path. The images is a GPT multipartition image you can raw copy to the target SDcard using a tool like dd.

A usual short fecth/build/load shell sequence is like the one below:

```
$ repo init -u https://github.com/OP-TEE/manifest.git -m stm32mp1.xml
$ repo sync
$ cd build
$ make toolchains
$ make PLATFORM=stm32mp1-157C_DK2 all
$ sudo dd if=../out/bin/sdcard.img of=/dev/sdX conv=fdatasync status=progress
```

3.2.10 Texas Instruments SoCs

The instructions here will tell how to run OP-TEE on Texas Instruments devices. Secure TI devices require a boot image that is authenticated by ROM code to function. Without this, even JTAG remains locked. In order to create a valid boot image for a secure device from TI, the initial public software image must be signed and combined with various headers, certificates, and other binary images.

Information on the details on the complete boot image format can be obtained from Texas Instruments. The tools used to generate boot images for secure devices are part of a secure development package (SECDEV) that can be downloaded from:

> http://www.ti.com/mysecuresoftware (login required)

The secure development package is access controlled due to NDA and export control restrictions. Access must be requested and granted by TI before the package is viewable and downloadable. Contact TI, either online or by way of a local TI representative, to request access.

Regular build

Start out by following the *Get and build the solution* as described in *build*. Stop before the section on flashing the device, this is currently not supported automatically.

Booting the device

SD Card boot

Create two partitions on an SD card, `boot` of type `FAT16` and `rootfs` of type `EXT4`. To prevent accidental data loss we do not attempt this automatically (the RPI3 *Build instructions* use a similar SD card layout, you can refer to that page for details).

Extract the generated rootfs to the `rootfs` partition

```
$ cd <SD card rootfs partition>
$ gunzip -cd <repo directory>/gen_rootfs/filesystem.cpio.gz | sudo cpio -idm
```

Add the bootloader to the `boot` partition

```
$ cd <SD card boot partition>
$ cp <repo directory>/u-boot/u-boot-spl_HS_MLO MLO
$ cp <repo directory>/u-boot/u-boot_HS.img u-boot.img
```

3.2.11 ZynqMP zcu10x and Ultra96

Instructions below show how to run OP-TEE on ZynqMP zcu10x and Ultra96 board.

Supported boards

This makefile supports the following ZynqMP boards:

- zcu102
- zcu104
- zcu106
- Ultra96v1

Setting up the toolchain

This build chain heavily relies on Petalinux 2018.2 therefore the first step is to download and install the Petalinux 2018.2 toolchain from the Xilinx website (Downloads). Then, you have to download the needed BSP file from the Xilinx website (Downloads). You may have to create a free Xilinx account to proceed with the two previous steps.

Configuring and building for zcu102 board

First, create a new directory which will be used as root directory:

```
$ mkdir -p ~/petalinux-optee
$ cd ~/petalinux-optee
```

Then, copy the zcu102 BSP file into the newly created directory:

```
$ cp ~/Downloads/xilinx-zcu102-v2018.2-final.bsp .
```

Git clone the `build` repository of the OP-TEE project and source the Petalinux settings:

```
$ git clone https://github.com/OP-TEE/build
$ cd ./build
$ source /path/to/petalinux/settings.sh
```

Finally, use the following commands to create, patch, configure and build the Petalinux project. Petalinux is a powerful but very slow tool, each command may take a while according to the capabilities of your computer.

```
$ make -f zynqmp.mk
```

Once the last command ends up you are ready to run QEMU tool or to make a bootable SD card. To run QEMU:

```
$ make -f zynqmp.mk qemu
```

QEMU will start and launch Petalinux distribution. At the end of the boot process, log in using username `root` and password `root`. Start the OP-TEE Normal World service and run xtest:

```
$ tee-supplicant -d
$ xtest
```

You can close QEMU session at any time by typing `Ctrl-A+C` and entering the `quit` command.

Configuring and building for other ZynqMP boards

To use this makefile with other supported boards, you have to download the corresponding BSP and add option `PLATFORM` to each make command.

```
$ make -f zynqmp.mk PLATFORM=zcu106
$ make -f zynqmp.mk PLATFORM=zcu106 qemu
```

Hereafter the list of available `PLATFORM`:

- `zcu102`
- `zcu104`
- `zcu106`
- `ultra96-reva`

Warning: On Ultra96 board, UART is not directly available. You have to connect through WIFI Access Point using the procedure detailed here Getting started.

SD card creation

After completion of building process, you can create a bootable SD card. Here, we consider that SD card corresponds to `/dev/sdb`. We will use `gparted` and `e2image` tools.

Using `gparted` or any other partition manager tool create two partitions on the card:

- 1GB FAT32 bootable partition (`/dev/sdb1` hereafter).
- EXT4 partition on the remaining memory space (`/dev/sdb2` hereafter).

Once SD card is partitioned, use the following commands:

```
$ cp /path/to/project/images/linux/BOOT.BIN /dev/sdb1
$ cp /path/to/project/images/linux/image.ub /dev/sdb1
$ sudo e2image -rap /path/to/project/images/linux/rootfs.ext4 /dev/sdb2
```

Now you can use the newly created SD card to boot your board.

Note: Check that your board is actually configured to boot on the SD card.

Building a given version of OP-TEE

By default, the lastest version of OP-TEE is built. If you wish you can build a given version of OP-TEE instead of the last one by using variable `OPTEE_VER` with target `petalinux-config`. See below an example where OP-TEE v3.4.0 is built.

```
$ make -f zynqmp.mk petalinux-create
$ make -f zynqmp.mk OPTEE_VER=3.4.0 petalinux-config
$ make -f zynqmp.mk petalinux-build
```

Customizing the Petalinux distribution

You can customize the Petalinux project (i.e. kernel, rootfs, ...) as any standard Petalinux project. Just enter the project directory and type your commands. For additional information, refer to Petalinux Tool Documentation (UG1144).

3.3 Linux kernel TEE framework

3.4 OP-TEE gits

These are the gits considered as the main OP-TEE gits which together makes up the entire TEE solution.

3.4.1 build

Why this particular git? As it turns out it's totally possible to put together everything on your own. You can build all the individual components, os, client, xtest, Linux kernel, TF-A, TianoCore, QEMU, Buildroot etc and put all the binaries at correct locations and write your own command lines, Makefiles, shell-scripts etc that will work nicely on the devices you are interested in. If you know how to do that, fine, please go a head. But for newcomers it's way to much behind the scenes to be able to setup a working environment. Also, if you for some reason want to run something in an automated way, then you need something else wrapping it up for you.

With this particular git **built.git** our goal is to simply to make it easy for newcomers to get started with OP-TEE using the devices we've listed in this document.

git location

https://github.com/OP-TEE/build

Why repo?

We discussed alternatives, initially we started out with having a simple shell-script, that worked to start with, but after getting more gits in use and support for more devices it started to be difficult to maintain. In the end we ended up choosing between repo from the Google AOSP project and git submodules. No matter which you choose, there will always be some person arguing that one is better than the other. For us we decided to use repo. Not directly for the features itself from repo, but for the ability to simply work with different manifests containing both stable and non-stable release. Using some tips and tricks you can also speed up setup time significantly. For day to day work with commits, branches etc we tend to use git commands directly.

Root filesystem

The rootfs in the builds that we cover here are as small as possible and is based on a stripped down Buildroot configuration adding just enough in the rootfs such that one can:

- Boot OP-TEE.
- Run xtest with no regressions.
- Easily add additional developer tools like, strace, valgrind etc.

Note: As a consequence of enabling "just enough", it is likely that **non-UART** based enviroments won't work out of the box. I.e., if you try to boot up an enviroment using HDMI and connect keyboards and other devices it is likely that things will not work. To make them work, you probably need to rebuild Linux kernel with correct drivers/frameworks enabled and in addition to that enable binaries/daemons in Buildroot that might be necessary (user space tools and drivers).

How do I build using AOSP / OpenEmbedded?

For guides how to build AOSP, please refer to our *AOSP* page. For OpenEmbedded we have no guide ready.

Platforms supported by build.git

Below is a table showing the platforms supported by build.git. OP-TEE as such supports many more platforms. To find out how to run OP-TEE on those, please reach out to the maintainer of that platform directly if you have build related questions etc. Please see the MAINTAINERS file for contact information.

Platform	Composite flag	Publicly available?
ARM Juno Board	`PLATFORM=vexpress-juno`	Yes
ARM Foundation FVP	`PLATFORM=vexpress-fvp`	Yes
DeveloperBox	`PLATFORM=synquacer`	Yes
HiKey Kirin 620	`PLATFORM=hikey`	Yes
HiKey 960	`PLATFORM=hikey-hikey960`	Yes
MediaTek MT8173 EVB Board (deprecated)	`PLATFORM=mediatek-mt8173`	No
Poplar	`PLATFORM=poplar`	Yes
QEMU	`PLATFORM=vexpress-qemu_virt`	Yes
QEMUv8	`PLATFORM=vexpress-qemu_armv8a`	Yes
Raspberry Pi 3	`PLATFORM=rpi3`	Yes
STM32MP157A-DK1	`PLATFORM=stm32mp1-157A_DK1`	Yes
STM32MP157C-DK2	`PLATFORM=stm32mp1-157C_DK2`	Yes
STM32MP157C-EV1	`PLATFORM=stm32mp1-157C_EV1`	Yes
Texas Instruments DRA7xx	`PLATFORM=ti-dra7xx`	Yes
Texas Instruments AM57xx	`PLATFORM=ti-am57xx`	Yes
Texas Instruments AM43xx	`PLATFORM=ti-am43xx`	Yes

Manifests

Current version

Here is a list of manifests for the devices currently supported in `build.git`. With these you will get a setup containing the all necessary software components to run OP-TEE on the chosen device. Beware that this will run latest available on OP-TEE gits meaning that if you re-sync then you will most likely get new commits. If you need a stable/tagged version with non-moving gits, then please refer to the next section instead.

Target	Manifest xml	Device documentation
AM43xx	`am43xx.xml`	*Texas Instruments SoCs*
AM57xx	`am57xx.xml`	*Texas Instruments SoCs*
DeveloperBox	`synquacer.xml`	*DeveloperBox*
ARM Juno board	`juno.xml`	*Juno*
DRA7xx	`dra7xx.xml`	*Texas Instruments SoCs*
FVP	`fvp.xml`	*FVP*
HiKey 960	`hikey960.xml`	*HiKey 960*
HiKey	`hikey.xml`	*HiKey 620*
Poplar Debian	`poplar.xml`	
QEMU	`default.xml`	*QEMU v7*
QEMUv8	`qemu_v8.xml`	*QEMU v8*
Raspberry Pi 3	`rpi3.xml`	*Raspberry Pi 3*
STM32MP1	`stm32mp1.xml`	*STM32MP1*

Stable releases

Starting from OP-TEE `v3.1` you can check out stable releases by using the same manifests as for current version above, but with the difference that **you also need to specify a branch** where the name corresponds to the release version. I.e., when we are doing releases we are creating a branch with a name corresponding to the release version. So, let's for example say that you want to checkout a stable OP-TEE `v3.4` for Raspberry Pi 3, then you do like this instead of what is mentioned further down in section "*Step 3 - Get the source code*" (note the `-b 3.4.0`):

```
...
$ repo init -u https://github.com/OP-TEE/manifest.git -m rpi3.xml -b 3.4.0
...
```

Stable releases prior to OP-TEE v3.1 (v1.0.0 to v3.0.0)

Before OP-TEE `v3.1` we used to have separate xml-manifest files for the stable builds. If you for some reason need an older stable release, then you can use the `xyz_stable.xml` file corresponding to your device. The way to init `repo` is almost the same as described above, the major difference is the name of manifest being referenced (`-m xyz_stable.xml`) and that we are referring to a tag instead of a branch (`-b refs/tags/MAJOR.MINOR.PATCH`). So as an example, if you need to setup the `2.1.0` stable release for HiKey, then you would do like this instead of what is mentioned further down in section "*Step 3 - Get the source code*".

```
...
repo init -u https://github.com/OP-TEE/manifest.git -m hikey_stable.xml -b refs/tags/
↪2.1.0
...
```

Here is a list of targets and the names of the stable manifests files which were supported by older releases:

Target	Stable manifest xml
AM43xx	`am43xx_stable.xml`
AM57xx	`am57xx_stable.xml`
ARM Juno board	`juno_stable.xml`
DRA7xx	`dra7xx_stable.xml`
FVP	`fvp_stable.xml`
HiKey 960	`hikey960_stable.xml`
HiKey Debian	`hikey_debian_stable.xml`
HiKey	`hikey_stable.xml`
MTK8173	`mt8173-evb_stable.xml`
QEMU	`default_stable.xml`
QEMUv8	`qemu_v8_stable.xml`
Raspberry Pi 3	`rpi3_stable.xml`

Get and build the solution

Below we will describe the general way of how to get the source, build the solution and how to run xtest on the device. For device specific instructions, please see the links in the table in the "*Current version*" section.

Step 1 - Prerequisites

Install prerequisites according to the *Prerequisites* page.

Step 2 - Install Android repo

Note that here you don't install a huge SDK, it's simply a Python script that you download and put in your `$PATH`, that's it. Exactly how to "install" repo, can be found at the Google repo pages, so follow those instructions before continuing.

Step 3 - Get the source code

Choose the manifest corresponding to the platform you intend to use (see the table in section "*Current version*". For example, if you intend to use Raspberry Pi3, then at line 3 below, `${TARGET}.xml` shall be `rpi3.xml`. The `<optee-project>` is whatever location where you want to store the entire OP-TEE developer setup.

```
1 $ mkdir -p <optee-project>
2 $ cd <optee-project>
3 $ repo init -u https://github.com/OP-TEE/manifest.git -m ${TARGET}.xml [-b ${BRANCH}]
4 $ repo sync -j4 --no-clone-bundle
```

Hint: By referencing an existing and locally saved repo forest you can save lots of time. We are talking about doing repo sync in 30 seconds instead of 15-30 minutes (see the *Tips and Tricks* section for more details).

Step 4 - Get the toolchains

In OP-TEE we're using different toolchains for different targets (depends on ARMv7-A ARMv8-A 64/32bit solutions). In any case start by downloading the toolchains by:

```
$ cd <optee-project>/build
$ make -j2 toolchains
```

Step 5 - Build the solution

We've configured our repo manifests, so that repo will always automatically symlink the `Makefile` to the correct device specific makefile, that means that you simply start the build by running (still in `<optee-project>/build`)

```
$ make -j `nproc`
```

This step will also take some time, but you can speed up subsequent builds by enabling ccache (again see *Tips and Tricks*).

Hint: **If you're having build issues**, then you can pipe the entire build log to a file, which makes it easier to search for the issue using a regular editor. In that case also avoid the `-j` flag so it's easier to see in what order things are happening. To create a `build.log` file do: `$ make 2>&1 | tee build.log`

Step 6 - Flash the device

On **non-emulated** solutions (this means that you shouldn't do this step when you are running QEMU-v7/v8 and FVP), you will need to flash the software in some way. We've tried to "hide" that under the following make target:

```
$ make flash
```

But, since some devices are trickier to flash than others, please see the *Device specific information*. See this just as a general instruction.

Step 7 - Boot up the device

This is device specific (see *Device specific information*).

Step 8 - Load tee-supplicant

On **most** solutions tee-supplicant is already running (check by running `$ ps aux | grep tee-supplicant`) on others not. If it's **not** running, then start it by running:

```
$ tee-supplicant -d
```

Note: If you've built using our manifest you should not need to modprobe any OP-TEE/TEE kernel driver since it's built into the kernel in all our setups.

Step 9 - Run xtest

The entire xtest test suite has been deployed when you we're making the builds in previous steps, i.e, in general there is no need to copy any binaries manually. Everything has been put into the *Root filesystem* automatically. So, to run xtest, you simply type:

```
$ xtest
```

If there are no regressions / issues found, xtest should end with something like this:

```
...
+-----------------------------------------------------
23476 subtests of which 0 failed
67 test cases of which 0 failed
0 test case was skipped
TEE test application done!
```

Hint: For other ways to run xtest, please refer to the "*Run xtest*" page at *optee_test*.

Tips and Tricks

Reference existing project to speed up repo sync

Doing a `repo init`, `repo sync` from scratch can take a fair amount of time. The main reason for that is simply because of the size of some of the gits we are using, like for the Linux kernel and EDK2. With repo you can reference an existing forest and by doing so you can speed up repo sync to taking 30 seconds instead of 15-30 minutes. The way to do this are as follows.

1. Start by setup a clean forest that you will not touch, in this example, let us call that `optee-ref` and put that under for `$HOME/devel/optee-ref`. This step will take somewhere between 15- to 45 minutes, depending on your connection speed to internet.
2. Then setup a cronjob (`crontab -e`) that does a `repo sync` in this folder particular folder once a night (that is more than enough).

3. Now you should setup your actual tree which you are going to use as your working tree. The way to do this is almost the same as stated in the instructions above (see the "*Step 3 - Get the source code*" section) , the only difference is that you **also** reference the other local forest when running `repo init`, like this

```
$ repo init -u https://github.com/OP-TEE/manifest.git --reference $HOME/devel/
↪optee-ref
```

4. The rest is the same above, but now it will only take a couple of seconds to clone a forest.

Normally '1' and '2' above is something you will only do once. Also if you ignore step '2', then you will **still** get the latest from official git trees, since repo will also check for updates that aren't at the local reference.

Use ccache

ccache is a tool that caches build object-files etc locally on the disc and can speed up build time significantly in subsequent builds. On Debian-based systems (Ubuntu, Mint etc) you simply install it by running:

```
$ sudo apt-get install ccache
```

The makefiles in build.git are configured to automatically find and use ccache if ccache is installed on your system, so other than having it installed you don't have to think about anything.

3.4.2 manifest

This page contains a couple of guidelines and rules that we want to try to follow when it comes to managing the manifests.

git location

https://github.com/OP-TEE/manifest

Remotes

Since most of our projects can be found on GitHub, we are using that as the main remote. If you need to include other remotes for some reason, then that is OK, but please double check of there is any **maintained** (and preferably official) mirror for the project at GitHub before adding a new remote.

Sections

To have some kind of structure of the files, we have split them up in three sections, one for pure OP-TEE gits, one for OP-TEE supporting gits found at linaro-swg and then a third, `misc` section where everything else can be found. I.e., a template looks like this (this also includes the default remote for clarity):

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
        <remote name="github" fetch="https://github.com" />

        <default remote="github" revision="master" />

        <!-- OP-TEE gits -->
        <!-- linaro-swg gits -->
```

(continues on next page)

(continued from previous page)

```
        <!-- Misc gits -->
</manifest>
```

Project XML elements

All <projects ... > lines should be on the format as shown below with the attributes in this order. The reason for this is to have it uniformly done across all manifests and that it will make it easier when comparing various versions of manifests with diff tools. All three attributes are **mandatory**. The only exception is `revision` which does not have to be stated if it is `master` that we are tracking.

```
<project path="name_and_path_on_disk" name="upstream_name.git" revision="git_revsion"␣
↪/>
```

Alphabetic order

Within each of the three sections, all <project ... > lines **shall** be sorted in alphabetic order (this is again for making it easier to diff manifests). The only expection here is `build.git` which uses the `linkfile` element. Having that at the end makes it look cleaner.

Additional XML attributes

If you are using another remote than the default, then that should come **after** the `revision` attribute (this is true for all attributes other than the `path`, `name` and `revision`).

Alignment of XML attributes

The three mandatory XML attributes `path`, `name` and `revision` should be column aligned. Alignment of additional XML attributes are optional.

When to use clone-depth="1"?

With `clone-depth="1"` you are telling `repo` and `git` that you only want a certain commit and not the entire git log history. You can only use this under two conditions and that is when `revision` is either a branch or a tag. Pure `SHA-1's` does not work and will even raise `repo` and `git` sync errors in some cases. So, the rules are, if you use either `revision="refs/tags/my_tag"` or `revision="refs/heads/my_branch"`, then you shall add `clone-depth="1"` right after the `revision` attribute.

Spaces or tabs?

Only use spaces!

Example

Here is an example showing the basis for an OP-TEE manifest. The names are fictive etc, but it describes everything said above.

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
        <remote name="github" fetch="https://github.com" />
        <remote name="other" fetch="https://someotherlocation.com" />

        <default remote="github" revision="master" />

        <!-- OP-TEE gits -->
        <project path="optee_abc" name="OP-TEE/optee_abc.git" />
        <project path="optee_def" name="OP-TEE/optee_def.git" />

        <!-- linaro-swg gits -->
        <project path="lswg_abc"  name="linaro-swg/lswg-abc.git" revision=
↪"aaaabbbbcccc93e64c2fdd6ae8b0be14a8c45719" />
        <project path="lswg_def"  name="linaro-swg/lswg-def.git" revision=
↪"ddddeeeeffff83e64c2fdd6ae8b0be14a8c45719" />

        <!-- Misc gits -->
        <project path="my_other"  name="my_other.git"            revision="refs/tags/
↪2017.11" clone-depth="1" remote="other" />
</manifest>
```

3.4.3 optee_benchmark

This page describes how to get and build the OP-TEE benchmark framework. For the architectural details, please refer to the *Benchmark framework* page instead.

git location

https://github.com/linaro-swg/optee_benchmark

License

The software is provided under the BSD 2-Clause license.

Build instructions

The benchmark framework spans across different architectural layers and therefore it doesn't make much sense to build it as a standalone build. Therefore we only give guidance telling how to enable it in full OP-TEE developer builds. For general instructions for full OP-TEE developer builds, please refer to instructions at the *build* page. But otherwise follow the instructions below to enable the benchmark framework.

Enable the benchmark framework

Before using Benchmark framework, OP-TEE should be rebuilt with the `CFG_TEE_BENCHMARK` flag enabled so that the benchmark framework will be enabled in all architectural layers. You do that by:

```
$ cd <optee-project>/build
$ make CFG_TEE_BENCHMARK=y
```

Benchmark application usage

When everything has been built (flashed) and you have booted up the device and you have a console ready to accept command, then the next step is to run the actual benchmark application together with the host/TA application you intend to benchmark. You do this my giving the host applicant as argument to the optee_benchmark binary. Let's say for example that you intend to benchmark the *hello_world* example. Then you invoke the benchmark like this:

```
$ benchmark hello_world
```

When client_app finish the execution, optee_benchmark will generate `<client_app>.ts` time stamp data file in the same directory, where Client Application is stored (i.e., relative to *hello_world* in this case).

3.4.4 optee_client

optee_client git contains the source code for the TEE client library in Linux. This component provides the TEE Client API as defined by the GlobalPlatform TEE standard. It is distributed under the BSD 2-clause open source license.

In this git there are two main targets/binaries to build. There is `libteec.so`, which is the library that contains that API for communication with the Trusted OS. Then there is `tee-supplicant` which is a daemon serving the Trusted OS in secure world with miscellaneous features, such as file system access.

git location

https://github.com/OP-TEE/optee_client

License

The software is provided under the BSD 2-Clause license.

Build instructions

You can build the code in this git only or build it as part of the entire system, i.e. as a part of a full OP-TEE developer setup. For the latter, please refer to instructions at the *build* page. For standalone builds we currently support building with both CMake as well as with regular GNU Makefiles.

Configure the toolchain

First step is to download and configure a toolchain, see the *Toolchains* page for instructions.

Clone optee_client

```
$ git clone https://github.com/OP-TEE/optee_client
$ cd optee_client
```

Build using CMake

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_C_COMPILER=arm-linux-gnueabihf-gcc ..
$ make
```

Note: This example uses the 32-bit toolchain (arm-linux-gnueabihf-), the same works using the 64-bit toolchain (aarch64-linux-gnu-).

After this step the compiled binaries can be sound in sub-folders of `build`. If you have a need or preference to install the binaries at some specific location, then on the cmake line above add `-DCMAKE_INSTALL_PREFIX=<my-install-path>` as an additional argument. With that you can then run `make install` and the binaries etc will be copied to the location that you gave as an argument. In this example `/tmp/optee_client`.

```
$ cmake -DCMAKE_C_COMPILER=arm-linux-gnueabihf-gcc -DCMAKE_INSTALL_PREFIX=/tmp/optee_
↪client ..
$ make
$ make install
```

Build using GNU Make

The `Makefile` is configured to use `arm-linux-gnueabihf-` by default.

```
$ make
```

Note: For a 64-bit builds (or any other toolchain) you will need to use `CROSS_COMPILE`.

```
$ make CROSS_COMPILE=aarch64-linux-gnu-
```

After this step the compiled binaries can be found in the sub-folder `out`.

Compiler flags

To be able to see all commands when building you could build using following flags:

GNU Make

```
$ make V=1
```

CMake

```
$ make VERBOSE=1
```

Coding standards

See *Coding standards*.

3.4.5 optee_docs

This is the Git where all official OP-TEE documentation resides and this is what you are reading right now. Here we will give instructions on how to write and build the documentation as well as give some guidelines on what to do and not to do. Note that the documentation is written for Sphinx. So, even though GitHub for example renders `*.rst` files somewhat OK, that is still not the preferred way to read and view the documentation. Instead head over to https://optee.readthedocs.io where the final output is stored and nicely rendered using Sphinx.

git location

https://github.com/OP-TEE/optee_docs

Install Sphinx

Before doing anything else, first install Sphinx and the dependencies.

```
$ sudo apt install graphviz python3-sphinx python3-sphinx-rtd-theme
```

Build optee_docs

```
$ git clone https://github.com/OP-TEE/optee_docs
$ cd optee_docs
$ make html
```

After this step all documentation should have been built and you can open `<optee_docs>/_build/html/index.html` in your browser to see the result and browse the documentation.

Hint: By using a Linux tool called `entr`. You can automatically rebuild the pages your are working with. First get the package `$ sudo apt install entr`, then:

```
$ cd <optee_docs>
$ find . -name "*.rst" | entr -c make html
```

With this, `entr` will automatically rebuild the documentation everytime you make change and save a file. Which means you only have to save the file in your editor and refresh the browser page to see the changes locally.

General guidelines

Linking

Internal links

Internally within a Sphinx project you can link various pages by referring to a keyword specified right above a section, chapter or subsection. This means that you don't have to make hardlinks to certain files. Instead Sphinx will just figure out where it is for you. Example I have to files, file *compiler.rst* and *toolchain.rst*. They could look like this:

compiler.rst example

```
########
Compiler
########
Bla bla bla

.. _compiler_flags:

Compiler Flags
**************
```

toolchain.rst example

```
########
Toolchain
########
Bla bla bla to see find out more about various flags, please refer
:ref:`compiler_flags`.
```

What we can see in the example, is that on line 5 in `toolchain.rst` we refer to the keyword in `compiler.rst` by using `:ref:`compiler_flags``. This would render a direct link to that section in `compiler.rst`.

General recommendation for OP-TEE internal linking

- Things about general things doesn't have to be prefixed with the "document name".
- Things that are specific should be prefixed with the "document name".

Example: the "Contact" section is generic so it's there is no need for prefix. But for example HiKey 620 build instructions are specific to HiKey 620, so there we shall prefix keyword for internal linking.

rst files

The rst files should have descriptive names, but even more important is where you decide to put the files. Even though it's not a problem to move files around, we have to remember that we tend to quite often give links to documentation from at GitHub, emails etc. If we move files, there is a high likelihood that they will become dead links in the future (404's). So think twice before adding a new file or moving an existing file.

Sections, chapters

We have adopted the Sphinx recommended way of using sections, chapters, subsections etc, those are:

- # with overline, for parts
- * with overline, for chapters
- =, for sections
- -, for subsections
- ^, for subsubsections
- ", for paragraphs

3.4.6 optee_examples

This document describes the sample applications that are included in the OP-TEE, that aim to showcase specific functionality and use cases.

For sake of simplicity, all OP-TEE example test application are prefixed with `optee_example_`. All of them works as standalone host and Trusted Application and can be found in separate directories.

git location

https://github.com/linaro-swg/optee_examples

License

The software is provided under the BSD 2-Clause license.

Build instructions

You can build the code in this git only or build it as part of the entire system, i.e. as a part of a full OP-TEE developer setup. For the latter, please refer to instructions at the *build* page. For standalone builds we currently support building with both CMake as well as with regular GNU Makefiles. However, since the both the host and the Trusted Applications have dependencies to files in *optee_client* (libteec.so and headers) as well as *optee_os* (TA-devkit), one **must first** build those and then refer to various files. Below we will show to to build the **hello_world** example for Armv7-A using regular GNU Make.

Configure the toolchain

First step is to download and configure a toolchain, see the *Toolchains* page for instructions.

Build the dependencies

Then you must build *optee_os* as well as *optee_client* first. Build instructions for them can be found on their respective pages.

Clone optee_examples

```
$ git clone https://github.com/linaro-swg/optee_examples.git
```

Build using GNU Make

Host application

```
$ cd optee_examples/hello_world/host
$ make \
    CROSS_COMPILE=arm-linux-gnueabihf- \
    TEEC_EXPORT=<optee_client>/out/export/usr \
    --no-builtin-variables
```

With this you end up with a binary `optee_example_hello_world` in the host folder where you did the build.

Trusted Application

```
$ cd optee_examples/hello_world/ta
$ make \
    CROSS_COMPILE=arm-linux-gnueabihf- \
    PLATFORM=vexpress-qemu_virt \
    TA_DEV_KIT_DIR=<optee_os>/out/arm/export-ta_arm32
```

With this you end up with a files named `uuid.{ta,elf,dmp,map}` etc in the ta folder where you did the build.

Note: For a 64-bit builds (or any other toolchain) you will need to change `CROSS_COMPILE` (and also use a `PLATFORM` corresponding to an Armv8-A configuration).

Coding standards

See *Coding standards*.

Example applications

acipher

Application name	UUID
`optee_example_acipher`	`a734eed9-d6a1-4244-aa50-7c99719e7b7b`

Generates an RSA key pair of specified size and encrypts a supplied string with it using the GlobalPlatform TEE Internal Core API.

aes

Application name	UUID
`optee_example_aes`	`5dbac793-f574-4871-8ad3-04331ec17f24`

Runs an AES encryption and decryption from a TA using the GlobalPlatform TEE Internal Core API. Non secure test application provides the key, initial vector and ciphered data.

hello_world

Application name	UUID
`optee_example_hello_world`	`8aaaf200-2450-11e4-abe2-0002a5d5c51b`

This is a very simple Trusted Application to answer a hello command and incrementing an integer value.

hotp

Application name	UUID
optee_example_hotp	484d4143-2d53-4841-3120-4a6f636b6542

HMAC based One Time Password in OP-TEE

HMAC based One Time Passwords or shortly just 'HOTP' has been around for many years and was initially defined in RFC4226 back in 2005. Since then it has been a popular choice for doing two factor authentication. With the implementation here we are showing how one could leverage OP-TEE for generating such HMAC based One Time Passwords in a secure manner.

Client (OP-TEE) / Server solution

The most common way of using HOTP is in a client/server setup, where the client needs to authenticate itself to be able to get access to some resources on the server. In those cases the server will ask for an One Time Password, the client will generate that and send it over to the server and if the server is OK with the password it will grant access to the client.

Technically how it is working is that the server and the client needs to agree on shared key ('K') and also start from the same counter ('C'). How that is done in practice is another topic, but RFC4226 has some discussion about it. You should at least have a secure channel between the client and the server when sharing the key, but even better would be if you could establish a secure channel all the way down to the TEE (currently we have TCP/UDP support in OP-TEE, but not TLS).

When both the server and the client knows about and use the same key and counter they can start doing client authentication using HOTP. In short what happens is that both the client and the server computes the same HOTP and the server compares the result of both computations (which should be the same to grant access). How that could work can be seen in the sequence diagram below.

In the current implementation we have OP-TEE acting as a client and the server is a remote service running somewhere else. There is no server implemented, but that should be pretty easy to add in a real scenario. The important thing here is to be able to register the shared key in the TEE and to get HOTP values from the TEE on request.

Since the current implementation works as a client we do not need to think about implementing the look-ahead synchronization window ('s') nor do we have to think about adding throttling (which prevents/slows down brute force attacks).

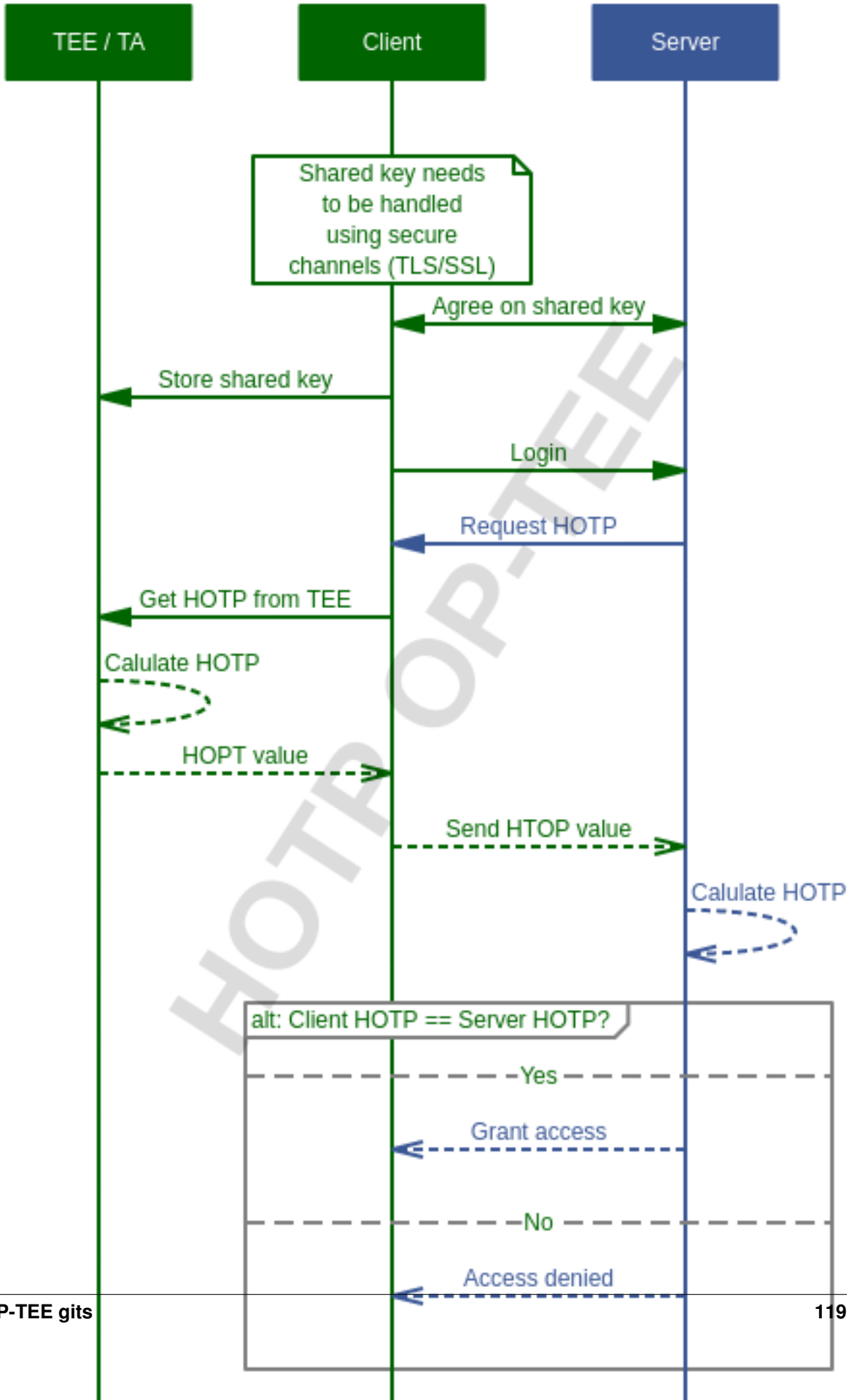
Sequence diagram - Client / Server

Client / Server (OP-TEE)?

Even though the current implementation works as a HOTP client, there is nothing saying that the implementation cannot be updated to also work as the validating server. One could for example have a simple device (a [security token] only generating one time passwords) and use the TEE as a validating service to open up other secure services.

random

Application name	UUID
optee_example_random	b6c53aba-9669-4668-a7f2-205629d00f86


TEE / TA
Client
Server
Shared key needs to be handled using secure channels (TLS/SSL)
Agree on shared key
Store shared key
Login
Request HOTP
Get HOTP from TEE
Calulate HOTP
HOPT value
Send HTOP value
Calulate HOTP
alt: Client HOTP == Server HOTP?
Yes
Grant access
No
Access denied
HOTP OP-TEE

Generates a random UUID using capabilities of TEE API (`TEE_GenerateRandom()`).

secure_storage

Application name	UUID
`optee_example_secure_storage`	`f4e750bb-1437-4fbf-8785-8d3580c34994`

A Trusted Application to read/write raw data into the OP-TEE secure storage using the GlobalPlatform TEE Internal Core API.

Further reading

Some additional information about how to write and compile Trusted Applications can be found at the *Trusted Applications* page.

3.4.7 optee_os

git location

https://github.com/OP-TEE/optee_os

License

The TEE core of optee_os is provided under the BSD 2-Clause license. But there are also other software such as libraries included in optee_os. This "other" software will have different licenses that are compatible with BSD 2-Clause (i.e., non-contaminating licenses unlike GPL-v2 for example).

Build instructions

You can build the code in this git only or build it as part of the entire system, i.e. as a part of a full OP-TEE developer setup. For the latter, please refer to instructions at the *build* page. For standalone builds optee_os uses only regular GNU Makefiles (i.e. **no** CMake support here unlike the other OP-TEE gits).

Configure the toolchain

First step is to download and configure a toolchain, see the *Toolchains* page for instructions.

Clone optee_os

```
$ git clone https://github.com/OP-TEE/optee_os
$ cd optee_os
```

Build using GNU Make

Since optee_os supports many devices and configurations it's impossible to give a examples to all variants. But below is how you for example would build for QEMU running Armv7-A (AArch32), with debugging enabled and the benchmark framework disabled and will put all built files in a folder name *out/arm* in the root of the git.

```
$ make \
    CFG_TEE_BENCHMARK=n \
    CFG_TEE_CORE_LOG_LEVEL=3 \
    CROSS_COMPILE=arm-linux-gnueabihf- \
    CROSS_COMPILE_core=arm-linux-gnueabihf- \
    CROSS_COMPILE_ta_arm32=arm-linux-gnueabihf- \
    CROSS_COMPILE_ta_arm64=aarch64-linux-gnu- \
    DEBUG=1 \
    O=out/arm \
    PLATFORM=vexpress-qemu_virt
```

The same for an QEMU Armv8-A (AArch64) would look like this:

```
$ make \
    CFG_ARM64_core=y \
    CFG_TEE_BENCHMARK=n \
    CFG_TEE_CORE_LOG_LEVEL=3 \
    CROSS_COMPILE=aarch64-linux-gnu- \
    CROSS_COMPILE_core=aarch64-linux-gnu- \
    CROSS_COMPILE_ta_arm32=arm-linux-gnueabihf- \
    CROSS_COMPILE_ta_arm64=aarch64-linux-gnu- \
    DEBUG=1 \
    O=out/arm \
    PLATFORM=vexpress-qemu_armv8a
```

Hint: To be able to see all commands when building you could build with:

```
$ make V=1
```

Build using LLVM/clang

optee_os can be compiled using llvm/clang. Start by downloading the toolchain (see *LLVM / Clang*). After that you can compile by running.

Note: On line one you need to adjust the path so it matches the version of clang you are using.

```
$ export PATH=<optee-project>/toolchains/clang-v9.0.1/bin:$PATH
$ make COMPILER=clang
```

Coding standards

See *Coding standards*.

Build system

The build system in optee_os consists of a main `Makefile` in the root of the project together with `sub.mk` files in all source directories. In addition, some supporting files are used to recursively process all `sub.mk` files and generate the build rules.

Name	Description
`core/core.mk`	Included from `Makefile` to build the TEE Core
`ta/ta.mk`	Included from `Makefile` to create the TA devkit
`mk/compile.mk`	Create rules to make objects from source files
`mk/lib.mk`	Create rules to make a libraries (.a)
`mk/subdir.mk`	Process `sub.mk` files recursively
`mk/config.mk`	Global configuration variable
`core/arch/$(ARCH)/$(ARCH).mk`	Arch-specific compiler flags
`core/arch/$(ARCH)/plat-$(PLATFORM)/conf.mk`	Platform-specific compiler flags and configuration variables
`core/arch/$(ARCH)/plat-$(PLATFORM)/link.mk`	Make recipes to link the TEE Core
`ta/arch/arm/link.mk`	Make recipes to link Trusted Applications
`ta/mk/ta_dev_kit.mk`	Main Makefile to be included when building Trusted Applications
`mk/checkconf.mk`	Utility functions to manipulate configuration variables and generate a C header file
`sub.mk`	List source files and define compiler flags

`make` is always invoked from the top-level directory; there is no recursive invocation of make itself.

Choosing the build target

The target architecture, platform and build directory may be selected by setting environment or make variables (`VAR=value make` or `make VAR=value`).

ARCH - CPU architecture

`$(ARCH)` is the CPU architecture to be built. Currently, the only supported value is `arm` for 32-bit or 64-bit Armv7-A or Armv8-A. Please note that contrary to the Linux kernel, `$(ARCH)` should **not** be set to `arm64` for 64-bit builds. The `ARCH` variable does not need to be set explicitly before building either, because the proper instruction set is selected from the `$(PLATFORM)` value. For platforms that support both 32-bit and 64-bit builds, `CFG_ARM64_core=y` should be set to select 64-bit and not set (or set to `n`) to select 32-bit.

Architecture-specific source code belongs to sub-directories that follow the `arch/$(ARCH)` pattern, such as: `core/arch/arm`, `lib/libutee/arch/arm` and so on.

CROSS_COMPILE

`$(CROSS_COMPILE)` is the prefix used to invoke the (32-bit) cross-compiler toolchain. The default value is `arm-linux-gnueabihf-`. This is the variable you want to change in case you want to use ccache to speed you recompilations:

```
$ make CROSS_COMPILE="ccache arm-linux-gnueabihf-"
```

If the build includes a mix of 32-bit and 64-bit code, for instance if you set `CFG_ARM64_core=y` to build a 64-bit secure kernel, then two different toolchains are used, that are controlled by `$(CROSS_COMPILE32)` and `$(CROSS_COMPILE64)`. The default value of `$(CROSS_COMPILE32)` is the value of `CROSS_COMPILE`, which defaults to `arm-linux-gnueabihf-` as mentioned above. The default value of `$(CROSS_COMPILE64)` is `aarch64-linux-gnu-`. Examples:

```
# For this example, select HiKey which supports both 32- and 64-bit builds
$ export PLATFORM=hikey

# 1. Build everything 32-bit
$ make

# 2. Same as (1.) but override the toolchain
$ make CROSS_COMPILE="ccache arm-linux-gnueabihf-"

# 3. Same as (2.)
$ make CROSS_COMPILE32="ccache arm-linux-gnueabihf-"

# 4. Select 64-bit secure 'core' (and therefore both 32- and 64-bit
# Trusted Application libraries)
$ make CFG_ARM64_core=y

# 5. Same as (4.) but override the toolchains
$ make CFG_ARM64_core=y \
       CROSS_COMPILE32="ccache arm-linux-gnueabihf-" \
       CROSS_COMPILE64="ccache aarch64-linux-gnu-"
```

PLATFORM / PLATFORM_FLAVOR

A *platform* is a family of closely related hardware configurations. A platform *flavor* is a variant of such configurations. When used together they define the target hardware on which OP-TEE will be run.

For instance `PLATFORM=stm PLATFORM_FLAVOR=b2260` will build for the ST Microelectronics 96boards/cannes2 board, while `PLATFORM=vexpress PLATFORM_FLAVOR=qemu_virt` will generate code for a para-virtualized Arm Versatile Express board running on QEMU.

For convenience, the flavor may be appended to the platform name with a dash, so `make PLATFORM=stm-b2260` is a shortcut for `make PLATFORM=stm PLATFORM_FLAVOR=b2260`. Note that in both cases the value of `$(PLATFORM)` is `stm` in the makefiles.

Platform-specific source code belongs to `core/arch/$(ARCH)/plat-$(PLATFORM)`, for instance: `core/arch/arm/plat-vexpress` or `core/arch/arm/plat-stm`.

O - output directory

All output files go into a platform-specific build directory, which is by default `out/$(ARCH)-plat-$(PLATFORM)`.

The output directory has basically the same structure as the source tree. For instance, assuming `ARCH=arm PLATFORM=stm`, `core/kernel/panic.c` will compile into `out/arm-plat-stm/core/kernel/panic.o`.

However, some libraries are compiled several times: once or twice for user mode, and once for kernel mode. This is because they may be used by the TEE Core as well as by the Trusted Applications. As a result, the `lib` source directory gives two or three build directories: `ta_arm{32,64}-lib` and `core-lib`.

The output directory also has an `export-ta_arm{32,64}` directory, which contains:

- All the files needed to build Trusted Applications.
  - In `lib/`: `libutee.a` (the GlobalPlatform Internal API), `libutils.a` (which implements a part of the standard C library), and other libraries which provide additional APIs.
  - In `include/`: header files for the above libraries
  - In `mk/`: `ta_dev_kit.mk`, which is a Make include file with suitable rules to build a TA, and its dependencies
  - `scripts/sign.py`: a Python script used by `ta_dev_kit.mk` to sign TAs.
  - In `src`: `user_ta_header.c`: source file to add a suitable header to the Trusted Application (as expected by the loader code in the TEE Core).
- Some files needed to build host applications (using the Client API), under `export-ta_arm{32,64}/host_include`.

Finally, the build directory contains the auto-generated configuration file for the TEE Core: `$(O)/include/generated/conf.h` (see below).

Configuration and flags

The following variables are defined in `core/arch/$(ARCH)/$(ARCH).mk`:

- `$(core-platform-aflags)`, `$(core-platform-cflags)` and `$(core-platform-cppflags)` are added to the assembler / C compiler / preprocessor flags for all source files compiled for TEE Core including the kernel versions of libraries such as `libutils.a`.
- `$(ta_arm{32,64}-platform-aflags)`, `$(ta_arm{32,64}-platform-cflags)` and `$(ta_arm{32,64}-platform-cppflags)` are added to the assembler / C compiler / preprocessor flags when building the user-mode libraries (`libutee.a`, `libutils.a`) or Trusted Applications.

  The following variables are defined in `core/arch/$(ARCH)/plat-$(PLATFORM)/conf.mk`:

- If `$(arm{32,64}-platform-cflags)`, `$(arm{32,64}-platform-aflags)` and `$(arm{32,64}-platform-cppflags)` are defined their content will be added to `$(\*-platform-\*flags)` when they are are initialized in `core/arch/$(ARCH)/$(ARCH).mk` as described above.
- `$(core-platform-subdirs)` is the list of the subdirectories that are added to the TEE Core.

Linker scripts

The file `core/arch/$(ARCH)/plat-$(PLATFORM)/link.mk` contains the rules to link the TEE Core and perform any related tasks, such as running `objdump` to produce a dump file. `link.mk` adds files to the `all:` target.

Source files

Each directory that contains source files has a file called `sub.mk`. This makefile defines the source files that should be included in the build, as well as any subdirectories that should be processed, too. For example:

```
# core/arch/arm/sm/sub.mk
srcs-y += sm_asm.S
srcs-y += sm.c
```

```
# core/sub.mk
subdirs-y += kernel
subdirs-y += mm
subdirs-y += tee
subdirs-y += drivers
```

The `-y` suffix is meant to facilitate conditional compilation. See section *Configuration variables* below.

`srcs-y` and `subdirs-y` are often not used together in the same `sub.mk`, because source files are usually alone in leaf directories. But this is not a hard rule.

In addition to source files, `sub.mk` may define compiler flags, include directories and/or configuration variables as explained below.

Compiler flags

Default compiler flags are defined in `mk/compile.mk`. Note that platform-specific flags must not appear in this file which is common to all platforms.

To add flags for a given source file, you may use the following variables in `sub.mk`:

- `cflags-<filename>-y` for C files (*.c)
- `aflags-<filename>-y` for assembler files (*.S)
- `cppflags-<filename>-y` for both C and assembler

For instance:

```
# core/lib/libtomcrypt/src/pk/dh/sub.mk
srcs-y += dh.c
cflags-dh.c-y := -Wno-unused-variable
```

Compiler flags may also be removed, as follows:

```
# lib/libutils/isoc/newlib/sub.mk
srcs-y += memmove.c
cflags-remove-memmove.c-y += -Wcast-align
```

Some variables apply to libraries only (that is, when using `mk/lib.mk`) and affect all the source files that belong to the library: `cppflags-lib-y` and `cflags-lib-y`.

Include directories

Include directories may be added to `global-incdirs-y`, in which case they will be accessible from all the source files and will be copied to `export-ta_arm{32,64}/include` and `export-ta_arm{32,64}/host_include`.

When `sub.mk` is used to build a library, `incdirs-lib-y` may receive additional directories that will be used for that library only.

Configuration variables

Some features may be enabled, disabled or otherwise controlled at compile time through makefile variables. Default values are normally provided in makefiles with the `?=` operator so that their value may be easily overridden by environment variables. For instance:

```
PLATFORM ?= stm
PLATFORM_FLAVOR ?= default
```

Some global configuration variables are defined in `mk/config.mk`, but others may be defined in `sub.mk` when then pertain to a specific library for instance.

Variables with the `CFG_` prefix are treated in a special way: their value is automatically reflected in the generated header file `$(out-dir)/include/generated/conf.h`, after all the included makefiles have been processed. `conf.h` is automatically included by the preprocessor when a source file is built.

Depending on their value, variables may be considered either boolean or non-boolean, which affects how they are translated into `conf.h`.

Boolean configuration variables

When a configuration variable controls the presence or absence of a feature, `y` means **enabled**, while `n`, empty value or an undefined variable means **disabled**. For instance, the following commands are equivalent and would disable feature `CFG_CRYPTO_GCM`:

```
$ make CFG_CRYPTO_GCM=n
```

```
$ make CFG_CRYPTO_GCM=
```

```
$ CFG_CRYPTO_GCM=n make
```

```
$ export CFG_CRYPTO_GCM=n
$ make
```

Configuration variables may then be used directly in `sub.mk` to trigger conditional compilation:

```
# core/lib/libtomcrypt/src/encauth/sub.mk
subdirs-$(CFG_CRYPTO_CCM) += ccm
subdirs-$(CFG_CRYPTO_GCM) += gcm
```

When a configuration variable is **enabled** (`y`), `<generated/conf.h>` contains a macro with the same name as the variable and the value `1`. If it is **disabled**, however, no macro definition is output. This allows the C code to use constructs like:

```
/* core/lib/libtomcrypt/src/tee_ltc_provider.c */

/* ... */

#if defined(CFG_CRYPTO_GCM)
struct tee_gcm_state {
        gcm_state ctx;      /* the gcm state as defined by LTC */
        size_t tag_len;     /* tag length */
};
#endif
```

Non-boolean configuration variables

Configuration variables that are not recognized as booleans are simply output unchanged into *<generated/conf.h>*. For instance:

```
$ make CFG_TEE_CORE_LOG_LEVEL=4
```

```
/* out/arm-plat-vexpress/include/generated/conf.h */

#define CFG_TEE_CORE_LOG_LEVEL 4 /* '4' */
```

Configuration dependencies

Some combinations of configuration variables may not be valid. This should be dealt with by custom checks in makefiles. `mk/checkconf.h` provides functions to help detect and deal with such situations.

Import branches

This section is more specifically intended for maintainers.

The optee_os repository contains branches with the `import/` prefix, which we call *import branches* below. This section describes their purpose and how they are used.

Import branches are meant to help import external libraries into the optee_os repository and maintain them:

- *Import* means copy source files from a given upstream version of the library and commit them locally (typically under `optee_os/lib` or `optee_os/core/lib`), along with OP-TEE specific changes (build and configuration files for instance)
- *Maintain* means carry local bug fixes or improvements that did not make their way upstream, and periodically upgrade the library by importing changes from upstream.

Import branches have the version of the imported library in their names. For example: `import/mbedtls-2.6.1`. They are forked from `master`. They record the history of all changes made for OP-TEE to a library for a given library version. For example, the import branch for Mbed TLS 2.6.1 illustrates how the Mbed TLS library was initially imported and later modified:

```
$ BRANCH=github/import/mbedtls-2.6.1
$ BASE=`git merge-base master $BRANCH`
$ git log --oneline --no-merges $BASE..$BRANCH
8ff963a6 (github/import/mbedtls-2.6.1) mbedtls: fix memory leak in mpi_miller_rabin()
213cce52 libmedtls: mpi_miller_rabin: increase count limit
f934e291 mbedtls: add mbedtls_mpi_init_static()
782fddd1 libmbedtls: add mbedtls_mpi_init_mempool()
33873834 libmbedtls: make mbedtls_mpi_mont*() available
e0186224 libmbedtls: refine mbedtls license header
215609ae mbedtls: configure mbedtls to reach for config
6916dcd9 mbedtls: remove default include/mbedtls/config.h
b60fc42a Import mbedtls-2.6.1
```

Commit b60fc42a imports the library under `lib/libmbedtls/mbedtls` with no modification to the code (not the whole library is imported however, since some files are not needed they are deleted as mentioned in the commit description). Then a couple of adjustments are made in commits 6916dcd9 and 215609ae in order to be able to build with optee_os. At this point the initial import is done and subsequent commits are local improvements or bug fixes made later on.

The initial import (commits b60fc42a, 6916dcd9 and 215609ae) is merged into master as a "squashed" commit to preserve bisectability – in other words, so that no commit in master breaks the build:

```
$ git show --quiet 817466cb
commit 817466cb476de705a8e3dabe1ef165fe27a18c2f
Author: Jens Wiklander <jens.wiklander@linaro.org>
Date:   Tue May 22 13:49:31 2018 +0200

    Squashed commit importing mbedtls-2.6.1 source

    Squash merging branch import/mbedtls-2.6.1

    215609ae4d8c ("mbedtls: configure mbedtls to reach for config")
    6916dcd9b9cd ("mbedtls: remove default include/mbedtls/config.h")
    b60fc42a5cd5 ("Import mbedtls-2.6.1")

    Acked-by: Joakim Bech <joakim.bech@linaro.org>
    Signed-off-by: Jens Wiklander <jens.wiklander@linaro.org>
```

Later changes are reviewed and merged into master normally, and are also recorded on top of the import branch via pull requests against the import branch. Consider for example the two following commits, one is in the master branch and the other is applied on top of the import branch.

```
18c5148d357e ("mbedtls: add mbedtls_mpi_init_static()")  # In master
f934e2913b7b ("mbedtls: add mbedtls_mpi_init_static()")  # In import/mbedtls-2.6.1
```

The master branch is occasionally merged into the import branches. Otherwise, some patches cherry-picked from master would not apply. Note that it is a "normal" merge (not a rebase), so the commits that are on master can easily be filtered out (`git log --oneline --first-parent import/...`).

When it is time to upgrade a library, a new import branch is created from master, for example: `import/mbedtls-2.16.0`. A pull request is created against this branch with the following commits:

- The first commit deletes the "old" version of the library and imports the new upstream version.
- The subsequent commits are cherry-picked from the previous import branch and adjusted as needed. These commits are effectively "rebased" onto the new library.
- Build files are updated if needed.

Here is the history of the `import/mbedtls-2.16.0` branch, for comparison with the initial import:

```
$ BRANCH=github/import/mbedtls-2.16.0
$ BASE=`git merge-base master $BRANCH`
$ git log --oneline --no-merges $BASE..$BRANCH
68df6eb0 libmbedtls: mbedtls_mpi_exp_mod(): reduce stack usage
f58facc6 libutee: increase MPI mempool size
be040a3e libmbedtls: preserve mempool usage on reinit
ae499f6a libmbedtls: mbedtls_mpi_exp_mod() initialize W
b95a6c5d libmbedtls: fix no CRT issue
ac34734a libmbedtls: add interfaces in mbedtls for context memory operation
9ee2a92d libmbedtls: compile new files added with 2.16.0
9b0818d4 mbedtls: fix memory leak in mpi_miller_rabin()
2d6644ee libmedtls: mpi_miller_rabin: increase count limit
d831db4c libmbedtls: add mbedtls_mpi_init_mempool()
df0f4886 libmbedtls: make mbedtls_mpi_mont*() available
7b079206 libmbedtls: refine mbedtls license header
2616e2d9 mbedtls: configure mbedtls to reach for config
d686ab1c mbedtls: remove default include/mbedtls/config.h
50a57cfa Import mbedtls-2.16.0
8bfc3de4 libutee: lessen dependency on mbedtls internals
```

Note that the first commit 8bfc3de4 ("libutee: lessen dependency on mbedtls internals") can be ignored, it was applied to master in anticipation of the 2.16.0 upgrade but the `import/mbedtls-2.16.0` was forked before.

The upgrade from 2.6.1 to 2.16.0 is made of all the commits up to and including commit 9ee2a92d ("libmbedtls: compile new files added with 2.16.0"):

- Commit 50a57cfa ("Import mbedtls-2.16.0") deletes the "old" files and library imports the new ones.
- Commits d686ab1c..9b0818d4 are cherry-picked from the previous import branch.
- Commit 9ee2a92d ("libmbedtls: compile new files added with 2.16.0") adapts the build files to the new version.

The master branch contains a squashed equivalent of the above:

```
$ git show --quiet 3d3b0591
commit 3d3b05918ec9052ba13de82fbcaba204766eb636
Author: Jens Wiklander <jens.wiklander@linaro.org>
Date:   Wed Mar 20 15:30:29 2019 +0100

    Squashed commit upgrading to mbedtls-2.16.0

    Squash merging branch import/mbedtls-2.16.0

    9ee2a92de51f ("libmbedtls: compile new files added with 2.16.0")
    9b0818d48d29 ("mbedtls: fix memory leak in mpi_miller_rabin()")
    2d6644ee0bbe ("libmedtls: mpi_miller_rabin: increase count limit")
    d831db4c238a ("libmbedtls: add mbedtls_mpi_init_mempool()")
    df0f4886b663 ("libmbedtls: make mbedtls_mpi_mont*() available")
    7b0792062b65 ("libmbedtls: refine mbedtls license header")
    2616e2d9709f ("mbedtls: configure mbedtls to reach for config")
    d686ab1c51b7 ("mbedtls: remove default include/mbedtls/config.h")
    50a57cfac892 ("Import mbedtls-2.16.0")

    Acked-by: Jerome Forissier <jerome.forissier@linaro.org>
    Signed-off-by: Jens Wiklander <jens.wiklander@linaro.org>
```

Subsequent commits in the `import/mbedtls-2.16.0` branch are modifications that happened later in master as a result of OP-TEE development.

3.4.8 optee_test

The optee_test.git contains the source code for the TEE sanity test suite in Linux using the ARM(R) TrustZone(R) technology. It is typically referred to as *xtest*. By default there are several thousands of tests when running the code that is in the git only. However, it is also possible to incorporate tests coming from GlobalPlatform (see *Extended test (GlobalPlatform tests)*). We typically refer to these to as:

- **Standard tests**: These are the test that are included in optee_test. They are free and open source.
- **Extended tests**: Those are the tests that are written directly by GlobalPlatform. They are **not** open source and they are **not** freely available (it's free to members of GlobalPlatform and can otherwise be purchased directly from GlobalPlatform).

git location

https://github.com/OP-TEE/optee_test

License

The client applications (`optee_test/host/*`) are provided under the GPL-2.0 license and the user Trusted Applications (`optee_test/ta/*`) are provided under the BSD 2-Clause.

Build instructions

At the moment you can **only** build the code in this git as part of the entire system, i.e. as a part of a full OP-TEE developer setup. So, please refer to the instructions at the *build* page to learn how to build a full OP-TEE developer setup. Building purely standalone is **not** possible (*) because:

- the host code (`xtest`) have dependencies to the *optee_client* (it links against `libteec`, `openssl` and uses various headers)
- the Trusted Applications have dependencies to the TA-devkit built by *optee_os*.

Note: (*) It is of course possible to build this without a full OP-TEE developer setup, but it will require a lot of tweaking with paths, flags etc. I.e., one would need to do exactly the same as the full OP-TEE developer setup does under the hood.

Extended test (GlobalPlatform tests)

One can purchase the GlobalPlatform Compliance Test suite which comes with .xml files describing the tests and the Trusted Applications. The standard tests (xtest + TA's) that are free and open source can be extended to also include the GlobalPlatform test suite. This is done by:

- Install the GlobalPlatform `xml` files in `$CFG_GP_PACKAGE_PATH`.
- Run `make patch` (or call make `xtest-patch` from the `build` repository) before compiling xtest. This must be run a single time after the installation of OP-TEE.

This will:

- Create new Trusted Applications, that can be found in `ta/GP_xxx`
- Create new tests in `host/xtest`, as for example `xtest_9000.c`
- Patches `xtest_7000.c`, adding new tests.

Then the tests must be compiled with `CFG_GP_PACKAGE_PATH=<path>`.

It makes use of the following environment variable:

- `COMPILE_NS_USER`: `32` or `64` if application shall be compiled in 32 bits mode on in 64 bits mode. If `COMPILE_NS_USER` is not specified, build relies on `CFG_ARM32_core=y` from OP-TEE core build to assume applications are in 32 bits mode, Otherwise, 64 bits mode is assumed.

Run xtest

It's important to understand that you run `xtest` on the device itself, i.e., this is nothing that you run on the host machine.

xtest - default

The most simple case is to run the default configuration:

```
$ xtest
```

xtest - all

This runs all tests within the standard xtest. Using the `-l` parameter you can tweak the amount of tests you will run. `15` is the most and `0` is the least.

```
$ xtest -l 15
```

xtest - single

To run a single test case, just specify its numbers, for example:

```
$ xtest 1001
```

xtest - family

To run a family (`1xxx`, `2xxx` and so on), just specify its number prefixed with an underscore. This for example will run the 1xxx family.

```
$ xtest _1
```

xtest - benchmark

To run the benchmark tests, run xtest like this:

```
$ xtest -t benchmark
```

Here it is also possible to state a number for a certain benchmark test, for example:

```
$ xtest -t benchmark 2001
```

xtest - regression

To run the regression tests, run xtest like this:

```
$ xtest -t regression
```

Here it is also possible to state a number for a certain regression test, for example:

```
$ xtest -t regression 2004
```

xtest - aes-perf

This is benchmark test for AES and you run it like this:

```
$ xtest --aes-perf
```

Note: There is an individual help for `--aes-perf`, i.e.

```
$ xtest --aes-perf -h
```

xtest - sha-perf

This is benchmark test for SHA-xxx and you run it like this:

```
$ xtest --sha-perf
```

Note: There is an individual help for `--sha-perf`, i.e.

```
$ xtest --sha-perf -h
```

There you can select other SHA algorithms etc.

Coding standards

See *Coding standards*.

The *optee_os* repository is required to run the checks. It's location may be passed using the *OPTEE_OS_PATH* environment variable:

```
export OPTEE_OS_PATH=/path/to/optee_os
```

In case *OPTEE_OS_PATH* is unset or empty, the dispatcher script will default to *../optee_os*.

3.5 Prerequisites

We believe that you can use any Linux distribution to build OP-TEE, but as maintainers of OP-TEE we are mainly using Ubuntu-based distributions and to be able to build and run OP-TEE there are a few packages that needs to be available.

First enable installation of i386 architecture packages and update the package managers database.

```
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
```

Install the following packages regardless of what target you will use in the end.

```
$ sudo apt-get install android-tools-adb android-tools-fastboot autoconf \
        automake bc bison build-essential ccache cscope curl device-tree-compiler \
        expect flex ftp-upload gdisk iasl libattr1-dev libc6:i386 libcap-dev \
        libfdt-dev libftdi-dev libglib2.0-dev libhidapi-dev libncurses5-dev \
```

(continues on next page)

(continued from previous page)

```
        libpixman-1-dev libssl-dev libstdc++6:i386 libtool libz1:i386 make \
        mtools netcat python-crypto python3-crypto python-pyelftools \
        python3-pycryptodome python3-pyelftools python-serial python3-serial \
        rsync unzip uuid-dev xdg-utils xterm xz-utils zlib1g-dev
```

For older versions, you might need to pull *pycryptodome* as a pip package:

```
$ python3 -m pip install --user pycryptodome
```

3.6 Toolchains

OP-TEE uses both 32bit as well as 64bit toolchains and it is even possible to mix them in some configurations. In theory you should be able to compile OP-TEE with the Arm toolchains that are coming with your Linux distribution. But instead of using those directly, we instead download the toolchains directly from Arm.

3.6.1 Download/install

We propose two ways to download the toolchains, both will put the toolchains under the same path(s).

Direct download

Go the Arm GCC download page and download the "*AArch32 target with soft float (arm-linux-gnueabi)*" for 32bit builds and the "*AArch64 GNU/Linux target (aarch64-linux-gnu)*" for 64bit builds. When the downloads have finished, you will untar them to a location that you later on will export to your `$PATH`. Here is an example

```
$ mkdir -p $HOME/toolchains
$ cd $HOME/toolchains

# Download 32bit toolchain
$ wget https://developer.arm.com/-/media/Files/downloads/gnu-a/8.2-2019.01/gcc-arm-8.
↪2-2019.01-x86_64-arm-linux-gnueabi.tar.xz
$ mkdir aarch32
$ tar xf gcc-arm-8.2-2019.01-x86_64-arm-linux-gnueabi.tar.xz -C aarch32 --strip-
↪components=1

# Download 64bit toolchain
$ wget https://developer.arm.com/-/media/Files/downloads/gnu-a/8.2-2019.01/gcc-arm-8.
↪2-2019.01-x86_64-aarch64-linux-gnu.tar.xz
$ mkdir aarch64
$ tar xf gcc-arm-8.2-2019.01-x86_64-aarch64-linux-gnu.tar.xz -C aarch64 --strip-
↪components=1
```

Using build.git

As an alternative, you can let *build*.git download them for you, but this of course involves getting a git that you might not otherwise use.

```
$ cd $HOME
$ git clone https://github.com/OP-TEE/build.git
```

(continues on next page)

(continued from previous page)

```
$ cd build
$ make -f toolchain.mk -j2
```

3.6.2 Export PATH

If you have downloaded the toolchains as described above, you should have them at `$HOME/toolchains/{aarch32/aarch64}`, so now we just need to export the paths and then you are ready to starting compiling OP-TEE components.

```
$ export PATH=$PATH:$HOME/toolchains/aarch32/bin:$HOME/toolchains/aarch64/bin
```

3.6.3 LLVM / Clang

It's possible to also compile *optee_os*.git using llvm/clang. To do that, you can download and extract Clang from the GitHub release page. You'll need an x86_64 cross-compiler capable of generating aarch64 and armv7a code **and** the compiler-rt libraries for these architectures (libclang_rt.*.a).

Clang is configured to be able to cross-compile to all the supported architectures by default (see <clang path>/bin/llc –version) which is great, but compiler-rt is included only for the host architecture. Therefore you need to combine several packages into one. Please refer to this get_clang.sh script for details on creating a llvm/clang toolchain ready to be used.

Using build.git

As an alternative, you can let *build*.git download them for you, but this of course involves getting a git that you might not otherwise use.

```
$ cd $HOME
$ git clone https://github.com/OP-TEE/build.git
$ cd build
$ make -f toolchain.mk clang-toolchains
```

The above instructions will download and install Clang in `$HOME/clang-9.0.1`.

You can also get the toolchain using your package manager or alternatively build it yourself, but these alternative methods risk being incomplete. For example, the Ubuntu clang package does not install the needed ld.lld package. The package also does not contain the cross-compiled compiler-rt libraries. Building by yourself is hard for the same reason, i.e. no cross-compiled compiler-rt libraries are generated.

3.7 Trusted Applications

This document tells how to implement a Trusted Application for OP-TEE, using OP-TEE's so called *TA-devkit* to both build and sign the Trusted Application binary. In this document, a *Trusted Application* running in the OP-TEE os is referred to as a *TA*. Note that in the default setup a private test key is distributed along with the *optee_os* source is used for signing Trusted Applications. See *TASign* for more details, including offline signing of TAs.

3.7.1 TA Mandatory files

The Makefile for a Trusted Application must be written to rely on OP-TEE TA-devkit resources in order to successfully build the target application. TA-devkit is built when one builds *optee_os*.

To build a TA, one must provide:

- **Makefile**, a make file that should set some configuration variables and include the TA-devkit make file.
- **sub.mk**, a make file that lists the sources to build (local source files, subdirectories to parse, source file specific build directives).
- **user_ta_header_defines.h**, a specific ANSI-C header file to define most of the TA properties.
- An implementation of at least the TA entry points, as extern functions: `TA_CreateEntryPoint()`, `TA_DestroyEntryPoint()`, `TA_OpenSessionEntryPoint()`, `TA_CloseSessionEntryPoint()`, `TA_InvokeCommandEntryPoint()`

TA file layout example

As an example, *hello_world* looks like this:

```
hello_world/
├── ...
└── ta
    ├── Makefile                  BINARY=<uuid>
    ├── Android.mk                Android way to invoke the Makefile
    ├── sub.mk                    srcs-y += hello_world_ta.c
    ├── include
    │   └── hello_world_ta.h      Header exported to non-secure: TA commands API
    ├── hello_world_ta.c          Implementation of TA entry points
    └── user_ta_header_defines.h  TA_UUID, TA_FLAGS, TA_DATA/STACK_SIZE, ...
```

3.7.2 TA Makefile Basics

Required variables

The main TA-devkit make file is located in *optee_os* at `ta/mk/ta_dev_kit.mk`. The make file supports make targets such as `all` and `clean` to build a TA or a library and clean the built objects.

The make file expects a couple of configuration variables:

TA_DEV_KIT_DIR Base directory of the TA-devkit. Used by the TA-devkit itself to locate its tools.

BINARY and LIBNAME These are exclusive, meaning that you cannot use both at the same time. If building a TA, `BINARY` shall provide the TA filename used to load the TA. The built and signed TA binary file will be named `${BINARY}.ta`. In native OP-TEE, it is the TA UUID, used by tee-supplicant to identify TAs. If one is building a static library (that will be later linked by a TA), then `LIBNAME` shall provide the name of the library. The generated library binary file will be named `lib${LIBNAME}.a`

CROSS_COMPILE and CROSS_COMPILE32 Cross compiler for the TA or the library source files. `CROSS_COMPILE32` is optional. It allows to target AArch32 builds on AArch64 capable systems. On AArch32 systems, `CROSS_COMPILE32` defaults to `CROSS_COMPILE`.

Optional variables

Some optional configuration variables can be supported, for example:

O Base directory for build objects filetree. If not set, TA-devkit defaults to **./out** from the TA source tree base directory.

Example Makefile

A typical Makefile for a TA looks something like this

```
# Append specific configuration to the C source build (here log=info)
# The UUID for the Trusted Application
BINARY=8aaaf200-2450-11e4-abe2-0002a5d5c51b

# Source the TA-devkit make file
include $(TA_DEV_KIT_DIR)/mk/ta_dev_kit.mk
```

sub.mk directives

The make file expects that current directory contains a file `sub.mk` that is the entry point for listing the source files to build and other specific build directives. Here are a couple of examples of directives one can implement in a sub.mk make file:

```
# Adds /hello_world_ta.c from current directory to the list of the source
# file to build and link.
srcs-y += hello_world_ta.c

# Includes path **./include/** from the current directory to the include
# path.
global-incdirs-y += include/

# Adds directive -Wno-strict-prototypes only to the file hello_world_ta.c
cflags-hello_world_ta.c-y += -Wno-strict-prototypes

# Removes directive -Wno-strict-prototypes from the build directives for
# hello_world_ta.c only.
cflags-remove-hello_world_ta.c-y += -Wno-strict-prototypes

# Adds the static library foo to the list of the linker directive -lfoo.
libnames += foo

# Adds the directory path to the libraries pathes list. Archive file
# libfoo.a is expected in this directory.
libdirs += path/to/libfoo/install/directory

# Adds the static library binary to the TA build dependencies.
libdeps += path/to/greatlib/libgreatlib.a
```

3.7.3 Android Build Environment

OP-TEE's TA-devkit supports building in an Android build environment. One can write an `Android.mk` file for the TA (stored side by side with the Makefile). Android's build system will parse the `Android.mk` file for the TA which in turn will parse a TA-devkit Android make file to locate TA build resources. Then the Android build will execute a `make` command to built the TA through its generic Makefile file.

A typical `Android.mk` file for a TA looks like this (`Android.mk` for *hello_world* is used as an example here).

```
# Define base path for the TA sources filetree
LOCAL_PATH := $(call my-dir)

# Define the module name as the signed TA binary filename.
local_module := 8aaaf200-2450-11e4-abe2-0002a5d5c51b.ta

# Include the devkit Android make script
include $(OPTEE_OS_DIR)/mk/aosp_optee.mk
```

3.7.4 TA Mandatory Entry Points

A TA must implement a couple of mandatory entry points, these are:

```
TEE_Result TA_CreateEntryPoint(void)
{
    /* Allocate some resources, init something, ... */
    ...

    /* Return with a status */
    return TEE_SUCCESS;
}

void TA_DestroyEntryPoint(void)
{
    /* Release resources if required before TA destruction */
    ...
}

TEE_Result TA_OpenSessionEntryPoint(uint32_t ptype,
                                    TEE_Param param[4],
                                    void **session_id_ptr)
{
    /* Check client identity, and alloc/init some session resources if any */
    ...

    /* Return with a status */
    return TEE_SUCCESS;
}

void TA_CloseSessionEntryPoint(void *sess_ptr)
{
    /* check client and handle session resource release, if any */
    ...
}

TEE_Result TA_InvokeCommandEntryPoint(void *session_id,
                                      uint32_t command_id,
                                      uint32_t parameters_type,
                                      TEE_Param parameters[4])
{
    /* Decode the command and process execution of the target service */
    ...

    /* Return with a status */
```

(continues on next page)

(continued from previous page)

```
    return TEE_SUCCESS;
}
```

3.7.5 TA Properties

Trusted Application properties shall be defined in a header file named `user_ta_header_defines.h`, which should contain:

- `TA_UUID` defines the TA uuid value
- `TA_FLAGS` define some of the TA properties
- `TA_STACK_SIZE` defines the RAM size to be reserved for TA stack
- `TA_DATA_SIZE` defines the RAM size to be reserved for TA heap (TEE_Malloc() pool)

Refer to *TA Properties* to understand how to configure these macros.

Hint: UUIDs can be generated using python

```
python -c 'import uuid; print(uuid.uuid4())'
```

or in most Linux systems using either

```
cat /proc/sys/kernel/random/uuid # Linux only
uuidgen # available from the util-linux package in most distributions
```

Example of a property header file

```
#ifndef USER_TA_HEADER_DEFINES_H
#define USER_TA_HEADER_DEFINES_H

#define TA_UUID
    { 0x8aaaf200, 0x2450, 0x11e4, \
        { 0xab, 0xe2, 0x00, 0x02, 0xa5, 0xd5, 0xc5, 0x1b} }

#define TA_FLAGS                    (TA_FLAG_EXEC_DDR | \
                        TA_FLAG_SINGLE_INSTANCE | \
                        TA_FLAG_MULTI_SESSION)
#define TA_STACK_SIZE                       (2 * 1024)
#define TA_DATA_SIZE                        (32 * 1024)

#define TA_CURRENT_TA_EXT_PROPERTIES \
    { "gp.ta.description", USER_TA_PROP_TYPE_STRING, "Foo TA for some purpose." }, \
    { "gp.ta.version", USER_TA_PROP_TYPE_U32, &(const uint32_t){ 0x0100 } }

#endif /* USER_TA_HEADER_DEFINES_H */
```

Note: It is recommended to use the `TA_CURRENT_TA_EXT_PROPERTIES` as above to define extra properties of the TA.

Note: Generating a fresh UUID with suitable formatting for the header file can be done using:

```
python -c "import uuid; u=uuid.uuid4(); print(u); \
   n = [', 0x'] * 11; \
   n[::2] = ['{:12x}'.format(u.node)[i:i + 2] for i in range(0, 12, 2)]; \
   print('\n' + '#define TA_UUID\n\t{ ' + \
         '0x{:08x}'.format(u.time_low) + ', ' + \
         '0x{:04x}'.format(u.time_mid) + ', ' + \
         '0x{:04x}'.format(u.time_hi_version) + ', \\ \n\n\t\t{ ' + \
         '0x{:02x}'.format(u.clock_seq_hi_variant) + ', ' + \
         '0x{:02x}'.format(u.clock_seq_low) + ', ' + \
         '0x' + ''.join(n) + '} }')"
```

3.7.6 Checking TA parameters

GlobalPlatforms TEE Client APIs `TEEC_InvokeCommand()` and `TEE_OpenSession()` allow clients to invoke a TA with some invocation parameters: values or references to memory buffers. It is mandatory that TA's verify the parameters types before using the parameters themselves. For this a TA can rely on the macro `TEE_PARAM_TYPE_GET(param_type, param_index)` to get the type of a parameter and check its value according to the expected parameter.

For example, if a TA expects that command ID 0 comes with `params[0]` being a input value, `params[1]` being a output value, and `params[2]` being a in/out memory reference (buffer), then the TA should implemented the following sequence:

```
TEE_Result handle_command_0(void *session, uint32_t cmd_id,
                            uint32_t param_types, TEE_Param params[4])
{
    if ((TEE_PARAM_TYPE_GET(param_types, 0) != TEE_PARAM_TYPE_VALUE_IN) ||
        (TEE_PARAM_TYPE_GET(param_types, 1) != TEE_PARAM_TYPE_VALUE_OUT) ||
        (TEE_PARAM_TYPE_GET(param_types, 2) != TEE_PARAM_TYPE_MEMREF_INOUT) ||
        (TEE_PARAM_TYPE_GET(param_types, 3) != TEE_PARAM_TYPE_NONE)) {
        return TEE_ERROR_BAD_PARAMETERS
    }

    /* process command */
    ...
}

TEE_Result TA_InvokeCommandEntryPoint(void *session, uint32_t command_id,
                      uint32_t param_types, TEE_Param params[4])
{
    switch (command_id) {
    case 0:
        return handle_command_0(session, param_types, params);

    default:
        return TEE_ERROR_NOT_SUPPORTED;
    }
}
```

3.7.7 Signing of TAs

All *REE Filesystem Trusted Applications* need to be signed. The signature is verified by *optee_os* upon loading of the TA. Within the *optee_os* source is a directory `keys`. The public part of `keys/default_ta.pem` will be compiled into the *optee_os* binary and the signature of each TA will be verified against this key upon loading. Currently `keys/default_ta.pem` must contain an RSA key.

Warning: *optee_os* comes with a default **private** key in its source to facilitate easy development, testing, debugging and QA. Never deploy an *optee_os* binary with this key in production. Instead replace this key as soon as possible with a public key and keep the private part of the key offline, preferably on an HSM.

Note: Currently only a single key for signing TAs is supported by *optee_os*.

TAs are signed using the `sign.py` script referenced from `ta/mk/ta_dev_kit.mk` in *optee_os*. Its default behaviour is to sign a compiled TA binary and attach the signature to form a complete TA for deployment. For **offline** signing, a three-step process is required: In a first step a digest of the compiled binary has to be generated, in the second step this digest is signed offline using the private key and finally in the third step the binary and its signature are stitched together into the full TA.

Offline Signing of TAs

The TA dev kit does sign an application as last step of the linking process. For example, the file `ta/arch/arm/link.mk` in the *optee_os* source tree contains the statement

```
$(q)$(SIGN) --key $(TA_SIGN_KEY) --uuid $(user-ta-uuid) \
        --in $$< --out $$@
```

To avoid build errors when signing offline, this make script needs to be adopted. The signing script can be found at `$(TA_DEV_KIT_DIR)/../scripts/sign.py`

Overall, offline signing is done with the following sequence of steps:

0. (Preparation) Generate a 2048 bit RSA key for signing in a secure, offline environment. Extract the public key and copy it to the `keys` directory in the *optee_os* source tree. Adjust `TA_SIGN_KEY` for different file/path names. (Copy and) modify the `link.mk` file for the default linking step to produce a digest of the TA binary instead of the full TA.

1. Manually (or with the modified linking script) generate a digest of the TA binary using

```
sign.py digest --key $(TA_SIGN_KEY) --uuid $(user-ta-uuid)
```

2. Sign this digest offline, e.g. with OpenSSL

```
base64 --decode digestfile | \
openssl pkeyutl -sign -inkey $TA_SIGN_KEY \
  -pkeyopt digest:sha256 -pkeyopt rsa_padding_mode:pkcs1 | \
base64 > sigfile
```

or with pkcs11-tool using a Nitrokey HSM

```
echo "0000: 3031300D 06096086 48016503 04020105 000420" | \
  xxd -c 19 -r > /tmp/sighdr
cat /tmp/sighdr $(base64 --decode digestfile) > /tmp/hashtosign
pkcs11-tool --id $key_id -s --login -m RSA-PKCS \
```

(continues on next page)

(continued from previous page)

```
  --input-file /tmp/hashtosign | \
  base64 > sigfile
```

3. Manually (or with an extra make target) stitch the TA together using

```
sign.py stitch --key $(TA_SIGN_KEY) --uuid $(user-ta-uuid)
```

By default the UUID is taken as the base file name for all files. Different file names and paths can be set through additional options to `sign.py`. Consult `sign.py --help` for a full list of options and parameters.

CHAPTER 4

Debugging techniques

4.1 Abort dumps / call stack

When OP-TEE encounters a serious error condition, it prints diagnostic information to the secure console. The message contains a call stack if `CFG_UNWIND=y` (enabled by default).

The following errors will trigger a dump:

- Data or prefetch abort exception in the TEE core (kernel mode) or in a TA (user mode),
- When a user-mode Trusted Application panics, either by calling `TEE_Panic()` directly or due to some error detected by the TEE Core Internal API,
- When the TEE core detects a fatal error and decides to hang the system because there is no way to proceed safely (core panic).

The messages look slightly different depending on:

- Whether the error is an exception or a panic,
- The exception/privilege level when the exception occurred (PL0/EL0 if a user mode Trusted Application was running, PL1/EL1 if it was the TEE core),
- Whether the TEE and TA are 32 or 64 bits,
- The exact type of exception (data or prefetch abort, translation fault, read or write permission fault, alignment errors etc).

Here is an example of a panic in a 32-bit Trusted Application, running on a 32-bit TEE core (QEMU):

```
E/TC:0 TA panicked with code 0x0
E/TC:0 Status of TA 484d4143-2d53-4841-3120-4a6f636b6542 (0xe07ba50) (active)
E/TC:0  arch: arm  load address: 0x101000  ctx-idr: 1
E/TC:0  stack: 0x100000 4096
E/TC:0  region 0: va 0x100000 pa 0xe31d000 size 0x1000 flags rw-
E/TC:0  region 1: va 0x101000 pa 0xe300000 size 0xf000 flags r-x
E/TC:0  region 2: va 0x110000 pa 0xe30f000 size 0x3000 flags r--
```

(continues on next page)

(continued from previous page)

```
E/TC:0  region 3: va 0x113000 pa 0xe312000 size 0xb000 flags rw-
E/TC:0  region 4: va 0 pa 0 size 0 flags ---
E/TC:0  region 5: va 0 pa 0 size 0 flags ---
E/TC:0  region 6: va 0 pa 0 size 0 flags ---
E/TC:0  region 7: va 0 pa 0 size 0 flags ---
E/TC:0 Call stack:
E/TC:0  0x001044a8
E/TC:0  0x0010ba59
E/TC:0  0x00101093
E/TC:0  0x001013ed
E/TC:0  0x00101545
E/TC:0  0x0010441b
E/TC:0  0x00104477
D/TC:0 user_ta_enter:452 tee_user_ta_enter: TA panicked with code 0x0
D/TC:0 tee_ta_invoke_command:649 Error: ffff3024 of 3
D/TC:0 tee_ta_close_session:402 tee_ta_close_session(0xe07be98)
D/TC:0 tee_ta_close_session:421 Destroy session
D/TC:0 tee_ta_close_session:447 Destroy TA ctx
```

The above dump was triggered by the TA when entering an irrecoverable error ending up in a `TEE_Panic(0)` call.

OP-TEE provides a helper script called `symbolize.py` to facilitate the analysis of such issues. It is located in the OP-TEE OS source tree in `scripts/symbolize.py` and is also copied to the TA development kit. Whenever you are confronted with an error message reporting a serious error and containing a `"Call stack:"` line, you may use the symbolize script.

`symbolize.py` reads its input from `stdin` and writes extended debug information to `stdout`. The `-d` (directories) option tells the script where to look for TA ELF file(s) (`<uuid>.stripped.elf`) or for `tee.elf` (the TEE core). Please refer to `symbolize.py --help` for details.

Typical output:

```
$ cat dump.txt | ./optee_os/scripts/symbolize.py -d ./optee_examples/*/ta
# (or run the script, copy and paste the dump, then press Ctrl+D)
E/TC:0 TA panicked with code 0x0
E/TC:0 Status of TA 484d4143-2d53-4841-3120-4a6f636b6542 (0xe07ba50) (active)
E/TC:0  arch: arm  load address: 0x101000  ctx-idr: 1
E/TC:0  stack: 0x100000 4096
E/TC:0  region 0: va 0x100000 pa 0xe31d000 size 0x1000 flags rw-
E/TC:0  region 1: va 0x101000 pa 0xe300000 size 0xf000 flags r-x .ta_head .text .
↪rodata
E/TC:0  region 2: va 0x110000 pa 0xe30f000 size 0x3000 flags r-- .rodata .ARM.extab .
↪ARM.extab.text.utee_panic .ARM.extab.text.__aeabi_ldivmod .ARM.extab.text.__aeabi_
↪uldivmod .ARM.exidx .got .dynsym .rel.got .dynamic .dynstr .hash .rel.dyn
E/TC:0  region 3: va 0x113000 pa 0xe312000 size 0xb000 flags rw- .data .bss
E/TC:0  region 4: va 0 pa 0 size 0 flags ---
E/TC:0  region 5: va 0 pa 0 size 0 flags ---
E/TC:0  region 6: va 0 pa 0 size 0 flags ---
E/TC:0  region 7: va 0 pa 0 size 0 flags ---
E/TC:0 Call stack:
E/TC:0  0x001044a8 utee_panic at optee_os/lib/libutee/arch/arm/utee_syscalls_a32.S:74
E/TC:0  0x0010ba59 TEE_Panic at optee_os/lib/libutee/tee_api_panic.c:35
E/TC:0  0x00101093 hmac_sha1 at optee_examples/hotp/ta/hotp_ta.c:63
E/TC:0  0x001013ed get_hotp at optee_examples/hotp/ta/hotp_ta.c:171
E/TC:0  0x00101545 TA_InvokeCommandEntryPoint at optee_examples/hotp/ta/hotp_ta.c:225
E/TC:0  0x0010441b entry_invoke_command at optee_os/lib/libutee/arch/arm/user_ta_
↪entry.c:207
```

(continues on next page)

(continued from previous page)

```
E/TC:0  0x00104477 __utee_entry at optee_os/lib/libutee/arch/arm/user_ta_entry.c:235
D/TC:0 user_ta_enter:452 tee_user_ta_enter: TA panicked with code 0x0 ???
D/TC:0 tee_ta_invoke_command:649 Error: ffff3024 of 3
D/TC:0 tee_ta_close_session:402 tee_ta_close_session(0xe07be98)
D/TC:0 tee_ta_close_session:421 Destroy session
D/TC:0 tee_ta_close_session:447 Destroy TA ctx
```

The Python script uses several tools from the GNU Binutils package to perform the following tasks:

1. Translate the call stack addresses into function names, file names and line numbers.
2. Convert the abort address to a symbol plus some offset and/or an ELF section name plus some offset.
3. Print the names of the ELF sections contained in each memory region of a TA.

Note that to successfully run `symbolize.py` you must also make your toolchain visible on the `PATH` (i.e., `export PATH=<my-toolchain-path>/bin:$PATH`).

4.2 Benchmark framework

Due to its nature, OP-TEE is being a solution spanning over several architectural layers, where each layer includes its own complex parts. For further optimizations of performance, there is a need of tool which will provide detailed and precise profiling information for each layer.

It is necessary to receive latency values for:

- The roundtrip time for going from a client application in normal world, down to a Trusted Application and back again.
- Detailed information for amount of time taken to go through each layer:
  - libTEEC -> Linux OP-TEE kernel driver
  - Linux OP-TEE kernel driver -> OP-TEE OS Core
  - OP-TEE OS Core -> TA entry point (**not supported yet**)
  - The same way back

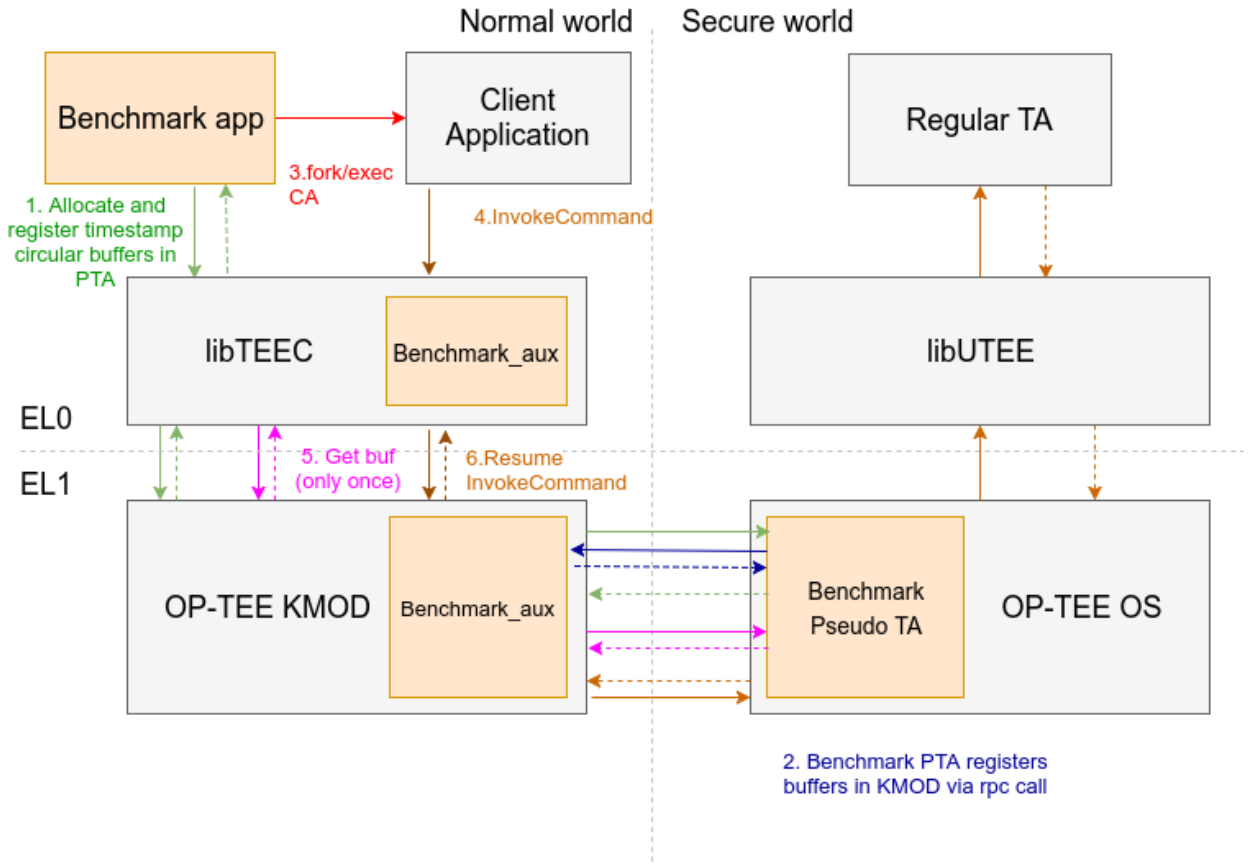
4.2.1 Implementation details

Design overview

Benchmark framework consists of such components:

1. **Benchmark Client Application (CA)**: a dedicated client application, which is responsible for allocating timestamp circular buffers, registering these buffers in the Benchmark PTA and consuming all timestamp data generated by all OP-TEE layers. Finally, it puts timestamp data into appropriate file with `.ts` extension. Additional build details can be found at *optee_benchmark*.
2. **Benchmark Pseudo Trusted Application (PTA)**: which owns all per-cpu circular non-secure buffers from a shared memory. Benchmark PTA must be invoked (by a CA) to register the timestamp circular buffers. In turn, the Benchmark PTA invokes the OP-TEE Linux driver (through some RPC mean) to register this circular buffers in the Linux kernel layer.
3. **libTEEC** and **Linux kernel OP-TEE driver** include functionality for handling timestamp buffer registration requests from the Benchmark PTA.

When the benchmark is enabled, all OP-TEE layers (libTEEC, Linux kernel OP-TEE driver, OP-TEE OS core) do fill the registered timestamp circular buffer with timestamp data for all invocation requests on condition that the circular buffer is allocated/registered.



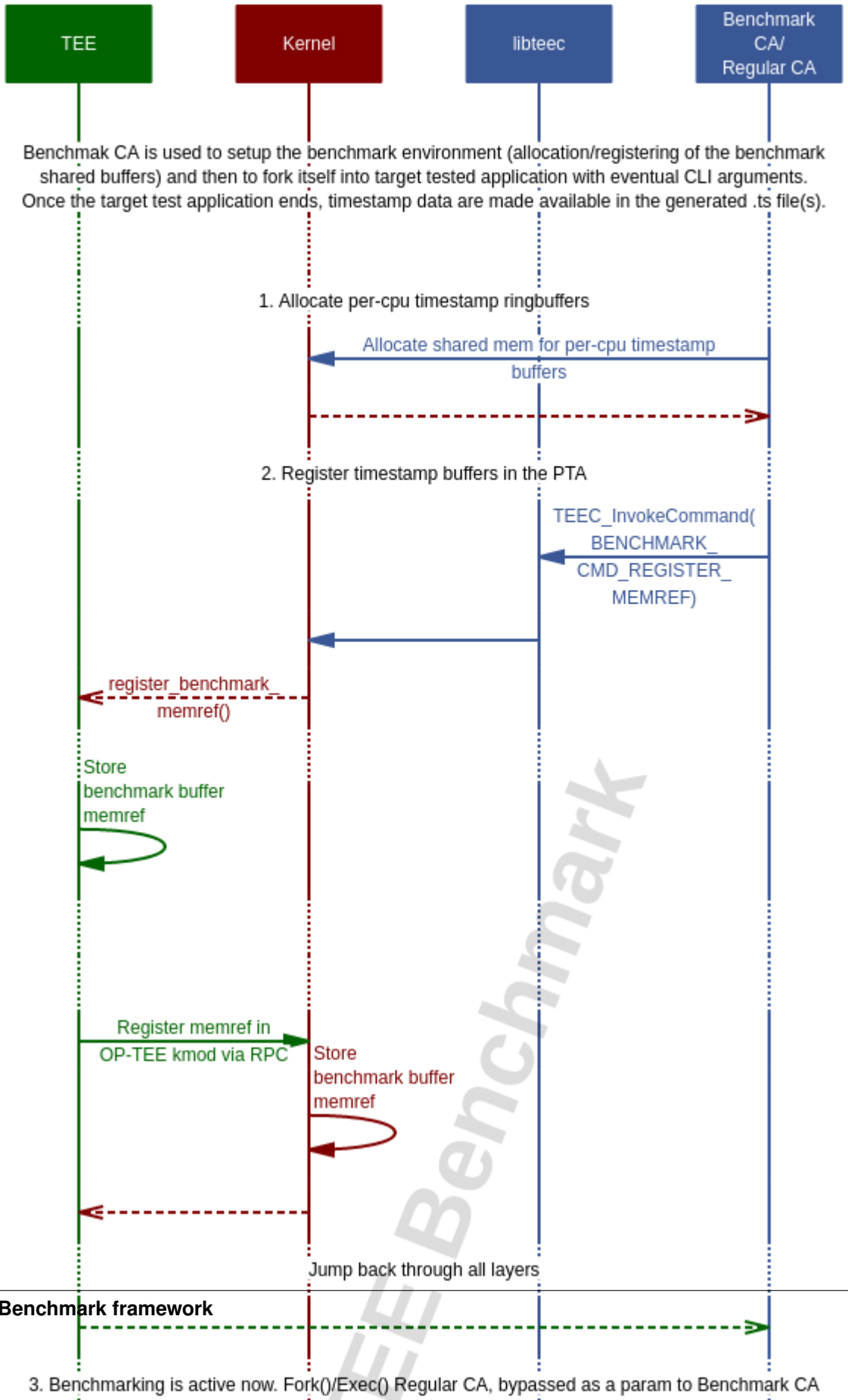
Timestamp source

Arm Performance Monitor Units are used as the main source of timestamp values. The reason why this technology was chosen is that it is supported on all Armv7-A/Armv8-A cores. Besides it can provide precise pre-cpu cycle counter values, it is possible to enable EL0 access to all events, so usermode applications can directly read cpu counter values from coprocessor registers, achieving minimal latency by avoiding additional syscalls to EL1 core.

Besides CPU cycle counter values, timestamp by itself contains also information about:

- Executing CPU core index
- OP-TEE layer id, where this timestamp was obtained from
- Program counter value when timestamp was logged, which can be used for getting a symbol name (a filename and line number)

Call sequence diagram



4.2.2 Adding custom timestamps

Currently, timestamping is done only for `InvokeCommand` calls, but it's also possible to choose custom places in the supported OP-TEE layers. To add timestamp storing command to custom c source file:

1. Include appropriate header:
   - OP-TEE OS Core: `bench.h`
   - Linux kernel OP-TEE module: `optee_bench.h`
   - libTEEC: `teec_benchmark.h`
2. Invoke `bm_timestamp()` (for linux kmod use `optee_bm_timestamp()`) in the function, where you want to put timestamp from.

4.2.3 Build and run benchmark

Please see the instructions available at *optee_benchmark*.

4.2.4 Limitations and further steps

- Implementation of application which will analyze timestamp data and provide statistics for different types of calls providing avg/min/max values (both CPU cycles and time values).
- Add support for all platforms, where OP-TEE is supported.
- Adding support of S-EL0 timestamping.
- Attaching additional payload information to each timestamp, for example, session.
- Timestamping within interrupt context in the OP-TEE OS Core.

4.3 Ftrace

This section describes how to generate function call graph for user Trusted Applications using `ftrace`.

The configuration option `CFG_FTRACE_SUPPORT=y` enables OP-TEE to collect function graph information from Trusted Applications running in user mode and compiled with `-pg`. Once collected, the function graph data is formatted in the `ftrace.out` format and sent to `tee-supplicant` via RPC, so they can be saved to disk, later processed and displayed using helper script called `symbolize.py` present as part of `optee_os` repo.

Another configuration option `CFG_SYSCALL_FTRACE=y` in addition to `CFG_FTRACE_SUPPORT=y` enables OP-TEE to collect function graph information for syscalls as well while running in kernel mode on behalf of Trusted Applications.

4.3.1 Usage

- Build OP-TEE OS and OP-TEE Client with `CFG_FTRACE_SUPPORT=y`. You may also set `CFG_ULIBS_MCOUNT=y` in OP-TEE OS to instrument the user TA libraries contained in `optee_os` (such as `libutee` and `libutils`).
- Optionally build OP-TEE OS with `CFG_SYSCALL_FTRACE=y` to dump additional syscall function graph information.

- Build user TAs with `-pg`, for instance enable `CFG_TA_MCOUNT=y` to instrument whole TA. Also, in case user wants to set `-pg` for a particular file, following should go in corresponding sub.mk: `cflags-<file-name>-y+=-pg`. Note that instrumented TAs have a larger `.bss` section. The memory overhead depends on `CFG_FTRACE_BUF_SIZE` macro which can be configured specific to user TAs using config: `CFG_FTRACE_BUF_SIZE=4096` (default value: 2048, refer to the TA linker script for details: `ta/arch/arm/ta.ld.S`).
- Run the application normally. When the current session exits or there is any abort during TA execution, `tee-supplicant` will write function graph data to `/tmp/ftrace-<ta_uuid>.out`. If the file already exists, a number is appended, such as: `ftrace-<ta_uuid>.1.out`.
- Run helper script called `symbolize.py` to translate the function graph addresses into function names: `cat ftrace-<ta_uuid>.out | ./optee_os/scripts/symbolize.py -d <ta_uuid>.elf -d tee.elf`

4.3.2 Typical output

```
           | __ta_entry() {
           |  __utee_entry() {
  1.664 us |   ta_header_get_session();
 11.264 us |   from_utee_params();
   .896 us |   memcpy();
           |   TA_InvokeCommandEntryPoint() {
           |    TEE_GenerateRandom() {
163.360 us |     utee_cryp_random_number_generate();
186.848 us |    }
214.288 us |   }
 19.088 us |   to_utee_params();
           |   ta_header_save_params() {
   .736 us |    memset();
  2.832 us |   }
304.880 us |  }
307.168 us | }
```

The duration (function's time of execution) is displayed on the closing bracket line of a function or on the same line in case the function is the leaf one. In other words, duration is displayed whenever an instrumented function returns. It comprises the time spent executing the function and any of its callees. The Counter-timer Physical Count register (CNTPCT) and the Counter-timer Frequency register (CNTFRQ) are used to compute durations. Time spent servicing foreign interrupts is subtracted.

4.4 Gprof

This describes to do profiling of user Trusted Applications with `gprof`.

The configuration option `CFG_TA_GPROF_SUPPORT=y` enables OP-TEE to collect profiling information from Trusted Applications running in user mode and compiled with `-pg`. Once collected, the profiling data are formatted in the `gmon.out` format and sent to `tee-supplicant` via RPC, so they can be saved to disk and later processed and displayed by the standard `gprof` tool.

4.4.1 Usage

- Build OP-TEE OS with `CFG_TA_GPROF_SUPPORT=y`. You may also set `CFG_ULIBS_MCOUNT=y` to instrument the user TA libraries contained in `optee_os` (such as `libutee` and `libutils`).

- Build user TAs with `-pg`, for instance enable: `CFG_TA_MCOUNT=y` to instrument whole user TA. Note that instrumented TAs have a larger `.bss` section. The memory overhead is 1.36 times the `.text` size for 32-bit TAs, and 1.77 times for 64-bit ones (refer to the TA linker script for details: `ta/arch/arm/ta.ld.S`).
- Run the application normally. When the last session exits, `tee-supplicant` will write profiling data to `/tmp/gmon-<ta_uuid>.out`. If the file already exists, a number is appended, such as: `gmon-<ta_uuid>.1.out`.
- Run gprof on the TA ELF file and profiling output: `gprof <ta_uuid>.elf gmon-<ta_uuid>.out`

4.4.2 Implementation

Part of the profiling is implemented in libutee. Another part is done in the TEE core by a pseudo-TA (`core/arch/arm/sta/gprof.c`). Two types of data are collected:

1. **Call graph information**
    - When TA source files are compiled with the -pg switch, the compiler generates extra code into each function prologue to call the instrumentation entry point (`__gnu_mcount_nc` or `_mcount` depending on the architecture). Each time an instrumented function is called, libutee records a pair of program counters (one is the caller and the other one is the callee) as well as the number of times this specific arc of the call graph has been invoked.
2. **PC distribution over time**
    - When an instrumented TA starts, libutee calls the pseudo-TA to start PC sampling for the current session. Sampling data are written into the user-space buffer directly by the TEE core.
    - Whenever the TA execution is interrupted, the TEE core records the current program counter value and builds a histogram of program locations (i.e., relative amount of time spent for each value of the PC). This is later used by the gprof tool to derive the time spent in each function. The sampling rate, which is assumed to be roughly constant, is computed by keeping track of the time spent executing user TA code and dividing the number of interrupts by the total time.
    - The profiling buffer into which call graph and sampling data are recorded is allocated in the TA's `.bss` section. Some space is reserved by the linker script, only when the TA is instrumented.

CHAPTER 5

Frequently Asked Questions

Table of Contents

- *Frequently Asked Questions*
  - *Abbreviations*
  - *Architecture*
    - *Q: Which platforms/architectures are supported?*
    - *Q: Are 32-bit as well as 64-bit support?*
    - *Q: Does OP-TEE support mixed-mode, i.e., both AArch32 and AArch64 Trusted Applications on top of an AArch64 core?*
    - *Q: What's the maximum size for heap and stack? Can it be changed?*
    - *Q: What is the size of OP-TEE itself?*
    - *Q: Can NEON optimizations be done in OP-TEE?*
    - *Q: Can I use C++ libraries in OP-TEE?*
    - *Q: Would using* malloc() *in OP-TEE give physically contiguous memory?*
    - *Q: Can I limit what CPUs / cores OP-TEE runs on?*
    - *Q: How is OP-TEE being scheduled?*
  - *Board support*
    - *Q: How do I port OP-TEE to another platform?*
  - *Building*
    - *Q: I got build errors running latest, why?*
    - *Q: I got build errors running stable tag x.y.z, why?*

    - *Q: I get* gcc XYZ *or* g++ XYZ *compiler error messages?*
    - *Q: I found this build.git, what is that?*
    - *Q: When running* make *from build.git it fails to download the toolchains?*
    - *Q: What is the quickest and easiest way to try OP-TEE?*
  - *Certification and security reviews*
    - *Q: Will TrustedFirmware.org be involved in GlobalPlatform certification/qualification?*
    - *Q: Has any test lab been testing OP-TEE?*
    - *Q: Have there been any code audit / code review done?*
  - *Contribution*
    - *Q: How do I contribute?*
    - *Q: Where can I get help?*
    - *Q: I'm new to OP-TEE but I would like to help out, what can I do?*
  - *Interfaces*
    - *Q: Which API's have been implemented in OP-TEE?*
  - *Hardware and peripherals*
    - *Q: Can I use my own hardware IP for crypto acceleration?*
  - *License*
    - *Q: Under what license is OP-TEE released?*
    - *Q: GlobalPlatform click-through license*
    - *Q: I've modified OP-TEE by using code with non BSD 2-Clause license, will you accept it?*
  - *Promotion*
    - *Q: I want to get my company logo on op-tee.org, how?*
  - *Security vulnerabilities*
    - *Q: I have a found a security flaw in OP-TEE, how can I disclose it with you?*
  - *Source code*
    - *Q: Where is the source code?*
    - *Q: Where do I download the test suite called xtest?*
    - *Q: Where is the Linux kernel TEE driver?*
  - *Testing*
    - *Q: How are you testing OP-TEE?*
  - *Trusted Applications*
    - *Q: How do I write a Trusted Application (TA)?*
    - *Q: How do I link a library into a Trusted Application?*
    - *Q: Where should I put my compiled Trusted Application on the device?*
    - *Q: What is a Pseudo TA and how do I write one?*

* *Q: Are Pseudo **user space** TAs supported?*
* *Q: Can a static TA Open/Invoke dynamic TA?*
* *Q: How can I extend the GlobalPlatform Internal Core API?*
* *Q: How are Trusted Applications verified?*
* *Q: Is multi-core TA supported?*
* *Q: Is multi-threading supported in a TA?*
* *Q: How can I use or call OP-TEE from native Android (apk) applications?*
* *Q: I've heard that there is a Widevine and PlayReady TA, how do I get access?*

5.1 Abbreviations

OP-TEE Open Portable TEE

TA Trusted Application

TEE Trusted Execution Environment

TZASC TrustZone Address Space Controller

TZPC TrustZone Protection Controller

5.2 Architecture

5.2.1 Q: Which platforms/architectures are supported?

- The *Platforms supported* page lists all platforms and architectures currently supported in the official tree.

5.2.2 Q: Are 32-bit as well as 64-bit support?

- Both 32- and 64-bit are fully supported for all OP-TEE components.

5.2.3 Q: Does OP-TEE support mixed-mode, i.e., both AArch32 and AArch64 Trusted Applications on top of an AArch64 core?

- Yes!

5.2.4 Q: What's the maximum size for heap and stack? Can it be changed?

- Yes, it can be changed. In the current setup (for vexpress for example), there are `32MB DDR` dedicated for OP-TEE. `1MB` for `TEE RAM` and `1MB` for `PUB RAM`, this leaves `30MB` for Trusted Applications. In the Trusted Applications, you set `TA_STACK_SIZE` and `TA_DATA_SIZE`. Typically, we set stack to `2KB` and data to `32K`. But you are free to adjust those according to the amount of memory you have available. If you need them

to be bigger than `1MB` then you also must adjust TA's MMU L1 table accordingly, since default section mapping is 1MB.

5.2.5 Q: What is the size of OP-TEE itself?

- As of 2016.01, optee_os is about `244KB` (release build). It is preferred to run *optee_os* entierly in SRAM, but if there is not enough room, DRAM can be used and protected with TZASC. We are also looking into the possibility of creating a 'minimal' OP-TEE, i.e. a limited OP-TEE usable even in a very memory constrained environment, by eliminating as many memory-hungry parts as possible. There is however no ETA for this at the moment.
- You can check the memory usage by using the `make mem_usage` target in *optee_os*, for example:

```
$ make ... mem_usage
# Which will output a file with the figures here:
# out/arm/core/tee.mem_usage
```

You will of course get different sizes depending on what compile time flags you have enabled when running *make mem_usage*.

5.2.6 Q: Can NEON optimizations be done in OP-TEE?

- Yes (for additional information, please also see Issue#953)

5.2.7 Q: Can I use C++ libraries in OP-TEE?

- C++ libraries are currently not supported. Technically, it is possible but will require a fair amount of work to implement, especially more so if exceptions are required. There are currently no plans to do this.
- See Issue#2628 for related information.

5.2.8 Q: Would using *malloc()* in OP-TEE give physically contiguous memory?

- `malloc()` in OP-TEE currently gives physically contiguous memory. It is not guaranteed as it is not mentioned anywhere in the documentation, but in practice the heap only has physically contiguous memory in the pool(s). The heap in OP-TEE is normally quite small, ~24KiB, and could be a bit fragmented.

5.2.9 Q: Can I limit what CPUs / cores OP-TEE runs on?

- Currently it's up to the kernel to decide which core it runs on, i.e, it will be the same core as the one initiating the SMC in Linux. Please also see Issue#1194.

5.2.10 Q: How is OP-TEE being scheduled?

- OP-TEE does not have its own scheduler, instead it is being scheduled by Linux kernel. For more information, please see *Issue#1036* and Issue#1183.

5.3 Board support

5.3.1 Q: How do I port OP-TEE to another platform?

- Start by reading the *Porting guidelines*.
- See the *Presentations* page. There might be some interesting information in the "LCU14-302 How To Port OP-TEE To Another Platform" deck and video. Beware that the presentation is more than five years old, so even though it is a good source, there might be parts that are not relevant any longer.
- As a good example for
  - **Armv8-A** patch enabling OP-TEE support on a new device, please see the ZynqMP port that enabled support for running OP-TEE on *Xilinx UltraScale+ Zynq MPSoC*. Besides that there are similar patches for Juno port, Raspberry Pi3 port, HiKey port.
  - **ARMv7-A**, please have a look at the Freescale ls1021a port, another example would be the TI DRA7xx port.

5.4 Building

5.4.1 Q: I got build errors running latest, why?

- What did you try to build? Only *optee_os*? A full OP-TEE developer setup using QEMU, HiKey, RPi3, Juno using repo? AOSP? OpenEmbedded? What we build on daily basis are the OP-TEE developer setups (see *Platforms supported by build.git*) , but other builds like AOSP and OpenEmbedded are builds that we try from time to time, but we have no CI/regression testing configured for those builds.
- By running latest instead of stable also comes with a risk of getting build errors due to version and/or interdependency skew which can result in build error. Now, such issues most often affects running xtest and not the building. If you however clean all gits and do a `repo sync -d`. Then we're almost 100% sure you will get back to a working state again, since as mentioned in next bullet, we build (and run xtest) on all QEMU on all patches sent to OP-TEE.
- Every pull request in OP-TEE are tested on hardware (see *Q: How are you testing OP-TEE?*).

5.4.2 Q: I got build errors running stable tag x.y.z, why?

- Stable releases are quite well tested both in terms of building for all supported platforms and running xtest on all platforms, so if you cannot get that to build and run, then there is a great chance you have something wrong on your side. All platforms that has been tested on a stable release can be found in CHANGELOG.md file. Having that said, we do make mistakes on stable builds also from time to time.

5.4.3 Q: I get *gcc XYZ* or *g++ XYZ* compiler error messages?

- Most likely you're trying to build OP-TEE using the regular x86 compiler and not the using the Arm toolchain. Please install the *Prerequisites* and make sure you have gotten and installed the Arm toolchains as described at the *Toolchains* page. (for additional information, please see Issue#846).

5.4.4 Q: I found this build.git, what is that?

- *build* is a git that is used in conjunction with the *manifest* to create full OP-TEE developer builds. It contains helper makefiles that makes it easy to get OP-TEE up and running on the setups that are using repo.

5.4.5 Q: When running *make* from build.git it fails to download the toolchains?

- We try to stay somewhat up to date with running recent GCC versions. But just like everywhere else on the net things moves around. In some cases like Issue#1195, the URL was changed without us noticing it. If you find and fix such an issue, please send the fix as pull request and we will be happy to merge it.

5.4.6 Q: What is the quickest and easiest way to try OP-TEE?

- That would be running it on QEMU on a local PC. To do that you would need to:
  - Install the OP-TEE *Prerequisites*.
  - Build for QEMU according to the instructions at *QEMU v7*.
  - And *Run xtest*.
- **By summarizing the above, you would need to:**

```
$ sudo apt-get install [pre-reqs]
$ mkdir optee-qemu && cd optee-qemu
$ repo init -u https://github.com/OP-TEE/manifest.git
$ repo sync
$ cd build
$ make toolchains -j2
$ make run
QEMU console:          (qemu) c
Normal world shell:    # xtest
```

5.5 Certification and security reviews

5.5.1 Q: Will TrustedFirmware.org be involved in GlobalPlatform certification/qualification?

- No, not as of now. Most often certification is performed using a certain software version and on a unique device. I.e., it is the combination software + hardware that gets certified. This is typically something that the SoC or OEM needs to do on their own.
- But it is worth mentioning that since OP-TEE is coming from a proprietary TEE solution that was GlobalPlatform certified on some products in the past and we regularly have people from some member companies running the extended test suite from GlobalPlatform we know that the gap to become GlobalPlatform certified/qualified isn't that big.

5.5.2 Q: Has any test lab been testing OP-TEE?

- Applus Laboratories have done some side-channel attack testing and fault injection testing on OP-TEE using the *HiKey 620* device. Their findings and fixes can be found at the Security Advisories page at optee.org.
- Riscure did a mini-audit of OP-TEE which generated a couple of patches (see *PR#2745*). The Security Advisories page at optee.org will be updated with more information regarding that in the future.

5.5.3 Q: Have there been any code audit / code review done?

- Full audit? No! But in the past Linaro have been collaborating with Riscure trying to identify and fix potential security issues. There has also been some companies that have done audits internally and they have then shared the result with us and where relevant, we have created patches resolving the issues reported to us (see *Q: Has any test lab been testing OP-TEE?*).
- Code review, yes! Every single patch going into OP-TEE has been reviewed in a pull request on GitHub. We more or less have a requirement that every patch going into OP-TEE shall at least have one "Reviewed-by" tag in the patch.
- Third party / test lab code review, no! Again some companies have reviewed internally and shared the result with us, but other than that no (see related *Q: Has any test lab been testing OP-TEE?*)

5.6 Contribution

5.6.1 Q: How do I contribute?

- Please see the *Contribute* page.

5.6.2 Q: Where can I get help?

- Please see the *Contact* page.

5.6.3 Q: I'm new to OP-TEE but I would like to help out, what can I do?

- We always need help with code reviews, feel free to review any of the open OP-TEE OS Pull Requests. Please also note that there could be open pull request in the other *OP-TEE gits* that needs reviews too.
- We always need help answering all the questions asked at OP-TEE OS Issues.
- If you want to try to solve a bug, please have a look at the OP-TEE OS Bugs or the OP-TEE OS Enhancements.
- Documentation tends to become obsolete if not maintained on regular basis. We try to do our best, but we're not perfect. Please have a look at *optee_docs* and try to update where you find gaps.
- Enable *repo* for the device in *manifest* and *build* (and also *Platforms supported*) currently not using repo.
- If you would like to implement a bigger feature, please reach out to us (see *Contact*) and we can discuss what is most relevant to look into for the moment. If you already have an idea, feel free to send the proposal to us.

5.7 Interfaces

5.7.1 Q: Which API's have been implemented in OP-TEE?

- **GlobalPlatform (see *GlobalPlatform API* for more details).**
  - GlobalPlatform's TEE Client API v1.1 specification
  - GlobalPlatform's TEE Internal Core API v1.1 specification.
  - GlobalPlatform's Secure Elements v1.0 (**now deprecated**, see `git log`).
  - GlobalPlatform's Socket API v1.0 (TCP and UDP, but not TLS).
- AOSP Keymaster (v3) and AOSP Gatekeeper (see *AOSP* for more details).
- Android Verified Boot 2.0 (AVB 2.0)

5.8 Hardware and peripherals

5.8.1 Q: Can I use my own hardware IP for crypto acceleration?

- Yes, OP-TEE has a Crypto Abstraction Layer (see *Cryptographic implementation* that was designed mainly to make it easy to add support for hardware crypto acceleration. There you will find information about the abstraction layer itself and what you need to do to be able to support new software/hardware "drivers" in OP-TEE.

5.9 License

5.9.1 Q: Under what license is OP-TEE released?

- The software is mostly provided under the BSD 2-Clause license.
- The TEE kernel driver is released under GPLv2 for obvious reasons.
- xtest (*optee_test*) uses BSD 2-Clause for code running in secure world (Trusted Applications etc) and GPLv2 for code running in normal world (client code).

5.9.2 Q: GlobalPlatform click-through license

- Since OP-TEE is a GlobalPlatform based TEE which implements the APIs as specified by GlobalPlatform one has to accept, the click-through license which is presented when trying to download the *GlobalPlatform API* specifications before start using OP-TEE.

5.9.3 Q: I've modified OP-TEE by using code with non BSD 2-Clause license, will you accept it?

- That is something we deal with case by case. But as a general answer, if it does not contaminate the BSD 2-Clause license we will accept it. Reach out to us (see *Contact*) and we will take it from there.

5.10 Promotion

5.10.1 Q: I want to get my company logo on op-tee.org, how?

- If your company has done significant contributions to OP-TEE, then please *Contact* us and we will do our best to include your company. Pay attention to that we will review this on regular basis and inactive supporting companies might be removed in the future again.

5.11 Security vulnerabilities

5.11.1 Q: I have a found a security flaw in OP-TEE, how can I disclose it with you?

- Please see the *Contact* page.

5.12 Source code

5.12.1 Q: Where is the source code?

- It is located on GitHub under the project OP-TEE and linaro-swg.

5.12.2 Q: Where do I download the test suite called xtest?

- All the source code for that can be found in the git called *optee_test*.
- The *Extended test (GlobalPlatform tests)* can be purchased separately.

5.12.3 Q: Where is the Linux kernel TEE driver?

- You can find both the generic TEE framework including the OP-TEE driver included in the official Linux kernel project since v4.12. Having that said, we "buffer up" pending patches on a our *Linux kernel TEE framework* branch. I.e., that is where we keep new features being developed for OP-TEE. In the long run we aim to completely stop using our own branch and just send all patches to the official Linux kernel tree directly. But as of now we cannot do that.

5.13 Testing

5.13.1 Q: How are you testing OP-TEE?

- There is a test suite called xtest that tests the complete TEE-solution to ensure that the communication between all architectural layers is working as it should. The test suite also tests the majority of the GlobalPlatform TEE Internal Core API. It has close to 50,000 and ever increasing test cases, and is also extendable to include the official GlobalPlatform test suite (see *Extended test (GlobalPlatform tests)*).
- Every pull request in OP-TEE are built for a multitude of different platforms automatically using Travis, Shippable and IBART. Please have a look there to see whether it failed building on the platform you're using before submitting any issue about build errors.
- For more information see *optee_test*.

5.14 Trusted Applications

5.14.1 Q: How do I write a Trusted Application (TA)?

- Have a look at the *Trusted Applications* page as well as the *optee_examples* page. Those provides guidelines and examples on how to implement basic Trusted Applications.
- If you want to see more advanced uses cases of Trusted Applications, then we encourage that you have a look at the Trusted Applications *optee_test*.

5.14.2 Q: How do I link a library into a Trusted Application?

- See the example in *sub.mk directives*.
- Also see Issue#280, Issue#601, Issue#901, Issue#1003.

5.14.3 Q: Where should I put my compiled Trusted Application on the device?

- `/lib/optee_armtz`, that is the default location where tee-supplicant will look for Trusted Applications.

5.14.4 Q: What is a Pseudo TA and how do I write one?

- A Pseudo TA is an OP-TEE firmware service offered through the generic API used to invoke Trusted Applications. Pseudo TA interface and services all runs in TEE kernel / core context. I.e., it will have access to the same functions, memory and hardware etc as the TEE core itself. If we're talking ARMv8-A it is running in `S-EL1`.

5.14.5 Q: Are Pseudo user space TAs supported?

- No!

5.14.6 Q: Can a static TA Open/Invoke dynamic TA?

- Yes, for a longer discussion see Issue#967, Issue#1085, Issue#1132.

5.14.7 Q: How can I extend the GlobalPlatform Internal Core API?

- You may develop your own "Pseudo TA", which is part of the core (see *Q: What is a Pseudo TA and how do I write one?* for more information about the Pseudo TA).

5.14.8 Q: How are Trusted Applications verified?

- Please see the section *Trusted Application private/public keypair* in the *Porting guidelines*.
- Alternatively one can also build a Trusted Application and embed its raw binary content into the OP-TEE firmware binary. At runtime, if invoked, the Trusted Application will be loaded from the OP-TEE firmware image instead of being fetched from the normal world and authenticated in the secure world (see *Early TA* for more information).

5.14.9 Q: Is multi-core TA supported?

- Yes, you can have two or more TAs running simultaneously. Please see also Issue#1194.

5.14.10 Q: Is multi-threading supported in a TA?

- No, there is no such concept as `pthreads` or similar. I.e, you cannot spawn thread from a TA. If you need to run tasks in parallel, then you should probably look into running two TAs or more simultaneously and then let them communicate with each other using the `TA2TA` interface.

5.14.11 Q: How can I use or call OP-TEE from native Android (apk) applications?

- Use the Java Native Interface (JNI).
- First get familiar with sample_hellojni.html and make sure you can run the sample. After that, replace the C-side Implementation with for example *hello_world* or one of the other examples in *optee_examples*.

Note: Note that *hello_world* and other binaries in optee_examples are built as executables, and have to be modified to be built as a .so shared library instead so that it can be loaded by the Java-side Implementation.

- Note that `*.apk` apps by default have no access to the TEE driver. See Issue#903 for details. The workaround is to disable SELinux before launching any `*.apk` app that calls into OP-TEE. The solution is to create/write SELinux domains/rules to allow any required access, but since this is not a TEE-related issue, it is left as an exercise for the users.

5.14.12 Q: I've heard that there is a Widevine and PlayReady TA, how do I get access?

- TrustedFirmware have no such implementation, but Linaro do have reference implementations for that that they share with their members who have signed the WMLA and NDA/MLA with Google and Microsoft. So the advice is to reach out to Linaro if you have questions about that.