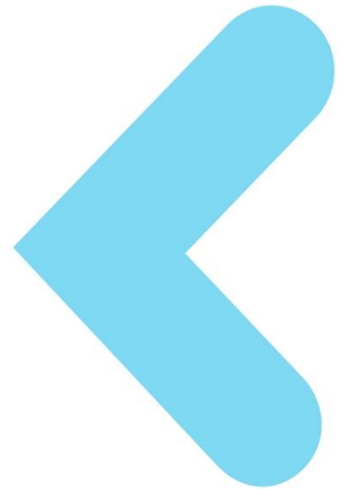
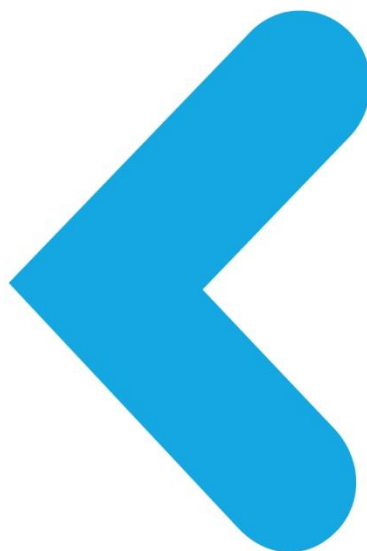


TRUSTONIC



Kinibi Developer's Guide



PREFACE

This specification is the confidential and proprietary information of Trustonic ("Confidential Information"). This specification is protected by copyright and the information described therein may be protected by one or more EC patents, foreign patents, or pending applications. No part of the Specification may be reproduced or divulged in any form by any means without the prior written authorization of Trustonic. Any use of the Specification and the information described is forbidden (including, but not limited to, implementation, whether partial or total, modification, and any form of testing or derivative work) unless written authorization or appropriate license rights are previously granted by Trustonic.

TRUSTONIC MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF SOFTWARE DEVELOPED FROM THIS SPECIFICATION, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. TRUSTONIC SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SPECIFICATION OR ITS DERIVATIVES.

VERSION HISTORY

Version	Date	Modification
1.0	July 18th, 2013	First version
2.0	November 18th, 2013	Updated for Kinibi-300
2.1	January 13th, 2014	Clarifications added
2.2	April 4th, 2014	SPPA for OTA provisioning added
2.3	May 13th, 2014	Clarifications added for API availability
2.4	June 9th, 2014	Updated for Kinibi-301
2.5	August 25th, 2014	Updated for Kinibi-301B <ul style="list-style-type: none"> - Update MobiConvert Manual - Explain usage of UUID attestation for GP TAs. - Clarify stack and heap constraints
2.6	November 18th, 2014	Updated for Kinibi-302A <ul style="list-style-type: none"> - Introduced extended memory layout. - Init Entry Point for TAs. - Floating Point support

		- Increased TEE virtual space to 128MB.
2.7	January 9 th , 2015	Updated for SDK-r7
2.8	August 12 th , 2015	Updated for Kinibi-310A - More APIs for Trusted Storage and GP - Increase number of supported crypto algorithm
2.9	October 29 th , 2015	Updated for Kinibi-310B - Introduce stack protection
3.0	November 19 th , 2015	Updated for Linux integration.
3.1	February 12, 2016	Updated for usage of new API TEEC_TT_RegisterPlatformContext()
4.0	September 22 nd , 2016	Updated for Kinibi-400A
4.1	March 14 th , 2017	Update the path of the mcRegistry
4.2	May 5 th , 2017	Updated for Kinibi-410a - Update new limitation of world-shared memory when using mcMap - Recommend protecting against TOCTOU issues - Recommend protecting against Stack corruption - Recommend protecting against TA rollback
4.3	July 10 th , 2017	Updated to include TUI event loop
4.4	November 13 th , 2017	Re-introducing documentation for KPH and HSM
4.5	July 3 th , 2018	Update for Kinibi-410a - Support for GCC 7.1.1 - ASLR – building TAs with TA_PIE - System TA encryption
5.0	July 31, 2018	MobiConvert changes for Kinibi-410a
5.1	September 12, 2018	Update for Kinibi-410a Commercial Release - Support for CLANG 6.0.0 - Debugging in the face of ASLR
5.2	February 28, 2019	Update for Kinibi-500a EA: - Remove ARM RVCT support - AArch64 for TA support added (ARM_V8A_AARCH64)

5.3	July 15, 2019	Update for Kinibi-500a Commercial Release: <ul style="list-style-type: none">- GCC and Clang version refreshed (8.3 and 8.0)- Documentation for legacy OTAv1/SP removed
-----	---------------	--

TABLE OF CONTENTS

1	Introduction	9
1.1	Glossary and Abbreviations.....	9
2	Kinibi Product Overview	12
2.1	Kinibi API for Developers	14
3	Development Guidelines	15
3.1	General Guidelines	15
3.1.1	Defining the scope of the Trusted Application	15
3.1.2	Trusted Application Design	15
3.1.3	Protecting TA code against reverse engineering and exploits.....	16
3.1.4	Trusted Application Address Space.....	17
3.1.4.1	World-shared memory.....	17
3.1.4.2	TOCTOU issues with shared memory.....	17
3.1.4.3	Stack	17
3.1.4.4	Heap	18
3.1.4.5	Stack corruption	18
3.1.4.6	ASLR.....	21
3.1.4.7	Position Independent Executable – PIE.....	21
3.1.5	Floating point support.....	22
3.1.6	TA Rollback Protection	22
3.1.7	System TA encryption	23
3.2	Using the GlobalPlatform API	23
3.2.1	Client Applications	23
3.2.2	Trusted Applications	24
3.2.2.1	TA Lifecycle.....	24
3.2.2.2	TA Manifest	25
3.2.2.3	TA Interface.....	25
3.2.2.4	TA Secure Storage	27
3.2.2.5	Implementing the TA Interface	27
3.2.2.6	Implementing the Operations.....	28
3.2.3	TA-to-TA communication	30
3.2.4	TEE Capabilities	30
3.2.5	Extended APIs.....	30
3.3	Using the Legacy API.....	31
3.3.1	Client Applications	31

3.3.1.1	Trusted Application Connector	31
3.3.1.2	Using the mcClient API	31
3.3.1.2.1	Device Sessions.....	31
3.3.1.2.2	Registering the platform specific context.....	32
3.3.1.2.3	Trusted Application Sessions.....	32
3.3.1.2.4	Message Exchange and Signaling	33
3.3.1.2.5	Kinibi version information	33
3.3.1.2.6	Memory Mapping.....	34
3.3.1.3	Client Kernel API.....	34
3.3.2	Trusted Applications	34
3.3.2.1	Trusted Application Structure	34
3.3.2.2	Advanced TCI Communication Protocol.....	35
3.3.2.3	Security Considerations.....	39
3.3.2.4	Kinibi tI API	40
3.3.2.5	Secure Objects	40
3.3.2.6	Endorsement API.....	43
3.3.2.7	Checking API return value	43
3.3.3	Trusted User Interface	43
3.3.3.1	TUI configuration.....	45
3.3.3.2	TUI Session	45
3.3.3.2.1	Normal world integration.....	45
3.3.3.2.2	Error code checking.....	45
3.3.3.2.3	Single user service	46
3.3.3.3	Secure display	46
3.3.3.4	Secure input	46
3.3.3.4.1	Asynchronous driver interface	47
3.3.3.4.2	Trusted Application state machine.....	47
3.3.3.4.3	Input filtering and button emulation.....	47
3.3.3.4.4	TUI Event loop	48
3.3.4	DRM API	49
4	Using the SDK.....	51
4.1	Toolchain Installation	51
4.1.1	System Requirements	51
4.1.2	Trusted Applications	51
4.1.2.1	GCC / Linaro Compiler	51
4.1.2.2	CLANG / LLVM Compiler	52

4.1.3	Client Applications	52
4.1.3.1	Android SDK / NDK.....	52
4.1.3.2	Using ADB.....	52
4.1.3.3	GCC / Linaro Compiler.....	53
4.2	Quick Start Guide.....	53
4.2.1	File structure	53
4.2.2	Setup instructions for Android.....	53
4.2.3	Setup instruction for Linux.....	54
4.2.4	Creating your first secure application	55
4.2.4.1	Trusted Application (TA)	55
4.2.4.2	Client Application (CA)	55
4.2.5	Running your first application.....	55
4.2.5.1	On a Development Board (running as System-TA or SP-TA).....	55
4.2.6	Creating secure application using legacy API.....	56
4.2.6.1	Trusted Application (TA)	57
4.2.6.2	Client Application (CA)	57
4.2.6.3	Android SDK	57
4.2.7	Running a secure application using legacy API	57
4.2.7.1	On a Commercial Device	57
4.2.7.2	On a Development Board (running as System-TA).....	58
4.3	Compiling and Testing Trusted Applications	58
4.3.1	Build Environment.....	59
4.3.2	Compiling and signing a Trusted Application.....	60
4.3.3	Running a Trusted Application in a Development Environment.....	61
4.3.4	Debugging a Trusted Application	61
4.3.4.1	The Debuggable Flag	62
4.3.4.2	Trusted Application start.....	62
4.3.4.3	Trusted Application crash	62
4.3.4.4	Logging	63
4.3.4.5	Debugging with ASLR	63
4.3.4.6	Development Platforms	64
4.4	Compiling and Testing Client Applications	65
4.4.1	Client Application Structure	65
4.4.2	Android Compilation	65
4.4.2.1	Compile a CA with the Android NDK	65

4.4.2.1.1	Android NDK Overview.....	65
4.4.2.2	Android APK	67
4.4.3	Linux Compilation.....	67
4.4.3.1	CMake overview (https://en.wikipedia.org/wiki/CMake)	67
4.4.4	Run your Client Application	69
4.4.4.1	Connect to the Device via USB.....	69
4.4.4.2	Connect to the Device via Ethernet	69
4.4.4.3	Upload and Test	70
4.4.4.4	Shared Libraries on the Device.....	71
4.4.4.5	Client Application usage from within your Android App	71
4.4.5	Debug	72
4.4.5.1	Segmentation Faults	72
4.4.5.2	GDB	73
4.4.5.2.1	Debug stand-alone binaries.....	73
4.5	Init entry point for Trusted Applications	75
5	MobiConvert Manual.....	76
5.1	Help output.....	76
5.2	RSA key generation.....	77
5.3	UUID generation	78
5.3.1	For GlobalPlatform Trusted Applications	78
5.3.2	For legacy services and drivers	78
5.4	Header mode	79
5.5	Conversion Mode	80
5.5.1	Help output	80
5.5.2	System TA conversion and signing	82
5.5.3	Driver conversion and signing	82
5.5.4	Conversion without signing.....	83
5.5.5	Conversion with tbin input file.....	83
5.6	Sign Modes for OEMs	84
5.6.1	Signing using an HSM-backed key store	84
5.6.2	Signing using a key file	85
5.6.3	Re-signing a signed tbin	85
5.6.4	Help output	85
5.7	Encrypt Mode for OEMs	87
5.8	Properties Editor Mode	89

LIST OF FIGURES

Figure 1: Kinibi Architecture Overview.....	12
Figure 2: Kinibi API Overview.	14
Figure 3: TA Virtual Address Space with extended layout, Kinibi-302	19
Figure 4: TA Virtual Address Space with legacy layout, Kinibi-300	19
Figure 5: TA Virtual Address Space with ASLR, Kinibi-410a.....	20
Figure 6: Trusted User Interface API.	44
Figure 7: Kinibi DRM API.....	50
Figure 8: Kinibi ecosystem overview	61
Figure 9 Illustrating the System TA signature structure	84

LIST OF TABLES

Table 1: TA Interface Functions	25
Table 2: Correspondence between CA operation and TA Interface.....	26
Table 3: TCI command- and response protocol structure.....	36
Table 4: SDK UUIDs for Development targets	61
Table 5: ADB commands.....	70
Table 6: Linux system debugging commands.....	72

1 INTRODUCTION

This Developer's Guide is a practical introduction for developing Trusted Applications for Kinibi and their corresponding Client Applications.

This guide provides an overview of the different sets of API available and explains how to use the SDK.

The full Kinibi API specification can be found in the Kinibi API Documentation.

1.1 GLOSSARY AND ABBREVIATIONS

ADB	Android Debug Bridge
API	Application Programming Interface
BSS	The BSS segment contains all uninitialized data.
CA	Client Application. Software running in the NWd providing a high level convenience interface to access a Trusted Application in the SWd by the NWd client.
DMA	Direct Memory Access
GDB	Gnu DeBugger
GNU	GNU's Not Unix
GP	GlobalPlatform
IPC	Inter-Process Communication, communication between two tasks running in different processes.
IWC	Inter-World Communication, communication between two tasks running in different worlds.
JNI	Java Native Interface
JTAG	Joint Test Action Group
JTAG DCC	JTAG Digital Command Control
Kinibi Ecosystem	The whole infrastructure enabling the deployment of Kinibi enabled devices as well as the installation and operation of security protected applications using the Kinibi Operating System.
MMU	Memory Mapping Unit
Multi ABI	Multiple Application Binary Interface

NDK	Android Native Development Kit
NWd	Normal World. The software system running on an ARM TrustZone enabled device in non-secure mode.
OS	Operating System
OTA	Over-The-Air
PC	Program Counter
PID	Process ID
RFC	Request For Comments published by the Internet Engineering Task Force (IETF)
RPC	Remote Procedure Call
SP-App	Service Provider Application: This is the implementation of the service function provided by the Service Provider (SP) to the end-user. The SP-app code is provided by the service provider. It includes the user interface and all the business logic which shall be implemented on the device side. For security sensitive operations SP-App requires functions implemented in a Trusted Application (TA).
SPPA	Service Provider Provisioning Agent: It is the library provided by the TAM, which provides the necessary normal world calls to provision service provider containers OTA.
SDK	Kinibi Software Development Kit
SWd	Secure World, software system running in secure mode in the ARM TrustZone
TA	Trusted Application on the SWd side executing specific security services in the Kinibi runtime environment. The security services provided by a Trusted Application are called by NWd client applications. Kinibi differentiates between System Trusted Application like the CM Trusted Application and Trusted Applications of Service Providers.
TAM	Trusted Application Manager: online server communicating with the RootPA and SPPA to provision containers in a safe and secured way.
TEE	Trusted Execution Environment. Kinibi is an instantiation of a TEE. The term TEE is defined in the OMTP specifications.
TZ	ARM TrustZone
UART	Universal Asynchronous Receiver/Transmitter

UUID	Universal Unique Identifier. The UUID is an identifier standard used in software construction, standardized by the Open Software Foundation (OSF) as part of the Distributed Computing Environment (DCE). The UUID in the Kinibi ecosystem is used to uniquely identify Trusted Applications. See RFC 4122.
WSM	World shared memory. Shared memory which will be used for IPC between NWd and SWd.

2 KINIBI PRODUCT OVERVIEW

Kinibi is a portable and open Trusted Execution Environment (TEE) aiming at executing Trusted Applications on a device. It includes built in cryptographic algorithms and secure objects for secure data persistence. It is a versatile environment that can be integrated on different System on Chip (SoC) supporting the ARM TrustZone technology.

Kinibi uses ARM TrustZone to separate the platform into two distinct areas, the Normal-World with a conventional rich operation system and rich applications and the Secure-World.

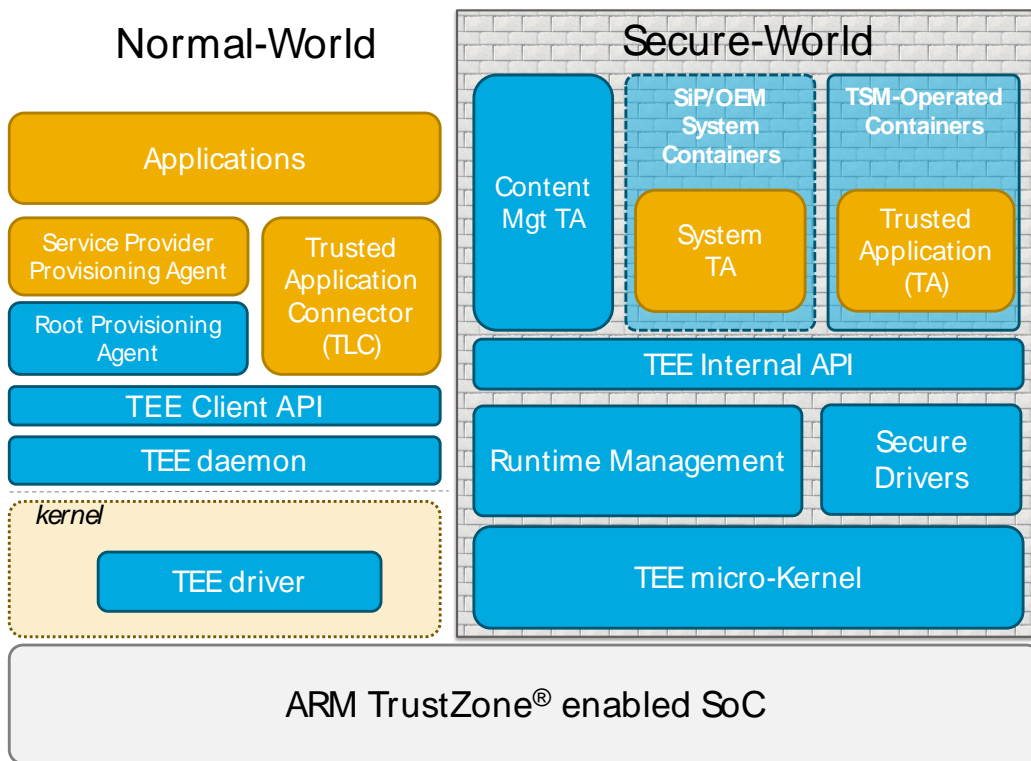


Figure 1: Kinibi Architecture Overview.

The Secure-World contains essentially the Kinibi core operating system and the Trusted Applications. It provides security functionality to the Normal-World with an on-device client-server architecture. The Normal-World contains mainly software which is not security sensitive (the sensitive code should be migrated to the Secure-World) and it calls the Secure-World to get security functionality via a communication mechanism and several APIs provided by Kinibi. The caller in the Normal-World is usually an application, also called a client.

The Trusted Applications in the Secure-World can be installed in 2 different security domains:

- ◀ The Service Provider Containers, which are created at runtime under the control of Trustonic. Trusted Applications of a Service Provider can be administrated Over-The-Air via a Trusted Application Manager (TAM). Current documentation and SDK are not covering technically this development and deployment model, it’s a different and dedicated Trustonic product (TAP).

- ◀ The System Container, which is pre-installed at the time of manufacture along with some Trusted Applications. Trusted Applications in the system container cannot be downloaded or updated Over-The-Air via a TAM.

Note that in order to enable Kinibi and the TAM-Operated Containers on a Device, the OEM must install Trustonic's Key Provisioning Host (KPH) at its manufacturing line. The KPH is a tool which injects a key on the device and which stores a copy in the Trustonic backend system.

Kinibi provides APIs in the Normal-World and the Secure-World. In the Normal-World, the Kinibi Client API is the API to communicate from the Normal-World to the Secure-World. This API establishes a communication channel with the Secure-World and sends commands to the Trusted Applications. It enables also to exchange some memory buffers between the Normal-World and the Trusted Applications.

In the Secure-World, Kinibi provides the Kinibi Internal API which is the interface to be used for the development of Trusted Applications. Trusted Applications use native execution on the ARM platform and can use the Floating Point and NEON instruction sets.

Furthermore, the Kinibi architecture supports Secure Drivers to interact with any secure peripherals.

2.1 KINIBI API FOR DEVELOPERS

Kinibi is an open environment which provides an API for developers to develop Trusted Applications and Client Applications. Kinibi provides two distinct set of APIs for developers:

- ◀ The GlobalPlatform API to develop Trusted Applications as defined by GlobalPlatform using the GlobalPlatform Client and Internal APIs.
- ◀ The Kinibi Legacy API to develop legacy Trusted Applications using the tIAPI and the mcClient API.

The DRM, TUI or GP APIs as well as support for NEON and Floating Point instruction set may not be available on certain commercial devices.

The implementation of these features may not be possible due to hardware limitations or other constraints.

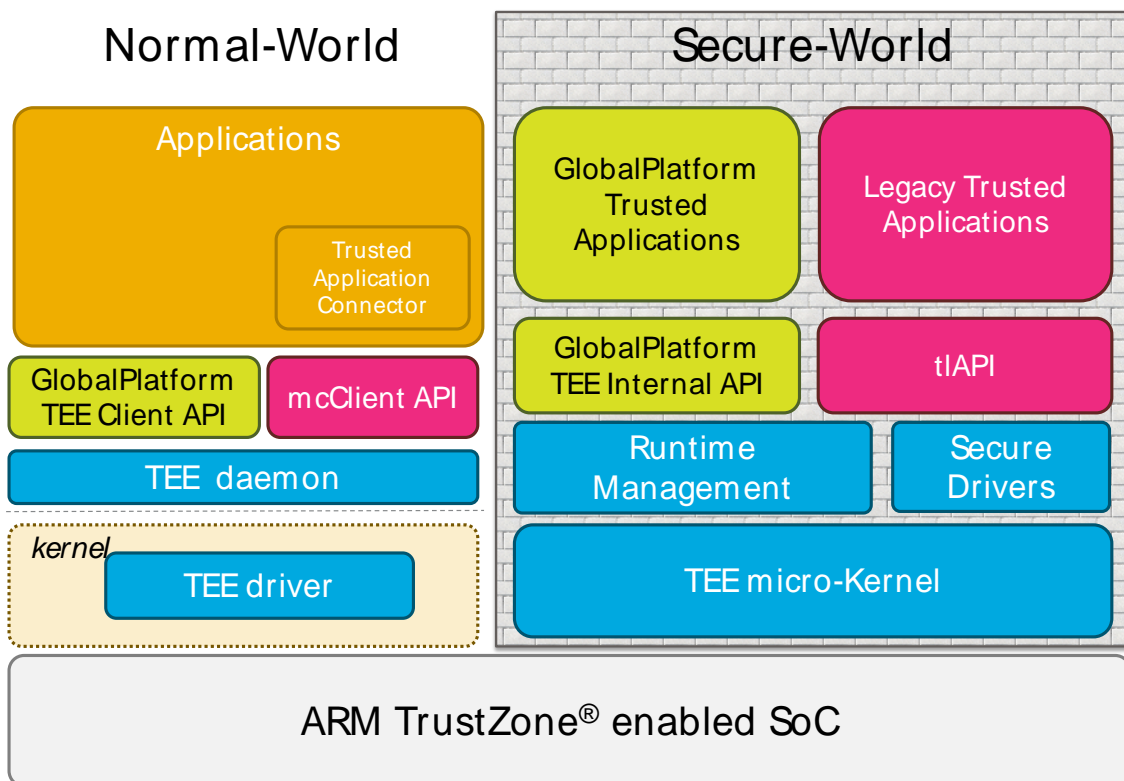


Figure 2: Kinibi API Overview.

3 DEVELOPMENT GUIDELINES

A rich operating system cannot provide the trustworthiness that is needed to keep up with the ongoing improvements of attacks and threats, coming with new functionalities, on its own. This means that security mechanisms need to be integrated in the system at the lowest possible level instead of being added on top. Kinibi is able to provide such a secure environment to protect security critical data and applications against attacks with the use of TrustZone.

The following chapter describes what is needed to fully take advantage of the functionality Kinibi provides for an application developer.

3.1 GENERAL GUIDELINES

3.1.1 Defining the scope of the Trusted Application

First, the developer needs to identify and define the security critical parts of an application.

The prerequisite for implementing a Trusted Application and Client Application is to determine the parts which should be incorporated within a Trusted Application. The aim should be to keep the Trusted Application part as small as possible. The main reason is that a big code base has potentially a lot of bugs; or, in other words, you should keep the system simple to have it working properly and securely.

In the following, a guideline is given for keeping the Trusted Application as small as possible:

1. Determine sensitive/critical information
2. Identify the parts that deal with this information
3. Isolate these parts in a Trusted Application

For example, in the payment Use Case, the security critical information would be the amount of money to be transferred and the participants of the transaction. Furthermore, the user must enter PIN and TAN to identify himself to get access to the banks service and sign a transaction. All the software components handling this sensitive information need to be identified. On the client side, this would mainly be the user interface, as it picks up the information from the user. Therefore, the PIN and TAN entry need to be controlled by a Trusted Application which can use a secure driver, making it impossible for the NWd to eavesdrop on the user secrets.

After the split into secure and non-secure part is decided, the application designer defines the state in the Trusted Application and the interface between the Trusted Application and the Client Application. The design must take into consideration the restrictions of the TA environment and the CA-TA interfaces.

3.1.2 Trusted Application Design

Developers should also note the following generic guidelines:

- ◀ Separate your Trusted Applications functionality into multiple operations, so the Trusted Application can return fast.
- ◀ Keep in mind that the Trusted Application execution may be interrupted by the Normal-World anytime (e.g. due to an incoming phone call).
- ◀ A Trusted Application process is limited to a single thread. In case you need to have multiple threads, you need to create separate Trusted Applications.

- ◀ GlobalPlatform Trusted Applications can use the TA-to-TA call API.
- ◀ Legacy Trusted Applications are not able to communicate directly with each other. But they can share Secure Objects to exchange data.
- ◀ Trusted Applications do not have direct hardware access, they can only use the Trusted Application API.
- ◀ Never write security critical data to the World-Shared Memory as it might be read by the Normal-World at any time.

3.1.3 Protecting TA code against reverse engineering and exploits

Trusted Applications can be a weak link in the security chain. Attackers can obtain the TA binary and perform reverse engineering to obtain TA code. That way attackers can identify weaknesses in the code and use them to extract secret keys and overcome the TEE protection guarantees. Kinibi contains a variety of features to protect TA code, such as TA encryption, TA Rollback Protection, as well as ASLR, Stack cookies, Guard pages and use of modern toolchains.

The security features depend on the Kinibi version and the integration of Kinibi on the device. As an OEM, all such features should be activated and must be integrated correctly. As a TA developer targeting a given device, the maximum number of security features should be activated when developing the TA. For TA developers targeting multiple devices, an assessment is required to find a compromise between security and compatibility.

This version of the SDK enables security by default:

- ◀ Latest TBASE_API_LEVEL is set by default
- ◀ Address layout randomization with TA_PIE by default set to PIE
- ◀ Stack canary activated by default
- ◀ TOOLCHAIN set to GCC
- ◀ Warnings as errors with -Wall -Wextra -Werror

Note that the NASA JPL Coding Standard for the C Programming Language also recommends additional compiler setting. Trustonic recommends the following additional flags:

- ◀ -Wshadow
- ◀ -Wpointer-arith
- ◀ -Wcast-qual
- ◀ -Wcast-align
- ◀ -Wstrict-prototypes
- ◀ -Wmissing-prototypes
- ◀ -Wredundant-decls
- ◀ -Wconversion
- ◀ -Wmissing-include-dirs
- ◀ -Wduplicated-branches
- ◀ -Wduplicated-cond
- ◀ -Wlogical-op
- ◀ -Wwrite-strings
- ◀ -Wnull-dereference

Also, Trustonic recommends use of static code analysis tools as well as sanitizers.

3.1.4 Trusted Application Address Space

Kinibi memory layouts supported for Trusted Applications, the extended layout (default), allows up to 120MB for program data.

Note that since Kinibi-500 the support of legacy memory layout (API Level lower than 5) has been fully removed thus loading TAs with an API level lower than 5 will fail.

3.1.4.1 World-shared memory

When the TA is being loaded, Kinibi tries to allocate all the required memory from secure memory. If the system does not have enough free memory, loading fails.

For sharing data with the Client Application, Kinibi provides an area in TA address space. The size of this area depends on the Kinibi version and the memory layout the TA is using.

- ◀ Starting from Kinibi 410A:
 - ◀ For extended memory layout, the area of World Shared Memory (WSM) in the TA address space is unlimited, only limit is the size of the TA virtual address space which is 124 MB.
- ◀ Prior Kinibi 410A:
 - ◀ For legacy memory layout and extended memory layout, Kinibi provides a 5 MB area of World Shared Memory (WSM) in the TA address space. The first WSM is mapped when the session with the NWd Client Application is opened. The address of this WSM and its length is given to the Trusted Application entry function (tlMain()). Then 4 additional slots of a maximum size of 1 MB can be mapped on-demand. Notice that 1 MB is a maximum size when the WSM is aligned to 4 KB.

3.1.4.2 TOCTOU issues with shared memory

The TA developer must avoid "Time of check to time of use" issues.



Normal world clients: Trusted Applications must be especially careful when handling data coming from the Client Application through World-Shared Memory buffers: the content of the buffers may change at any time, even between two consecutive memory accesses by the Trusted Application. This means that the Trusted Application should be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the Trusted Application should always read data only once from a World-Shared Memory buffer and then validate it. It must not assume that data written to the buffer can be read unchanged

later. This applies both for legacy TAs and GlobalPlatform TAs.

Another TA as a client: GlobalPlatform Trusted Applications can invoke each other and pass memory references as parameters. For the duration of the invoke command call, the TEE establishes shared memory between the client TA and the server TA. The client TA is blocked during the call and cannot itself modify the shared memory. However, a client TA can use a world-shared memory area, coming from its own invoke command input parameters, as invoke command parameters for the server TA

3.1.4.3 Stack

The developer must use the following macro to declare the required stack size:

```
DECLARE_TRUSTED_APPLICATION_MAIN_STACK(4096);
```

Since Kinibi-310B with API LEVEL 8, Kinibi protects the main stack using a MMU-guard page. When you select level 8, the binary only contains an integer with the minimum stack size. The startup code of the application will allocate a stack with one unmapped MMU page before and after the stack area. That way, stack overflows and underflows will not silently overwrite global variables and heap, but cause a segmentation fault that helps discover stack problems during the development phase.

With Kinibi-410a, the main stack will be allocated at a randomized address.

With API LEVEL < 8, the stack is part of the program data and is statically allocated during compile time. This is not recommended because it also allows attackers to overwrite global variables.

3.1.4.4 Heap

A Trusted Application can use common heap functionality using `tlApiMalloc`, `tlApiFree` and `tlApiRealloc` to organize its dynamic memory.

< Extended memory layout

The heap is allocated during the loading of the TA. The developer must set the following lines in the makefile

```
HEAP_SIZE_INIT := 4096 (for example)
HEAP_SIZE_MAX := 8192 (optional)
```

With Kinibi-410a, the heap will be allocated at a randomized address.

3.1.4.5 Stack corruption



The TA developer shall use the latest API LEVEL to take advantage of the best stack corruption protection. By using the extended memory layout, the heap is separated from the stack and global variables. The TA developer shall allocate big data structures and arrays on the heap to avoid likelihood of heap overflows corrupting the stack. When using also API LEVEL 8 and higher, the stack is further separated from the global variables using guard pages. That way, a corruption in the global variables cannot overwrite the stack. The TA developer should also use the latest version of the SDK for all TAs to leverage the recent addition of the Stack Smashing

Protection in the compiler settings.

For informative purpose, see Figure 3 for the different layouts of the virtual address space of a TA:

Trusted Application Address space 124MB

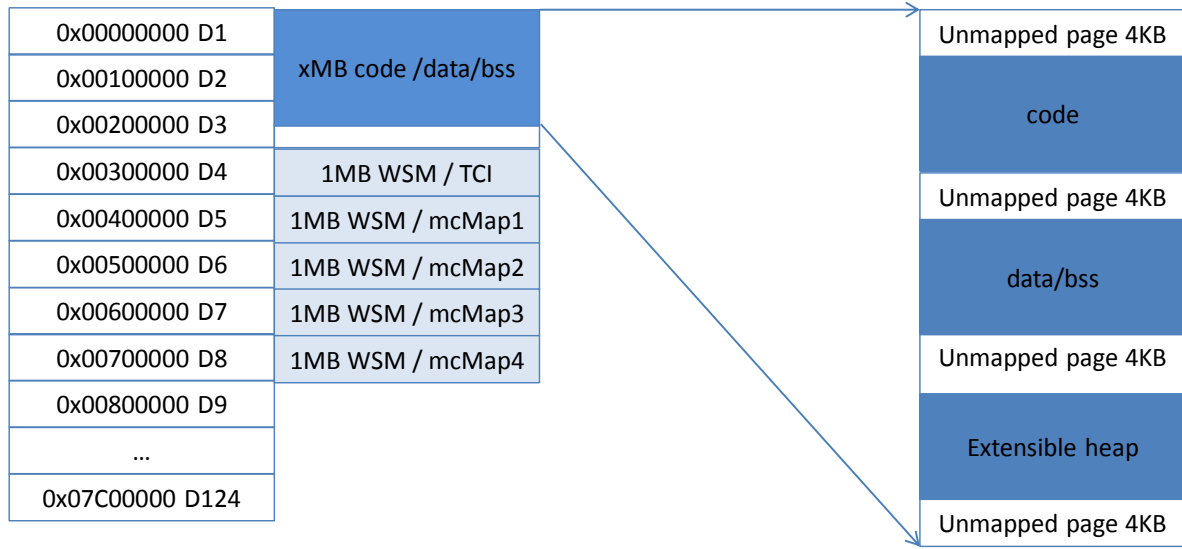


Figure 3: TA Virtual Address Space with extended layout, Kinibi-302

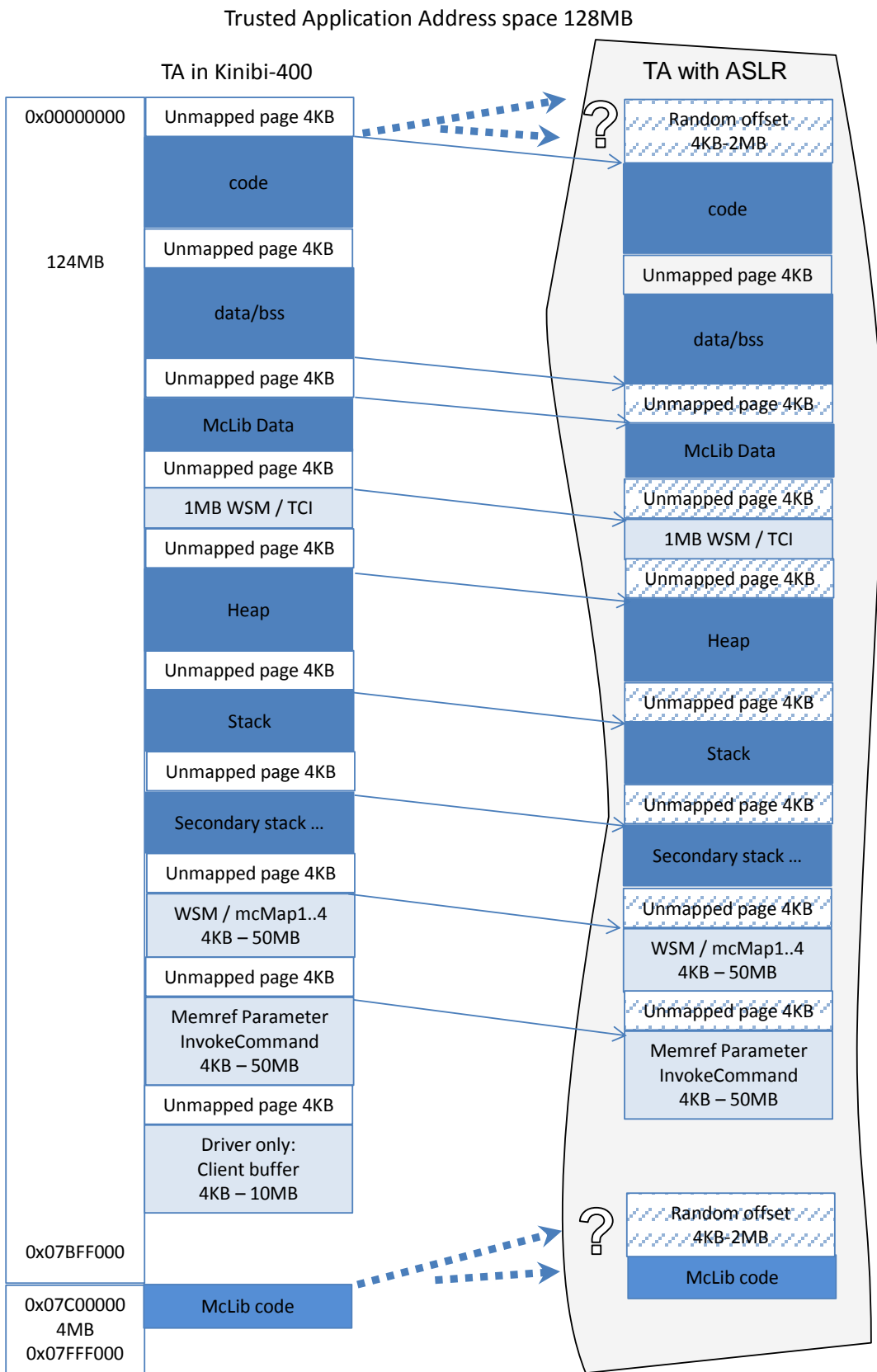


Figure 4: TA Virtual Address Space with ASLR, Kinibi-410a

3.1.4.6 ASLR

Kinibi supports Address Space Layout Randomization for McLib, TA code, and dynamically allocated sections. The TA address space is randomized on each loading of the TA. The Kinibi library McLib is mapped into each TA address space and has a random address per TA. The on-demand memory allocations like stack, heap, TCI, WSM, Memref Parameters of GP Invoke and Driver client mapping are allocated to positions relative to TA code and last mapping. If the TA code is compiled with PIE option, the TEE will also load the TA code to a random position on load. See Figure 4: TA Virtual Address Space with ASLR, Kinibi- for the layout of the virtual address space of a TA with ASLR.

The task ASLR offset and mclib ASLR offset are printed for debuggable TAs on crash, see TA debug at 4.3.4.

3.1.4.7 Position Independent Executable – PIE

To use ASLR for the TA code, the SDK compiles TAs and Drivers as Position Independent Executables using the `-fPIE` option.

Requirements:

- ◀ Deployment on Kinibi-410a
- ◀ recent TBASE_API_LEVEL (5 or higher, or not set)
- ◀ built with GCC or CLANG toolchain
- ◀ TA_PIE variable is activated

A TA developer can change the API level in the TA makefile.mk as:

```
TBASE_API_LEVEL := 5
```

However, Trustonic recommends to not define an API level, such that the latest level is selected by default.

A TA developer can change the toolchain either in setup.sh or in the TA makefile.mk as:

```
TOOLCHAIN := GNU or CLANG
```

PIE is activated by default in the SDK in setup.sh, using `TA_PIE=PIE`. The SDK sets `TA_PIE` to `NON_PIE` when `TBASE_API_LEVEL < 5`. The TA developer can deactivate ASLR in the TA makefile.mk by overloading `TA_PIE` like that:

```
TA_PIE := NON_PIE
```

Obviously, Trustonic does not endorse running TAs without PIE, since it makes the TA less secured.

Compatibility

TAs compiled with `-fPIE` have additional sections in the TA format. On Kinibi-410a interprets these sections to relocate the TA. On TEE version prior to Kinibi 410a, TAs will be loaded without relocation, the additional sections will be ignored.

A TA that uses static libraries must use versions of these libraries compiled with `-fPIE` option. Note that the SDK supports creating TAs with PIE and without PIE. Consequently, the SDK comes with two versions of the runtime libraries, one for PIE and one for `NON_PIE` in respective sub-folders.

3.1.5 Floating point support

Kinibi supports the hardware floating point and NEON instruction set usage in a Trusted Application.

The SDK supports using the float and double data types as well as the functions from math.h in Trusted Application code. By default, the SDK will instruct the compiler to generate software floating point code. Such code will run on all versions of Kinibi.

To use hardware floating point in the TA, the following lines must be added to the TA makefile:

```
HW_FLOATING_POINT := Y
```

The SDK will then instruct the compiler to generate hardware floating point code.

The same TA binary runs on ARMv7 and ARMv8 platforms. For ARMv8 platforms, Kinibi is running in AARCH32 mode, so 32 floating point registers of 64-bit are available (D0-D31), and 16 floating point registers of 128-bit (Q0-Q15).

Using floating point instructions can considerably speed up the execution time of relevant algorithms. However, only the platforms supporting the Kinibi API Level 5 or higher can benefit from hardware floating point acceleration.

Starting from Kinibi-400, the TA developer can use "com.trustonic.tee.isa.arm.neon" property to find out if NEON is supported on the current device. Use `TEE_GetPropertyAsBool()` function to retrieve its value.

3.1.6 TA Rollback Protection

Kinibi supports TA versioning to protect against rollback of a Trusted Application to an older, vulnerable version. To activate the rollback protection for his TA, the TA developer must inject a version into the TA binary using MobiConvert. The service provider must establish a process of deploying new versions of the TA whenever he learns about security vulnerability in his TA. The new version of the TA must contain fixes for the security vulnerability and have a higher version number.

Marking TAs for downgrade protection

Set a minimum version

With MobiConvert:

```
-iv 0.1, --interface-version <major.minor> : activate downgrade
                                             protection for the TA
```

In TA makefile:

```
TA_VERSION:=0.1
```

Enable automatic enrolment (for System TAs)

MobiConvert and TA makefiles accept new options:

New MobiConvert option:

```
-dp, --downgrade-protection: activate automatic enrolment for System TA
```

New TA makefile option:

```
TA_ROLLBACK_PROTECTED:=Y
```


- ◀ Opening and closing sessions to one or more Trusted Applications
- ◀ Registering and unregistering shared memory
- ◀ Invoking commands on sessions, passing command data and memory buffers to a Trusted Application
- ◀ Handling Trusted Application return codes and data returned either directly or in memory.

Operations

When a CA sends a command to a TA, the command contains a command identifier and an operation payload in accordance with the protocol that the TA exposes for that command. An operation contains 4 parameters that can be used to exchange data from the CA to the TA and from the TA to the CA. Parameters are typed and can either denote an immediate value or a reference to a memory region. The answer to a command invocation always contains a return code. Return codes are typed and give an indication as to where in the software stack a possible error occurred.

Cancellation

Calls to the TEE Client API are synchronous, meaning that a long-running operation in the TA blocks the CA. When a CA is reactive also to user input, network timeouts and other events, the CA should use multiple threads. If a thread of a CA is blocked by a TA, another thread of the CA can call the cancel function on this TA session, effectively stopping the TA session and unblocking the other CA thread.

This version of Kinibi supports the cancellation of blocking API functions (TEEC_OpenSession and TEE_InvokeCommand).

3.2.2 Trusted Applications

When developing a Trusted Application, keep in mind that it will be running in a resource constrained environment. While restricting the access of your Trusted Application, it allows Kinibi to protect your Trusted Application from other Trusted Applications.

3.2.2.1 TA Lifecycle

A Trusted Application is command-orientated. Client Applications access a service by opening a session with the Trusted Application and sending commands within the session. When the Trusted Application receives a command, it parses any message associated with the command, performs any required processing and then sends a response back to the Client Application.

When a Client Application opens a session with a Trusted Application, Kinibi creates a new process and loads the TA binary into it and starts execution of this process. This process is then called an instance of that Trusted Application. Trusted Application processes are single-threaded and have their own address space separating them from all other processes.

Kinibi-400A supports the full lifecycle model as described in the specification. Older Kinibi versions support only the multi-instance model. In these versions, there is a one-to-one relationship between a Client Application's session and an instance of a Trusted Application: When a CA opens a session to a TA, the TA is started and accepts commands in this session. A TA processes one command at a time. When the CA closes the session, the TA will destroy itself. If a CA opens another session to an already open TA, a new independent instance of this TA is created.

Limitations of older Kinibi versions, w.r.t. the TA lifecycle defined in the GP TEE API (400A):

- All TAs are multi-instance TAs, there are no single-instance TAs.

- One TA instance can only have one session at a time, there is no multi-session TA.
- TA lifetime is linked to CA session, no support for keeping alive instances.
- No support for TA Client API, no communication between TAs

3.2.2.2 TA Manifest

Starting from Kinibi-400A, developers can set static properties for their TA in a manifest file. That way a developer can set the TA lifecycle properties like `gpd.ta.singleInstance`, `gpd.ta.multiSession` and `gpd.ta.instanceKeepAlive`, as well as `gpd.ta.version` a `gpd.ta.description`. Custom properties are supported as well. The maximum size for all properties combined including name and value is 1024 bytes.

3.2.2.3 TA Interface

Each Trusted Application must implement a few callback functions, collectively called the "TA interface". These functions are the entry points called by the Kinibi TEE to create the instance, notify the instance that a new client is connecting, notify the instance when the client invokes a command and to notify when a client is disconnecting and also before destroying the instance. These functions must be implemented in order to link the TA binary. The TA is single-threaded, all entry points are linked into the same address space and Kinibi ensures that only one entry point is called at a time. When a TA returns from an entry point, the TA is either destroyed or blocked until the next entry point is called.

Table 1 lists the functions in the TA interface.

Table 1: TA Interface Functions

TA Interface Function	Description
TA_CreateEntryPoint	This is the Trusted Application constructor. It is called once and only once in the life-time of the Trusted Application instance. If this function fails, the instance is not created and the process associated with this Trusted Application is freed. After this function, Kinibi always calls the TA_OpenSessionEntryPoint.
TA_OpenSessionEntryPoint	This function is called whenever a client attempts to connect to the Trusted Application instance to open a new session. If this function returns an error, the connection is rejected and no new session is opened. In this function, the Trusted Application can attach an opaque void* context to the session. This context is recalled in all subsequent TA calls within the session.
TA_InvokeCommandEntryPoint	This function is called whenever a client invokes a Trusted Application command. The Kinibi TEE gives back the session context reference to the Trusted Application in this function call.
TA_CloseSessionEntryPoint	This function is called when the client closes a session and disconnects from the Trusted Application instance. The session context reference is given back to the Trusted Application by the Kinibi TEE.

	It is the responsibility of the Trusted Application to deallocate the session context if memory has been allocated for it.
TA_DestroyEntryPoint	<p>This is the Trusted Application destructor. The Kinibi TEE calls this function just before the Trusted Application instance is terminated.</p> <p>Termination of the TA depends on the TA lifecycle model and Kinibi version:</p> <ul style="list-style-type: none"> • keepAlive TAs will never be destroyed • multisession TAs will be destroyed after the last session has been closed • multi-instance TAs will be destroyed directly after the TA_CloseSessionEntryPoint. This is the only way before Kinibi-400 <p>When TA_DestroyEntryPoint returns, Kinibi destroys the process, frees all resources and notifies all secure drivers that the Trusted Application instance was destroyed.</p>

Table 2 summarizes client operations and the resulting Trusted Application effect.

Table 2: Correspondence between CA operation and TA Interface

Client Operation	Trusted Application effect
TEEC_OpenSession or TEE_OpenTASession	<p>If a new Trusted Application instance is needed to handle the session, Kinibi creates a new process and calls TA_CreateEntryPoint.</p> <p>Then TA_OpenSessionEntryPoint is called.</p>
TEEC_InvokeCommand or TEE_InvokeTACommand	<p>Kinibi unblocks the process and calls TA_InvokeCommandEntryPoint.</p>
TEEC_CloseSession or TEE_CloseTASession	<p>Kinibi unblocks the process and calls TA_CloseSessionEntryPoint.</p> <p>For a multi-instance TA or for a single-instance, non-keep-alive TA, if the session closed was the last session on the instance, then TA_DestroyEntryPoint is called.</p> <p>Otherwise, the instance is kept until the TEE shuts down.</p>
Client terminates unexpectedly	<p>Kinibi supports ordered shutdown of TA in case of CA crash.</p> <p>From the point of view of the TA instance, the behavior is identical to the situation where the CA requests the cancellation of all pending operations in that session, waits for the completion of all these operations in that session, and finally closes that session.</p> <p>Note that there is no way for the TA to distinguish between the client gracefully cancelling all its operations and closing all its</p>

	sessions and the implementation taking over when the client dies unexpectedly.
--	--

3.2.2.4 TA Secure Storage

To keep state between instances Kinibi has support for the GP persistent storage API. Using functions similar to `fopen`, `fclose`, `read`, `write`, and `seek`, Trusted Application can put object data into files. Those files are identified with a byte-string of up to 64 bytes. TAs cannot share files. Files or persistent objects are bound to one device and one TA. Only one instance of a TA can open a file at a time.

Persistent objects are stored in normal world files, with cryptographic protection to maintain compatibility and avoid tampering. The security guarantees offered by Kinibi depend on the version:

- Kinibi 300–303: Each object is stored in an individual file. The individual file size is exposed to the normal world, but the object contents and identifier are encrypted. Integrity checking is provided for individual objects as follows: the normal world may be able to modify the object store, but any such modification is detected and reported to the application via the error code `TEE_ERROR_CORRUPT_OBJECT`, except for the following:
 - A rollback of an individual object to a prior genuine version may not be detected.
 - The deletion of an individual object may not be distinguishable from the object being legitimately absent.
- Kinibi 310: All objects are stored inside the same normal world file. The normal world can observe the total storage size and rate of change, but object contents, name and ownership are not exposed. The object store is globally monitored for integrity; the normal world may be able to modify the object store, but any such modification is reported to the application at the latest when the object is accessed via the error code `TEE_ERROR_CORRUPT_OBJECT`, except for the following:
 - If the value of the TEE property `gpd.tee.trustedStorage.antiRollback.protectionLevel` is less than 1000, a rollback of the entire secure storage to a prior genuine version may not be detected.

Different size limitations apply depending on the version of Kinibi and on the integration:

- Kinibi 300–303 only supports relatively small files: The entire file must fit into the heap of the TA.
- Since Kinibi 310, the theoretical maximum file size is $2^{32}-1$ bytes. In practice, file size may be limited by the maximum storage size, which varies between integrations. Large files can be read and modified efficiently.

3.2.2.5 Implementing the TA Interface

A Trusted Application has to implement the TA Interface and can use the TEE Internal API. See the example code provided with the SDK.

An example of a basic Trusted Application would be something like:

```
TEE_Result TA_EXPORT TA_CreateEntryPoint(void)
{
    return TEE_SUCCESS;
}

void TA_EXPORT TA_DestroyEntryPoint(void)
{
}
```

```

TEE_Result TA_EXPORT TA_OpenSessionEntryPoint(
    uint32_t    nParamTypes,
    INOUT      TEE_Param pParams[4],
    OUT void**  ppSessionContext)
{
    return TEE_SUCCESS;
}

void TA_EXPORT TA_CloseSessionEntryPoint(INOUT void* pSessionContext)
{
}

TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(
    INOUT void* pSessionContext,
    uint32_t    nCommandID,
    uint32_t    nParamTypes,
    TEE_Param    pParams[4])
{
    switch(nCommandID)
    {
    case CMD_HELLO:
        // do some processing
        return TEE_SUCCESS;
    }
}

```

In between the entry points, the Kinibi TEE has code that runs when the TA starts and synchronizes with the CA. Every Trusted Application has to implement these functions, similar to the main() function in regular c-programs.

3.2.2.6 Implementing the Operations

To support a comprehensive communication protocol, a Trusted Application exports several commands and associated parameters. The TA_OpenSessionEntryPoint and the TA_InvokeCommandEntryPoint from the TA Interface define operation objects that can be used to communicate with the Client Application. An operation contains 4 parameters that have one of three possible directions: IN, OUT, INOUT. Parameters are typed and can either denote an immediate value or a reference to a memory region:

```

typedef union {
    struct {
        void*    buffer;
        size_t   size;
    } memref;
    struct {
        uint32_t a;
        uint32_t b;
    } value;
} TEE_Param;

```

The nParamTypes parameter defines the type and direction of the 4 parameters:

```
/* Parameter types */
```

```

#define TEE_PARAM_TYPE_NONE 0x0
#define TEE_PARAM_TYPE_VALUE_INPUT 0x1
#define TEE_PARAM_TYPE_VALUE_OUTPUT 0x2
#define TEE_PARAM_TYPE_VALUE_INOUT 0x3
#define TEE_PARAM_TYPE_MEMREF_INPUT 0x5
#define TEE_PARAM_TYPE_MEMREF_OUTPUT 0x6
#define TEE_PARAM_TYPE_MEMREF_INOUT 0x7

#define TEE_PARAM_TYPES(t0,t1,t2,t3) ((t0) | ((t1) << 4) | ((t2) << 8) |
((t3) << 12))

```

The macro `TEE_PARAM_TYPES` can be used to combine the 4 parameter definitions into the `nParamTypes` value.

The following sample code illustrates the usage:

```

TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(void* pSessionContext,
                                               uint32_t nCommandID,
                                               uint32_t nParamTypes,
                                               TEE_Param pParams[4])
{
    switch(nCommandID)
    {
        case CMD_SAMPLE_ROT13:
            if (nParamTypes != TEE_PARAM_TYPES(
                TEE_PARAM_TYPE_MEMREF_INOUT,
                TEE_PARAM_TYPE_NONE,
                TEE_PARAM_TYPE_NONE,
                TEE_PARAM_TYPE_NONE))
            {
                tLDbgPrintf(TATAG "bad params\n");
                return TEE_ERROR_BAD_PARAMETERS;
            }
            uint8_t* pOutput = pParams[0].memref.buffer;
            uint32_t nOutputSize = (uint32_t)pParams[0].memref.size;
            for(i=0; i<nOutputSize; i++) {
                pOutput[i] = pOutput[i]+13;
            }
            return TEE_SUCCESS;
        }
    }
}

```

The code shows the selection of a command with the switch statement and the treatment of the ROT13 command. The first check verifies the parameter type: Here only one parameter of type memory reference and direction input and output is accepted. The variables are declared to have easier access to the provided parameters. The for loop applies the rot13 disguise to the data.

In addition to OUT parameters, entry points also have a return value that can be used to communicate the overall result of the operation.

The Trusted Application developer defines the service interface by choosing the required command IDs and by associating a set of typed parameters for each command. Those definitions, together with application-specific data types and return codes should be put into a header file.

The following header illustrates a sample service interface:

```

/** Command ID's */
#define CMD_SAMPLE_ROT13    1
#define CMD_SAMPLE_HELLO   2

#define CMD_ROT13_PTYPEES  TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_INOUT,
TEE_PARAM_TYPE_NONE, TEE_PARAM_TYPE_NONE, TEE_PARAM_TYPE_NONE)
/**
 * ROT13 parameter description.
 * The ROT13 command will apply disguise to data provided in parameter 1.
 * Only characters (a-z, A-Z) will be processed.
 * Other data is ignored and stays untouched.
 */

#define RET_ERR_INVALID_BUFFER    1
#define RET_ERR_INVALID_COMMAND  2

```

3.2.3 TA-to-TA communication

Kinibi from 400A supports the GP TA-to-TA communication using the `TEE_OpenTASession()`, `TEE_InvokeTACommand()` and `TEE_CloseTASession()` functions. Any GlobalPlatform TA can use this API to call another GP TA (any TA can be a *client*). It depends on the way a TA is installed if the TA can be called in TA-to-TA communication (only some TAs can be a *server*).

For a GlobalPlatform TA to be a server, the TA must be already running or installed into Trusted Storage (TA-to-TA is not loading automatically System TA and SP TAs installed in mcRegistry). To make sure a TA can be called independently of the way the TA is installed, the developer has to use a multi-session TA and first open a session from a Client Application before opening a second session from a Trusted Application.

3.2.4 TEE Capabilities

Kinibi from 400A implements TEE properties via the GP properties API that give information about the capabilities of the TEE on this specific device. For example:

<code>com.trustonic.tee.isa.arm.neon</code>	Boolean	True if Kinibi supports NEON and Hardware Floating Point for TA
<code>com.trustonic.tee.tui.available</code>	Boolean	True if TUI is available

A TA can query such properties to decide which code path to execute. See the Kinibi API Documentation for the complete list of properties defined in each version of the product.

3.2.5 Extended APIs

Some extended APIs such as the Trusted User Interface API, the DRM API and others are also available for Trusted Applications developed with the GlobalPlatform APIs. The exact names and function signatures for these APIs are indicated in the Kinibi API Documentation.

3.3 USING THE LEGACY API

3.3.1 Client Applications

For Kinibi, communication between the Normal-World and the Secure-World is facilitated by writing data to a world shared memory buffer. This memory is configured such that both tasks have access to the shared section, each of them as part of their specified execution context. This is followed by initiating a notification, which initiates immediate transfer of control to the other world.

Notifications, also called signals or events, are required to have a corresponding synchronization mechanism. Signals, like interrupts, occur asynchronously and need to inform a thread to process the results of the event. On a TrustZone platform, these notifications are realized through hardware interrupts.

3.3.1.1 Trusted Application Connector

The Trusted Application Connector (TLC) in the Normal World is the counterpart to the Trusted Application in the Secure World. The TLC is responsible for establishing a connection with the Trusted Application and for providing the Trusted Application's features to the upper layers, e.g. an Android App.

It uses the Kinibi mcClient API to initiate starting and stopping of a Trusted Application and to exchange data with it. To the upper layer, e.g. the Android application, the TLC makes the Trusted Application's security features accessible. Therefore the TLC developer designs an API for the Normal-World application that corresponds to these security features. Android applications then use this Trusted Application via its specific TLC API.

A Trusted Application Connector can be viewed as a library for the functionality provided by one or more Trusted Applications. This involves:

- ◀ loading and opening a session to one or more Trusted Applications
- ◀ opening and closing sessions to one or more Trusted Applications
- ◀ marshalling function calls to a Trusted Application via TCI buffer and waiting for the Trusted Application to respond
- ◀ mapping and un-mapping additional memory to a Trusted Application session
- ◀ handling Trusted Application return codes
- ◀ closing sessions with the Trusted Applications

3.3.1.2 Using the mcClient API

The following sections provide a brief description of the Kinibi mcClient API functions to be used by the TLC for communication with the Trusted Application. See the Kinibi API Documentation for the full function definitions.

3.3.1.2.1 Device Sessions

Before communicating with a Trusted Application, the TLC needs to open a session to a Kinibi device. Currently there is only one standard device available (deviceId = MC_DEVICE_ID_DEFAULT), but in the future this may be extended by other TrustZone runtime environments, or Secure Elements providing the same API.

```
mcResult_t mcOpenDevice(  
    [in] uint32_t deviceId  
);  
  
mcResult_t mcCloseDevice(  
    [in] uint32_t deviceId  
);
```

```
[in] uint32_t deviceId
);
```

The counterpart to the `mcOpenDevice()` command is `mcCloseDevice()`, freeing all still allocated device resources. Trusted Application sessions need to be closed prior to closing the device.

3.3.1.2.2 Registering the platform specific context

To ensure proper interaction with the rich OS, the TEE Client library may need to interact with the host platform. For instance on Android, the TEE Client library needs to start a Service for proper operation of the TEE. The TEE Client library must be provided some platform specific data for that purpose:

```
void TEEC_TT_RegisterPlatformContext (
    void                *globalContext,
    void                *localContext);
```

The value of the context is platform dependent. Currently it is only used on Android.

3.3.1.2.2.1 On Android

The Client Application SHOULD call this function to provide both the current Java Virtual Machine (JVM) pointer (`JavaVM*`) as `globalContext` and an `android.app.Context` object as `localContext`, to be used to bind to the Android service(s) required for TEE operation. The caller must ensure that the context provided remains valid for any subsequent TEE Session. For this reason, the context provided MUST be the Application Context, returned by the `Context.getApplicationContext()` method, since this context is valid for the whole lifetime of an Android Client Application.

If the JVM pointer and `android.app.Context` are not registered before the call to either `TEEC_InitializeContext` (for GP applications) or `mcOpenDevice` (for legacy applications), some TEE functionality may not be available.

3.3.1.2.2.2 On other platforms

Calling this function has no effect.

3.3.1.2.3 Trusted Application Sessions

A Trusted Application session is the initial communication channel to the Trusted Application. It is required for any subsequent signaling and message passing actions.

```
mcResult_t mcOpenTrustlet(
    mcSessionHandle_t *session,
    mcSpid_t          spid,
    uint8_t           *trusted_application,
    uint32_t          tLen,
    uint8_t           *tci,
    uint32_t          tciLen
);
```

`mcOpenTrustlet()` opens a session to a Trusted Application specified by the Service Provider ID and returns a handle for the session. It is required that a device has already been opened before. You have to prepare the 'session' structure's `deviceId` field with the `deviceId` that you opened before. The caller must provide the Trusted Application binary so that the Trusted Application is loaded in Kinibi. The caller must also allocate the communication buffer before calling this function.

```
mcResult_t mcCloseSession(
```

```
[in] mcSessionHandle_t *session
);
```

mcCloseSession() closes a session and all its allocated resources.

3.3.1.2.4 Message Exchange and Signaling

Kinibi's Inter World Communication signaling part is based on two functions:

```
mcResult_t mcNotify(
[in] mcSessionHandle_t *session
);
```

mcNotify() sends a signal to a Trusted Application to inform it about the availability of new data within the WSM buffer.

```
mcResult_t mcWaitNotification(
[in] mcSessionHandle_t *session,
[in] int32_t timeout
);
```

The TLC uses mcWaitNotification() to wait for the Trusted Application to respond. The timeout parameter sets the maximum amount of time that the TLC will wait for a certain Trusted Application session to respond.

Errors in the Secure World can be queried with the mcGetSessionErrorCode() call. See the Kinibi API Documentation for further details.

3.3.1.2.5 Kinibi version information

TLC can use mcGetMobiCoreVersion() to retrieve the following version information.

```
__MC_CLIENT_LIB_API mcResult_t mcGetMobiCoreVersion (
    uint32_t deviceId,
    mcVersionInfo_t* versionInfo)
```

Parameter	Meaning
productId	Product ID of Mobicore; zero-terminated
versionMci	Version of Mobicore Control Interface
versionSo	Version of Secure Objects
versionMclf	Version of MobiCore Load Format
versionContainer	Version of MobiCore Container Format
versionMcConfig	Version of MobiCore Configuration Block Format
versionTlApi	Version of MobiCore Trustlet API Implementation
versionDrApi	Version of MobiCore Driver API Implementation
versionCmp	Version of Content Management Protocol

3.3.1.2.6 Memory Mapping

Memory mapping enables provisioning large amounts of data to a Trusted Application with zero copy.

```
mcResult_t mcMap(
[in] mcSessionHandle_t *sessionHandle,
[in] void *buf,
[in] uint32_t len,
[out] mcBulkMap_t *MapInfo
);
```

mcMap() maps additional memory to a Trusted Application. A TLC should use this whenever it needs to provide a Trusted Application with data stored in Normal-World memory (like a buffer holding the clear text for a cryptographic operation).

The amount of memory that a TLC could share with a TA depends on the Kinibi version and the memory layout that the TA is using. The TLC can query the MCI version using mcGetVersionInfo to find out if Kinibi has the right version:

- ◀ Prior to Kinibi 410A and MCI version 1.7:
 - ◀ When the TA uses the legacy or the extended memory layout, the TLC can map up to 4 chunks of max. 1 MB each to the TA. See also Fig. 3 Trusted Application Address Space Layout.
- ◀ Starting from Kinibi 410A and MCI version 1.7:
 - ◀ When the TA uses the extended memory layout, the size of the memory block that the TLC can map to the TA is limited by the TA's virtual address space which is 124 MB.

Hint: The amount of memory blocks concurrently mapped to the Trusted Application is limited to four (4). Use the mcUnmap(...) function to unmap not in use memory blocks.

```
mcResult_t mcUnmap(
[in] mcSessionHandle_t *sessionHandle,
[in] void *buf,
[in] mcBulkMap_t *MapInfo
);
```

mcUnmap() is the counterpart to the mapping function and unmaps previously mapped blocks of memory.

3.3.1.3 Client Kernel API

The mcClient API is also available at the kernel level and can be called by kernel modules. The mcClient API functions are the same in user mode and in kernel mode.

3.3.2 Trusted Applications

3.3.2.1 Trusted Application Structure

A Trusted Application implements a fairly simple startup, main loop, and shutdown process. See the example code provided with the SDK.

An example of a basic Trusted Application main routine block would be something like:

```
void tlMain(
uint8_t *tciData,
uint32_t tciLen
) {
    // Check TCI size
    if (sizeof(tci_t) > tciLen) {
        // TCI too small -> end Trusted Application
        tlApiExit(EXIT_ERROR);
    }
    // Trusted Application main loop
    for (;;) {
        // Wait for a notification to arrive
        tlApiWaitNotification(INFINITE_TIMEOUT);

        // ***** //
        // Process command
        // ***** //

        // Notify the TLC
        tlApiNotify();
    }
}
```

The tlMain routine has two parameters:

- ◀ The first one is a pointer to the world shared memory TCI buffer, allocated by the Kinibi driver during the initialization of a new session.
- ◀ The later provides the length of the buffer. This must be something below 4kB, which is the maximum length of an allocatable block of memory.

After a successful initialization (where you should check if the provided TCI memory is big enough) the Trusted Application will loop infinitely until its process is killed by the Kinibi runtime due to a close command from the TLC or a fatal exception.

In its main loop, the Trusted Application will repeatedly:

1. wait for a notification from the TLC via the tlApiWaitNofitication() command

Hint: Infinite timeout is recommended, not polling.

2. process the data from the TCI buffer and write back the result to the buffer
3. signal the TLC that there is a response available with tlApiNotify()

This pretty much looks the same for every Trusted Application. Have a look at the samples at the end of this document for more details about this approach.

3.3.2.2 Advanced TCI Communication Protocol

If your Trusted Application needs to handle more than one command, or needs to support a more comprehensive communication protocol, it is advised to add a command / response protocol on top of the TCI. This kind of protocol allows the Trusted Application interface to define commands and responses reflecting the provided functionality of the Trusted Application. The messages need to be exchanged via the

TCI world share memory buffer, established by the TLC during the creation of a new Trusted Application session.

See for a suggestion of such a protocol.

Pos	Type	Size	Description
1	Header	uint32_t (4 bytes)	Header may be: 0 < Command ID < 0x7fffffff, or a Response ID (Command ID 0x80000000)
2	Payload	TCI_SIZE - 4	Holds the command or response data

Table 3: TCI command- and response protocol structure

The first field in the communication structure would hold the header which would either be a command for the Trusted Application or a response for the TLC.

The second field defines a payload field to hold the data belonging to the corresponding header.

Hint: The length of the data can be of arbitrary size, but the header and data field must fit in the maximum TCI buffer size, which is currently at 4kB.

For each function your Trusted Application provides to the TLC, you would define a command ID, along with a specific data structure for the payload.

Here is an excerpt of a Trusted Application header file (API) of the definitions of command IDs (TCI definition) as well as their data structures:

```
/** Command ID's */
#define CMD_SAMPLE_SHA1          1
#define CMD_SAMPLE_SHA256      2

/** Message digest command (SHA1 or SHA256) */
typedef struct {
    tciCommandHeader_t commandId;
    uint8_t* srcBuffer;
    uint32_t srcLen;
} cmdMD_t;
```

The code shows the API for a Trusted Application providing the SHA1 and SHA256 hash functions to the Normal-World. Both hashes share the same command structure:

- ◀ An ID to tell the Trusted Application which kind of hash we want
- ◀ A pointer to the source buffer holding the data we want to calculate a hash from
- ◀ The length of the input data

As the two hash functions differ in their resulting output size, two separate response structures have been defined, both starting with the header field followed by the hash result.

```
/** SHA1 response */
typedef struct {
    tciResponseHeader_t responseID;
    uint8_t hash[20];
} rspSha1_t;

/** SHA256 response */
```

```
typedef struct {
    tciResponseHeader_t responseID;
    uint8_t hash[32];
} rspSha256_t;
```

Now that all possible command and response structures have been defined, declare the TCI message as a Union over them:

```
/** TCI message data. */
typedef union {
    tciCommandHeader_t    commandHeader;
    tciResponseHeader_t  responseHeader;
    cmdMD_t              cmdMD;
    rspSha1_t            rspSha1;
    rspSha256_t          rspSha256;
} tciMessage_t;
```

The Union allows the Trusted Application to easily access the command ID within the TCI buffer and map it to the according command structure.

On top of that, the Trusted Application can define various error codes:

```
#define RET_ERR_INVALID_BUFFER      3
#define RET_ERR_INVALID_COMMAND    4
```

Hint: Notice that there is no need for an "OK" return code for the good case. Using a bit mask flipping the most significant bit of the command ID is the suggested way to signal the TLC a valid response to a certain command.

Now getting back to the previously described tlMain loop, applying the new definitions would change it in the following way:

1. Whenever the Trusted Application receives a notification, it first checks the provided command ID from the TCI. If the command ID is unknown, the Trusted Application sets the corresponding response ID and the payload to RET_ERR_INVALID_COMMAND.
2. The Trusted Application executes the corresponding command.
3. If an error occurs during the execution of a command the header field will be set to an error value and an empty response will be returned.
4. In case the Trusted Application function did return successfully, the command ID is converted to a response ID and written to the response header. Furthermore, the response data structure will be returned in the payload field.

Here is the resulting code:

```
/**< Responses have bit 31 set */
#define RSP_ID_MASK (1U << 31)
#define RSP_ID(cmdId) (((uint32_t)(cmdId)) | RSP_ID_MASK)
#define IS_CMD(cmdId) (((uint32_t)(cmdId)) & RSP_ID_MASK) == 0)
#define IS_RSP(cmdId) (((uint32_t)(cmdId)) & RSP_ID_MASK) == RSP_ID_MASK)
```

```
void tlMain(
uint8_t *tciData,
uint32_t tciLen
) {
    tciResponseHeader_t responseId;
    tciCommandHeader_t commandId;

    // Check TCI size
    if (sizeof(tci_t) > tciLen) {
        // TCI too small -> end Trusted Application
        tlApiExit(EXIT_ERROR);
    }

    tciMessage_t* tciMessage = (tci_t*) tciData;
    // Trusted Application main loop
    for (;;) {
        // Wait for a notification to arrive
        tlApiWaitNotification(INFINITE_TIMEOUT);

        commandId = tciMessage->commandHeader.commandId;

        // Check if the message received is (still) a response
        if (!IS_CMD(commandId)) {
            // Tell the Normal-World a response is still pending
            tlApiNotify();
            continue;
        }
        // Call Trusted Application functions according to command ID
        switch (commandId) {
            case CMD_SAMPLE_SHA1:
                ret = processCmdSha1(tciMessage->cmdMD);
                break;
            case CMD_SAMPLE_SHA256:
                ret = processCmdSha256(tciMessage->cmdMD);
                break;

            default:
                ret = RET_ERR_UNKNOWN_COMMAND;
                break;
        }

        // Set up response header
        tciMessage->responseHeader.responseId = RSP_ID(commandId);
        tciMessage->responseHeader.returnValue = ret;

        // Notify back the TLC
        tlApiNotify();
    }
}
```

3.3.2.3 Security Considerations



Trusted Applications must be especially careful when handling data coming from the Client Application through World-Shared Memory buffers: the content of the buffers may change at any time, even between two consecutive memory accesses by the Trusted Application. This means that the Trusted Application should be carefully written to avoid any security problem if this happens. If values in the buffer are security critical, the Trusted Application should always read data only once from a World-Shared Memory buffer and then validate it. It must not assume that data written to the buffer can be read unchanged later on.

The Trusted Application's code and data regions lie within the first MBs (Only 1st MB with the legacy layout) of its memory / address space. Any constant data structures linked into the Trusted Application binary will be mapped to this memory area. Depending on the type of Trusted Application this may include security critical data like keys, cryptographic seeds, PINs...

It is recommended to call `tlApiGetVirtMemType()` to check if a buffer is only accessible by the Secure-World or shared with the Normal-World.

Kinibi and the TrustZone concept will prevent direct access to Secure-World memory by the Normal-World, but it cannot stop a TA accidentally leaking security relevant data by copying it to WSM regions.

Using the SHA256 sample Trusted Application provided in this document, one could think of the following attack scenario:

- ◀ The Trusted Application calculates a hash over a memory area, it gets mapped by the TLC and is informed by the `srcBuffer` pointer element of the `cmdMD_t` structure.
- ◀ The Trusted Application assumes that this buffer has been mapped by the TLC using the Kinibi Client API prior to notifying the Trusted Application. The Non Secure buffers are mapped into the TA space but there is no guarantee on the location.
- ◀ As the Trusted Application never verifies that this is actually the case, a malicious TLC could point the Trusted Application to its own code or data region, providing a pointer with a value below 1MB ($< 0x100000$)!
- ◀ Although the Trusted Application only returns the SHA256 hash of this memory, a TLC could repeatedly call this function only hashing one byte at a time.
- ◀ Comparing each result with a table containing the hash results for the values 0 – 256 (a hash map), one can quickly derive the content of the memory address.
- ◀ This would allow an attacker to read the clear text Trusted Application binary, including code and data, like keys!

Note: It is vital to carefully define and implement the TA API to protect direct or indirect access to internal memory structures. Trusted Application should use `tlApiGetVirtMemType` to test memory type, and accept only pointers pointing to Normal-World memory.

3.3.2.4 Kinibi tlAPI

The Kinibi tlAPI is used by Legacy Trusted Applications. It defines all the functionality that Kinibi can offer to a Trusted Application:

- ◀ A set of functions for inter-world communication.
- ◀ Kinibi system information and functions.
- ◀ Cryptographic processing.
- ◀ Secure object functions for binary data encryption.

Please see the Kinibi API Documentation for complete reference.

3.3.2.5 Secure Objects

For storing Trusted Application data persistently, Kinibi provides the Secure Object API functions. Secure objects are wrapped and unwrapped using internal Kinibi keys and therefore provide convenience functions for secure data storage in the Normal-World.

Major Secure Object functionalities are:

- ◀ **Data Integrity.** A Secure Object contains a message digest (hash) that ensures data integrity of the user data. The hash value is computed and stored during the wrap operation (before data encryption takes place) and recomputed and compared during the unwrap operation (after the data has been decrypted).
- ◀ **Confidentiality.** Secure Objects are encrypted using context-specific keys that are never exposed, neither to the normal world, nor to the Trusted Application. It is up to the user to define how many bytes of the user data are to be kept in plain text and how many bytes are to be encrypted. The plain text part of a Secure Object always precedes the encrypted part.
- ◀ **Authenticity.** As a means of ensuring the trusted origin of Secure Objects, the unwrap operation stores the Trusted Application ID (SPID, UUID) of the calling Trusted Application in the Secure Object header (as Producer). This allows Trusted Applications to only accept Secure Objects from certain partners. This is most important for scenarios involving secure object sharing.

Design Considerations:

- ◀ The TLC is responsible for storing the Secure Object in Normal-World. Kinibi does not support direct access to the rich OS file system. Therefore the TA needs to copy the wrapped Secure Object into WSM and notify the TLC to store it persistently. Later the TLC needs to provide Secure Object to the TA via WSM once needed.
- ◀ Secure Objects can be bound to the TA itself or a broader scope. This enables devices binding scenarios as well as sharing data securely between TAs (e.g. via a common TLC).
- ◀ Having an unencrypted, but signed, data part enables the implementation of a header or meta-data part. This can simplify Secure Object handling in both worlds.

Note: Source of wrap must be in internal TA memory (static buffers, BSS). Destination of unwrap must be in internal TA memory.

Don't do in-place decryption in WSM to avoid leaking unencrypted data!

The API is composed of the following two operations:

```
tlApiResult_t tlApiWrapObject(
```

```

const void *src,
size_t plainLen,
size_t encryptedLen,
mcSoHeader_t *dest,
size_t *destLen,
mcSoContext_t context,
mcSoLifeTime_t lifetime,
const tlApiSpTrusted ApplicationId_t *consumer,
uint32_t flags);

tlApiResult_t tlApiUnwrapObject(
void *src,
size_t srcLen,
void *dest,
size_t *destLen,
uint32_t flags);

```

Secure Object Context

The concept of context allows for sharing of Secure Objects. There are three kinds of context:

- ◀ MC_SO_CONTEXT_TLT: Trusted Application context. The secure object is confined to a particular Trusted Application. This is the standard use case.
 - ◀ PRIVATE WRAPPING: If no consumer was specified, only the Trusted Application that wrapped the Secure Object can unwrap it.
 - ◀ DELEGATED WRAPPING: If a consumer Trusted Application is specified, only the Trusted Application specified as 'consumer' during the wrap operation can unwrap the Secure Object. Note that there is no delegated wrapping with any other contexts.
- ◀ MC_SO_CONTEXT_SP: Service provider context. Only Trusted Applications that belong to the same Service Provider can unwrap a secure object that was wrapped in the context of a certain service provider.
- ◀ MC_SO_CONTEXT_DEVICE: Device context. All Trusted Applications can unwrap secure objects wrapped for this context.

Secure Object Lifetime

The concept of a lifetime allows limiting how long a Secure Object is valid. After the end of the lifetime, it is impossible to unwrap the object.

Three lifetime classes are defined:

- ◀ MC_SO_LIFETIME_PERMANENT: Secure Object does not expire. HINT: Only if the platform supports the required hardware features (like Secure-World non-volatile monotonic counters), this is protected against replay attacks.
- ◀ MC_SO_LIFETIME_POWERCYCLE: Secure Object expires on reboot.
- ◀ MC_SO_LIFETIME_SESSION: Secure Object expires when Trusted Application session is closed. The secure object is thus confined to a particular session of a particular Trusted Application. Note that session lifetime is only allowed for private wrapping in the Trusted Application context MC_SO_CONTEXT_TLT.

Secure Object Consumer

The consumer parameter is only valid for the context MC_SO_CONTEXT_TLT (DELEGATED WRAPPING). It defines which Trusted Application (SPID, UUID) is allowed to unwrap that Secure Object.

This identity (of type tlApiSpTrusted ApplicationId_t) is used for delegation purposes and to which Trusted Application we delegate the secure object, i.e. which Trusted Application should unwrap the secure object. Here we state that the Trusted Application (and only that Trusted Application) having the UUID "0,9,0,9,0,0,0,0,0,0,0,0,0,0,0,0" can unwrap the secure object which will be saved in the normal world.

```
static const tlApiSpTrusted ApplicationId_t consumerTid = {
    0,
    { 9,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0 }
};

ret = tlApiWrapObject(userData,
                      UDATA_PLAIN_LEN,
                      UDATA_ENC_LEN,
                      (mcSoHeader_t *)soDataBuf,
                      &soLength,
                      MC_SO_CONTEXT_TLT,
                      MC_SO_LIFETIME_PERMANENT,
                      &consumerTid,
                      TLAPI_WRAP_DEFAULT);
```

The TLAPI_WRAP_DEFAULT flag is mapped (using #define) in the Kinibi and the Trusted Application developer should use this flag when calling the wrapper function.

When wanting to unwrap the secure object, this is done within the Trusted Application of the UUID "0,9,0,9,0,0,0,0,0,0,0,0,0,0,0,0".

The Trusted Application calls:

```
ret = tlApiUnwrapObject((mcSoHeader_t*)pMessage->data,
                       pMessage->cmdSOUnwrapDelegate.len,
                       userdataDest,
                       &userdataDestLength,
                       TLAPI_UNWRAP_PERMIT_DELEGATED |
                       TLAPI_UNWRAP_DEFAULT);
```

The description of the use case is described below.

Precondition: The Trusted Application container is installed.

Starting point: Trusted Application Connector A is in waiting for data from Trusted Application A. Trusted Application B is waiting for data from Trusted Application Connector B.

1. Data buffer and identity (for delegation) of Trusted Application B is sent as input to tlApiWrapObjectExt
2. The data buffer is wrapped and (some part) encrypted
3. The wrapped data (SO) is sent to TLC A
4. The TLC A saves the data to file

5. The SO is read from file by TLC B
6. The SO is sent to Trusted Application B
7. Trusted Application B calls `tlApiUnwrapObjectExt` using the SO as input.
8. The data buffer (see 1.) is returned. The Security functionality checks if Trusted Application B has the right to decrypt the SO. Since the identity of Trusted Application B was set (see 1.) this is possible.

3.3.2.6 Endorsement API

Kinibi provides an API which allows a Trusted Application to sign data and prove that it is generated in a genuine TEE and in the right Trusted Application. This feature is important for service providers such as network operators.

The data is signed in Kinibi with a key derived from the Hardware Unique Key and is encrypted with Trustonic public endorsement key. The encrypted signed data can be exported to a server of the service provider and the service provider can then verify the signature through the Trustonic Directory server interface.

The function to generate such endorsement is `tlApiEndorse()` and is described in the Kinibi API Documentation.

3.3.2.7 Checking API return value

Most of the API calls return error code of type `tlApiResult_t`.

It is always recommended to check that the returned value is `TLAPI_OK`.

In some cases, Trusted Application may want to check for a specific error.

This checking should be implemented using the `TLAPI_ERROR_MAJOR` macro.

The macro returns the stable part of the error code. The remaining part of error code contains detail code, which may change between Kinibi releases.

3.3.3 Trusted User Interface

The Trusted User Interface (TUI) feature in Kinibi allows a TEE to interact directly with the user via the common display and touch screen. Kinibi temporarily protects those using TrustZone.

The main objective of the TUI is to protect the confidentiality and integrity of the information exchanged between a Trusted Application and the user against the NWd OS (Android).

This main objective is achieved by three features:

- ◀ Secure Input: The information entered by the user to a Trusted Application cannot be derived or modified by any software within the NWd OS or by another unauthorized Trusted Application.
- ◀ Secure Display: The information displayed by the Trusted Application cannot be accessed, modified, or obscured by any software within the NWd OS or by another unauthorized Trusted Application.

The secure display should be completed by a Secure Indicator.

- Security Indicator: The Trusted Application securely displays a secret, previously shared with the user, making the user confident that the screen displayed is actually displayed by a Trusted Application.

The TUI targets quite simple use cases like message display, PIN or password entry. These use cases require exclusive access of UI resources, not permanently but for a short period of time. This period is called a TUI session.

Within a TUI session, the SWd must have the full control of the UI resources. Thus the NWD drivers are suspended at the beginning of the session, and resumed after the session. Particularly the NWD must not be aware of any input event within the session.

A TUI session may be cancelled in case of some system events:

- Reset or turn off the device,
- Screen off,
- Incoming call

Trusted Applications can use the Trusted User Interface through a simple API which let them display data on the screen and get user inputs.

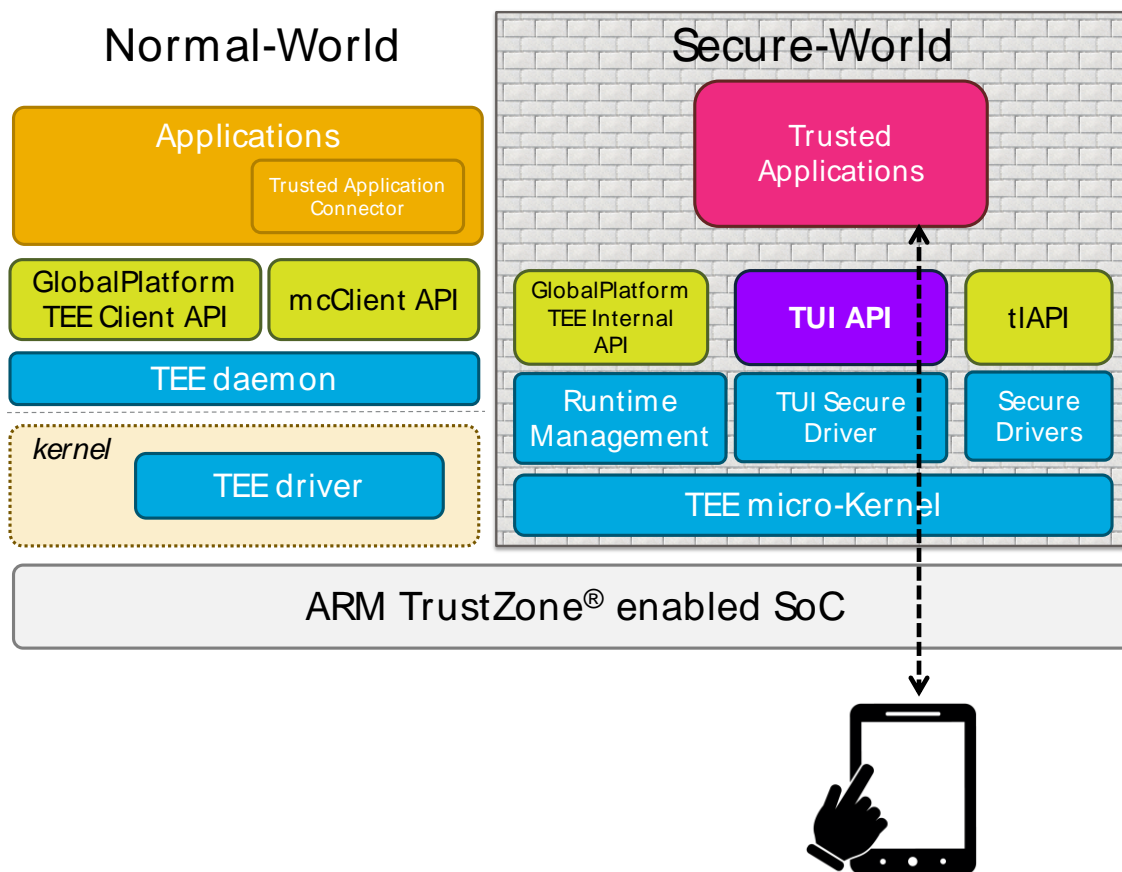


Figure 5: Trusted User Interface API.

3.3.3.1 TUI configuration

A Trusted Application that uses TUI is responsible for its own display layout. It may be composed knowing the touchscreen metrics.

```
typedef struct {
    uint32_t    grayscaleBitDepth;
    uint32_t    redBitDepth;
    uint32_t    greenBitDepth;
    uint32_t    blueBitDepth;
    uint32_t    width;
    uint32_t    height;
    uint32_t    wDensity;
    uint32_t    hDensity ;
} tlApiTuiScreenInfo_t, *tlApiTuiScreenInfo_ptr;

__TLAPI_EXTERN_C tlApiResult_t tlApiTuiGetScreenInfo (
    tlApiTuiScreenInfo_ptr screenInfo)
```

3.3.3.2 TUI Session

To use the TUI, a Trusted Application has to open a TUI session. When the Trusted Application has finished the user interaction, it must close the TUI session.

Except for the configuration, all TUI functions must be called within a TUI session.

```
tlApiResult_t tlApiTuiInitSession(
    void
);

tlApiResult_t tlApiTuiCloseSession(
    void
);
```

3.3.3.2.1 Normal world integration

The Kinibi TUI handles the necessary Normal-World integration.

The TUI application-developer does not have to care about incoming phone calls, screen off or power down events in his Normal-World application. He can use a regular TLC to communicate in the usual way with the Trusted Application. Only the Trusted Application interacts with the TUI feature using the TlApiTui functions.

When the Trusted Application starts the TUI session, Kinibi takes control over the display. For the TLC it is as if another Android activity occupies the screen for the duration of the TUI session.

3.3.3.2.2 Error code checking

The Trusted UI session can be interrupted at any time by phone calls and power management events. Hence the Trusted Application cannot expect to complete the user interaction every time. The developer has to foresee restarting the interaction and possibly saving and restoring interaction state. As a consequence, every TlApiTui call that is associated with the TUI session can fail with an error code indicating that the Trusted Application no longer has a TUI session. It is therefore mandatory that the Trusted Application always checks the return values of all TlApiTui functions.

```
#define E_TLAPI_TUI_NO_SESSION    0x00000501
```

3.3.3.2.3 Single user service

Only one Trusted Application can use the Trusted UI at a time and it controls the whole screen. If a Trusted Application tries to open a TUI session while another Trusted Application already uses it, an error code is returned.

```
#define E_TLAPI_TUI_BUSY 0x00000502
```

3.3.3.3 Secure display

To give the user visual information, a Trusted Application can draw image files on the secured display. Kinibi supports PNG files.

```
tlApiResult_t tlApiTuiSetImage(
    [in] tlApiTuiImage_t *image,
    [in] tlApiTuiCoordinates_t coordinates
);

typedef struct {
    void*      imageFile;
    uint32_t   imageFileLength;
} tlApiTuiImage_t;
```

Using the coordinates, the position of the image can be defined. Kinibi will read the image header to find out about the width and height of the image. Requests to draw partially outside of the display will be rejected. The Trusted Application has to provide the image in a buffer in the Trusted Application address space.

Note on address space limitations and buffer sizes

Modern smartphone display resolutions range from 1024 x 600 pixels to 1920 x 1080 pixels, totaling in a frame buffer size of 2 MB to 6 MB. Tablets devices with screen resolutions of up to 2560 x 1600 pixels feature a frame buffer size of around 12 MB. PNG compression algorithms can reduce images with graphical content by a factor of 6 to 25. So a typical full-screen user interface should well fit into a 200 KB PNG image.

Note that the developer is responsible for protecting the resources to be displayed, such as the images.

3.3.3.4 Secure input

To receive trusted input from the user during a TUI session, the Trusted Application has to wait for a notification and then call the driver to get the touch event data.

```
tlApiResult_t tlApiTuiGetTouchEvent(
    [out] tlApiTuiTouchEvent_t *touchEvent
);

typedef struct {
    tlApiTuiTouchEventType_t   type;
    tlApiTuiCoordinates_t     coordinates;
} tlApiTuiTouchEvent_t;

typedef enum {
    TUI_TOUCH_EVENT_RELEASED = 0,
    TUI_TOUCH_EVENT_PRESSED  = 1,
} tlApiTuiTouchEventType_t;
```

```
typedef struct {
    uint32_t    xOffset;
    uint32_t    yOffset;
} tlApiTuiCoordinates_t;
```

3.3.3.4.1 Asynchronous driver interface

Trusted Applications in Kinibi are already event driven, based on the notification and wait- for-notification primitives. These functions were primarily used to communicate with the TLC. For the Trusted User Interface, the Trusted Application can actually wait for an event from a TLC and for an event from a driver. That way, the Trusted Application remains single threaded, can receive abort or timeout commands from the TLC and, at the same time, wait for hardware-induced events like touches on a touch screen.

At the API level, Trusted Applications can call a function like `tlApiGenerateKey()` and this blocks the Trusted Application until the function returns. But the function flow can also be split into three parts:

- < `driverProgramOperation()`,
- < `waitNotification`,
- < `driverGetOperationResult()`

For the Trusted User Interface, this means that once the Trusted Application has opened a TUI session, it receives notifications for touch events and can receive the event data with the `getTouchEvent()` functionality. Note also that any normal-world-induced abort event creates one last notification that also closes the TUI session.

3.3.3.4.2 Trusted Application state machine

The developer has to decide which parts of the application logic have to be protected through a Trusted Application. A Trusted User Interface is also a Graphical User Interface and should follow common design principles for GUIs. The developer may implement helper functionalities such as font management to manage UI components in addition to the business logic.

Since the `waitNotification()` function does not indicate the origin of the notification to the Trusted Application, the Trusted Application has to maintain state and act depending on what it expects in a certain situation. If the Trusted Application hasn't enabled the Trusted UI session yet, there is no use in calling the `getTouchEvent()` function. If the TUI session is started, maybe certain commands from the TLC should be ignored. If the Trusted Application implements an input field, characters already entered and the current cursor position must be remembered. If the Trusted Application implements multiple dialogs it might want to track the dialog that is currently active.

3.3.3.4.3 Input filtering and button emulation

Modern touch controllers create many touch events per second and they are all sent to the Trusted Application that has opened a Trusted UI session. It is up to the Trusted Application to apply relevant event filtering and cumulate hardware touch events to application logic input events. For every touch event, Kinibi TUI holds the coordinates and the Trusted Application has to match those coordinates against a list of possible sensitive areas to determine a "button" press action.

3.3.3.4.4 TUI Event loop

3.3.3.4.4.1 Double buffering

Starting at API level 6, the TUI features double buffering. All the graphical drawing functions of the TUI draws into an off-screen buffer. The content drawn does not appear to the user on screen. The TA must call the function `tlApiTuiFlipFrameBuffer()` to display the copy the backbuffer into the front buffer, hence making it appear on screen.

3.3.3.4.4.2 Maximizing performances

To prevent any tearing effect from appearing on screen, the function `tlApiTuiFlipFramebuffer()` is blocking until the content of the backbuffer is displayed on screen. Therefore the function `tlApiFlipFramebuffer()` may take up to 16ms (on a typical 60 Hz display) to complete. This is a relatively long time during which the TA is not able to process any events. As a consequence, the good balance between these two objectives must be found:

1. Call `tlApiTuiFliipFramebuffer` frequently enough so that the UI remains smooth.
2. Let the TA process as may events as possible between two calls to `tlApiTuiFlipFramebuffer`.

To achieve this goal, the following event loop structure is recommended.

```

while (!(is_done() && touch_released) ) {

    if(!did_something) {
        tlApiWaitNotification(TLAPI_INFINITE_TIMEOUT);
    }

    tlApiGetSecureTimestamp(&ts1);
    unsigned int x, y;
    do {
        tlApiTuiTouchEvent_t touchEvent[32] = {};
        int i = 0;

        touch_released = false;

        /* Read as many events as possible to empty the queue */
        do {
            res = tlApiTuiGetTouchEvent(&touchEvent[i%32]);
            if (res == E_TLAPI_TUI_NO_EVENT) {
                break;
            }
            i++;
            did_something++;
        } while((TUI_TOUCH_EVENT_RELEASED != touchEvent[(i-1)%32].type));

        /* if no events were available in the queue since the last time,
         * just wait for another notification
         */
        if (i < 1) {
            break;
        }

        /* Process all of the events just read, such as drawing feedback on the
back
         * buffer, etc. You should not flip the framebuffer at this point.
         * The exact processing here depends on the use case. For some use
         * case, all the touch events may be useful, for some other only the
         * latest touch event is important.
         */

        /* half of the period dedicated to processing events, once this time
         * has elapsed, just refresh the screen. */
        tlApiGetSecureTimestamp(&ts2);
        if((ts2 - ts1) >= 8333)
        {
            break;
        }
    } while(1);
    res = tlApiTuiFlipFramebuffers();
}

```

3.3.4 DRM API

Kinibi provides a DRM API for Trusted Applications to process DRM Content. This API is simple and hides all the complexity of data decryption, decoding and rendering to the system and Secure Driver.

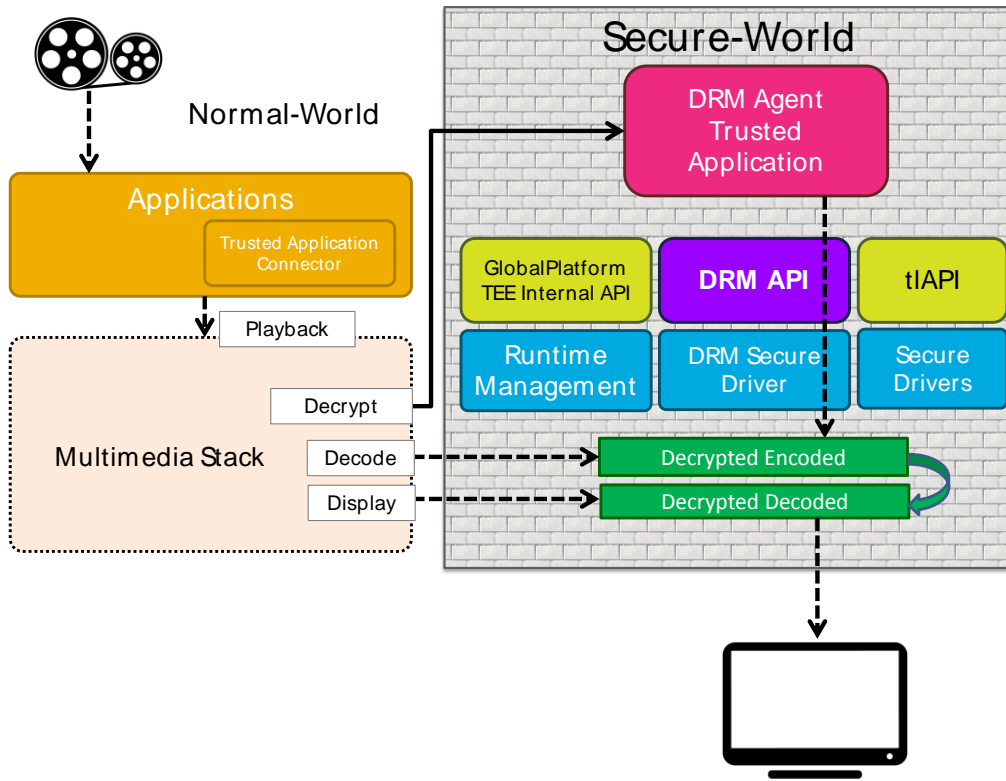


Figure 6: Kinibi DRM API.

The DRM API is made of one main function, `tIApiDrmProcessContent()` which processes the encrypted content. The caller must have created a decryption context first with the appropriate key and parameters for the content decryption.

```
int32_t tIApiDrmProcessDrmContent (
    uint8_t          sHandle,
    TL_DRM_DecryptContext decryptCtx,
    uint8_t          *input,
    TL_DRM_InputSegmentDescriptor inputDesc,
    uint16_t         processMode,
    uint8_t          *rfu);
```

4 USING THE SDK

4.1 TOOLCHAIN INSTALLATION

Different toolchains are used to build the Client and the Trusted Applications.

The Trusted Applications can be built with a GCC NONE EABI toolchain from Linaro.

The toolchain to be used for the Client Applications depends on the targeted Rich OS.

- For Android, an ARMv7 or ARMv8 from the Android NDK must be used.
- For other Linux distributions, it is recommended to use the default toolchain used in your BSP.
- For Windows, Visual Studio must be used.

4.1.1 System Requirements

For the development PC, TRUSTONIC recommends at least a 2.0GHz CPU and 2-4GB RAM.

Trusted Application development requires a Shell environment.

We recommend using Ubuntu 16.04 or later (www.ubuntu.com).

It is also possible to use MinGW + MSYS (<http://www.mingw.org/>) under Microsoft Windows 7.

For the debugging on a device or Kinibi emulation, a development device with Kinibi is needed.

Hint: Commercial devices [ComDev] with Kinibi can be used to run Trusted Applications. But debugging interfaces like COM or JTAG are disabled, user rights are restricted and older release software prevents debug outputs.

4.1.2 Trusted Applications

4.1.2.1 GCC / Linaro Compiler

The Linaro GCC 8.3 is the GNU development tool-chain for ARM Embedded Processors. It includes compiler tools, libraries and debugger for ARM base platforms.

2 versions of the toolchain are currently recommended by Trustonic, both can be downloaded from web site: <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads>.

- The GCC Compiler to build 64-bit versions, AArch64 ELF bare-metal target (aarch64-elf) :
`gcc-arm-8.3-2019.03-x86_64-aarch64-elf.tar.xz`
- The GCC Compiler to build 32-bit versions, AArch32 bare-metal target (arm-eabi)
`gcc-arm-8.3-2019.03-x86_64-arm-eabi.tar.xz`

You could install the whole tool-chain by simply decompressing the archive inside the chosen directory.

1. Download the archive
On Linux the archive name is `gcc-linaro-7.2.1-2017.11-x86_64_aarch64-elf.tar.xz`
2. Untar the archive in the current folder

```
tar xf gcc-arm-8.3-2019.03-x86_64-aarch64-elf.tar.xz
```

4.1.2.2 CLANG / LLVM Compiler

The Clang compiler is only in early stage support.

The Clang 8.0.0 is the C compiler of the LLVM Compiler Infrastructure. It includes the compiler tools only. The libraries and headers need to be taken from GCC compiler. The Clang compiler can be obtained from this website: <http://releases.llvm.org/8.0.0/>. Clang toolchain is not yet supported for AArch64.

You need to first install the GCC toolchain and then untar the archive inside the chosen directory.

1. Download the archive

Trustonic validated with clang+llvm-6.0.0-x86_64-linux-gnu-ubuntu-14.04.tar.xz

2. Untar the archive in the current folder

```
tar xf clang+llvm-8.0.0-x86_64-linux-gnu-ubuntu-14.04
```

3. Modify setup.sh to point to the paths

4.1.3 Client Applications

4.1.3.1 Android SDK / NDK

- Download and install the Android SDK suitable for your development platform from <http://developer.android.com/sdk/index.html>
 - Add the Android SDK platform as described in <http://developer.android.com/sdk/installing/adding-packages.html>
 - For Windows add the USB driver as described in <http://developer.android.com/sdk/win-usb.html>.
 - In case you need to connect to a non-standard development board, the version provided with the platform must be used, so that ADB is able to recognize the ID of the device.
- Download the Android NDK (version r8 or above) suitable for your development platform from <http://developer.android.com/tools/sdk/ndk/index.html> and install it (means uncompressing into a directory on your computer).

Hint: if you are using Ubuntu 64-bit OS, make sure that the ia32-libs are installed. See <http://packages.ubuntu.com/de/lucid/ia32-libs>

Otherwise building the CA together with NDK may not work.

4.1.3.2 Using ADB

The Android Debug Bridge (ADB) is required to connect a host PC to the Android device. It is part of the Android SDK.

On Linux set your path variable to point to the folder with the ADB binary:

```
PATH="$PATH:<path-to-SDK>/platform-tools"
```

4.1.3.3 GCC / Linaro Compiler

The Linux toolchain used to compile your Client Application may depend on Linux distribution used (arm-gnu-linux? Poky...). It is recommended to simply use the same toolchain than the one used to build application on your targeted platform.

4.2 QUICK START GUIDE

This chapter contains a 'quick start guide' to SDK.

Please read also the chapters:

- Compiling and Testing Trusted Applications
- Compiling and Testing Client Applications

They hold more detailed information and might answer your questions.

4.2.1 File structure

The SDK package provides everything you need for TA and CA development under Linux:

- ◀ Documentation\
Documentation including API references and this guide.
- ◀ Samples\
Sample projects to get started quickly.
- ◀ Tools\
Binaries/APK to show up information like SUID, ProductId.
- ◀ t-sdk\
Contains header files and libraries to compile TAs and CAs.
- ◀ \
 - index.html
Documentation entry point to get started using the SDK.
 - setup.sh
Set the paths to your toolchains there.
 - setup.bat
Set the paths in order to build Windows Client Applications.

4.2.2 Setup instructions for Android

Find below the setup and installation guide of SDK to create sample Trusted Applications, CAs and apps.

- ◀ update paths in

```
~/workspace/t-base-XXX/setup.sh
```

to point to your local toolchains. This file setup.sh is in the top folder of the package. It is called by the individual components build.sh files to get the toolchain paths and the paths to the different components required by some Makefiles included in the release.

1. In order to build the Client Applications set COMP_PATH_AndroidNdk to your Android NDK installation (r8 or later).
2. In order to build 64-bit version of the Trusted Applications set CROSS_GCC64_PATH and CROSS_GCC64_PATH_BIN to your extracted Linaro GCC directory (aarch64-elf)
Example: /opt/toolchain/gcc-arm-8.3-2019.03-x86_64-aarch64-elf
3. If you plan to build 32-bit version of the Trusted Applications set CROSS_GCC_PATH and CROSS_GCC_PATH_BIN to your extracted Linaro GCC directory (arm-eabi)
Example: /opt/toolchain/gcc-arm-8.3-2019.03-x86_64-arm-eabi

- ◀ cross check that the paths are well set by running setup.sh:

```
./setup.sh
```

4.2.3 Setup instruction for Linux

Find below the setup and installation guide of SDK to create sample Trusted Applications, CAs and apps.

- ◀ update paths in

```
~/workspace/t-base-XXX/setup.sh
```

to point to your local toolchains. This file setup.sh is in the top folder of the package. It is called by the individual components build.sh files to get the toolchain paths and the paths to the different components required by some Makefiles included in the release. The toolchain value for Linux project must be GNU_EABI.

Hint: you need to install on your PC cmake and pkg-config.

Otherwise building the apps together for linux target may not work.

1. In order to build the Client Applications, set CROSS_GCC_PATH to your extracted Linaro GCC directory
Example: /opt/toolchain/gcc-linaro-4.9-2016.02-x86_64_arm-linux-gnueabi/hf/
2. In order to build 64-bit version of the Trusted Applications set CROSS_GCC64_PATH and CROSS_GCC64_PATH_BIN to your extracted Linaro GCC directory (aarch64-elf)
Example: /opt/toolchain/gcc-linaro-7.2.1-2017.11-x86_64_aarch64-elf
3. If you plan to build 32-bit version of the Trusted Applications set CROSS_GCC_PATH and CROSS_GCC_PATH_BIN to your extracted Linaro GCC directory (arm-eabi)
Example: /opt/toolchain/gcc-linaro-7.1.1-2017.08-x86_64_arm-eabi

- ◀ cross check that the paths are well set by running setup.sh:

```
./setup.sh
```

4.2.4 Creating your first secure application

Now you can use the Rot13-GP SDK -Sample to create your own Trusted Application and Client Application using the GlobalPlatform APIs.

4.2.4.1 Trusted Application (TA)

- ◀ update the demo makefile.mk and build.sh for your own TA
- ◀ for printing debug output in your Trusted Applications, use functions in tee_internal_api.h (TEE_LogvPrintf TEE_LogPrintf TEE_DbgPrintf TEE_DbgvPrintf).
- ◀ call the build script

```
./build.sh
```

- ◀ use the same build options as for TAs following the legacy API.

4.2.4.2 Client Application (CA)

- ◀ update the demo Android.mk and build.sh for your own CA
- ◀ to get the CA build running, you have to export additional the path to the output folder of your TA

```
export COMP_PATH_<yourTAName>=  
${COMP_PATH_ROOT}/../projects/<yourProject>/<yourTADir>/O  
ut
```

- ◀ use the same build options as for CAs following the legacy API.

4.2.5 Running your first application

You can also run secure applications following the GlobalPlatform scheme. Secure applications following the GlobalPlatform scheme can be either System-TAs or SP-TAs.

4.2.5.1 On a Development Board (running as System-TA or SP-TA)

To execute your Trusted Application on a Development Board it has to be compiled either as System Trusted Application and signed by the dummy OEMs RSA private key or as Service Provider Trusted Application and signed with the development key provided in the SDK.

Use a signed GP System Trusted Application:

1. Use the RSA System TA sample private key that comes with the SDK
(t-sdk-rXXX\Samples\Rot13_GP\TASampleRot13\
Locals\Build\pairVendorTltSig.pem)
2. Compile your Trusted Application as a System TA (adapt makefile.mk)

```
GP_ENTRYPOINTS := Y  
TA_UUID := 08010000000000000000000000000000  
TA_SERVICE_TYPE := 3 # System Trusted Application
```


4.2.6.1 Trusted Application (TA)

- ◀ update the demo makefile.mk and build.sh for your own TA
- ◀ for printing debug output in your Trusted Applications, use functions in tlApiLogging.h (tlApiLogvPrintf tlApiLogPrintf tIDbgPrintf tIDbgvPrintf).
- ◀ call the build script

```
./build.sh #default values
```

When a Trusted Application is built using this command, it will use the default build options and will compile a Trusted Application for ARM_V8A_AARCH64 in Debug with the GNU compiler.

- ◀ it is possible to override the default values of those build options with the following environment variables:

MODE can be Debug* or Release

TOOLCHAIN can be GNU* or CLANG

PLATFORM can be ARM_V8A_AARCH64* or ARM_V7A_STD

The script should be called this way:

```
MODE=Release TOOLCHAIN=CLANG PLATFORM=ARM_V7A_STD
./build.sh
```

4.2.6.2 Client Application (CA)

- ◀ update the demo Android.mk and build.sh for your own CA
- ◀ to get the CA build running, you have to export additional the path to the output folder of your TA

```
export COMP_PATH_<yourTAName>=
${COMP_PATH_ROOT}/../projects/<yourProject>/<yourTADir>/O
ut
```

- ◀ call the build script

```
./build.sh
```

4.2.6.3 Android SDK

- open the Android SDK eclipse and import the android-app project.properties
- build the app

4.2.7 Running a secure application using legacy API

If you have a Kinibi enabled device already, you can try to run your sample CA and TA.

4.2.7.1 On a Commercial Device

System Trusted Applications cannot be loaded on commercial devices for testing (require to be signed by the System TA signing key, usually Device OEM Keypair controlled)

4.2.7.2 On a Development Board (running as System-TA)

To execute your Trusted Application on a Development Board it has to be compiled as System Trusted Application and signed by the dummy OEMs RSA private key and signed with the development key provided in the SDK.

Hint: Kinibi images are available for some development platforms / boards.
See <https://trustonic.zendesk.com> for details or contact support@trustonic.com.

Use a signed System Trusted Application:

1. Use the RSA System TA sample private key that comes with the SDK
(t-sdk-rXXX\Samples\Sha256\TISampleSha256\
Locals\Build\pairVendorTltSig.pem)
2. Compile your Trusted Application as a System TA (adapt makefile.mk)

```
TA_UUID := 06010000000000000000000000000000
TA_SERVICE_TYPE := 3 # System Trusted Application
TA_KEYFILE := <your path>/pairVendorTltSig.pem
```

3. load both modules (from TA/CA out-folders) onto your device

```
adb push 06010000000000000000000000000000.tlbin data/misc/mcRegistry/
adb push <myCAname> data/app/
```

Notice that prior to Android Nougat, the path of the registry was "/data/app/mcRegistry".

4. start your CA

```
adb shell data/app/<myCAname>
```

Use a SP Trusted Application

See 0 for more details on how to use SP-PA and provisioning of SP-TA.

4.3 COMPILING AND TESTING TRUSTED APPLICATIONS

The following section describes what you need to get started writing a first Trusted Application, compiling it and loading it to the device.

Kinibi's main feature, for you as a developer, is that it can load additional code during run-time. In detail, it allows writing small programs which get pushed to the device and started as a Kinibi process once they are needed.

Developing a Trusted Application is similar to developing a C process, except for the fact that the Trusted Application is running in a separate environment. As described in Chapter 3.3.1, Trusted Application and Client Applications, inter world communication is done using sessions and invoking commands with memory references. See the next section for a basic example of these IPC mechanisms.

4.3.1 Build Environment

For creating new Trusted Applications you can start with one of the existing Samples in the SDK, which has the following layout:

- Locals Holds everything that belongs to the Trusted Application.
- Build Holds the Trusted Application build script. A build is started via build.sh. It sets the build environment and starts the build process.
- Code Holds the Trusted Application code. Public header files should be placed in a subfolder called public. The compilation parameters can be set in makefile.mk.
- Out Holds the generated Trusted Application binary.
- Bin Generated binaries
- Public Exported headers

4.3.2 Compiling and signing a Trusted Application

Trusted Applications are being directly run on the target platform without any interpretation. Thus, they can be written in assembly code, C or in any other high level language where a compiler exists.

Using the provided SDK, Trusted Applications need to be written in C according to the C99 standard.

The GNU Compiler is the recommended tool chain for compiling and linking. Clang toolchain is not yet supported for AArch64.

Prerequisites:

- ◀ Select a new UUID for your Trusted Application. On a development environment, the UUID can be selected freely according to RFC 4122 from the SDK Development Trusted Application UUIDs listed in the table below. Please use MobiConvert to generate proper UUID, see section MobiConvert Manual.
- ◀ Generate an AES256 key (32 byte) and enter it in the <Key> tag in key.xml in ASCII hexadecimal notation.
- ◀ See the section MobiConvert Manual for more information about their format.

For use of SDK development targets following UUIDs are predefined and must be used:

Name	UUID	Description
Sample ROT13	0401-0000-0000-0000-0000-0000-0000-0000	sample code
Sample SHA256	0601-0000-0000-0000-0000-0000-0000-0000	sample code
TIAes	0702-0000-0000-0000-0000-0000-0000-0000	sample code
TIRsa	0704-0000-0000-0000-0000-0000-0000-0000	sample code
TIFloat	6670-0000-0000-0000-0000-0000-0000-0000	sample code
TICryptoCatalog	0713-0000-0000-0000-0000-0000-0000-0000	sample code
Sample Pinpad	0789-0000-0000-0000-0000-0000-0000-0000	sample code
Sample TIPlay	0712-0000-0000-0000-0000-0000-0000-0000	sample code
SDK Development	0801-0000-0000-0000-0000-0000-0000-0000 0802-0000-0000-0000-0000-0000-0000-0000 0803-0000-0000-0000-0000-0000-0000-0000 0804-0000-0000-0000-0000-0000-0000-0000 0805-0000-0000-0000-0000-0000-0000-0000 0806-0000-0000-0000-0000-0000-0000-0000 0807-0000-0000-0000-0000-0000-0000-0000 0808-0000-0000-0000-0000-0000-0000-0000 0809-0000-0000-0000-0000-0000-0000-0000 080A-0000-0000-0000-0000-0000-0000-0000 080B-0000-0000-0000-0000-0000-0000-0000 080C-0000-0000-0000-0000-0000-0000-0000 080D-0000-0000-0000-0000-0000-0000-0000 080E-0000-0000-0000-0000-0000-0000-0000 080F-0000-0000-0000-0000-0000-0000-0000 0810-0000-0000-0000-0000-0000-0000-0000	for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only for testing only reserved reserved reserved reserved

Table 4: SDK UUIDs for Development targets

Developers must use for development target a UUID among the 12 first UUIDs used for testing from 0801-0000-0000-0000-0000-0000-0000-0000 until 0810-0000-0000-0000-0000-0000-0000-0000.

The 4 last UUIDs of the table are reserved for Trustonic and should not be used.

Hint: For production use you should use a key with sufficient entropy and a UUID which can be used throughout the life time of the TA.

Using the SDK lets you compile and sign a Trusted Application by using the Build process included in the sample Trusted Application structure, in two easy steps.

1. Write your parameters in `makefile.mk`, which can be found in the Code folder of the Trusted Application (for more information about the possible parameters for MobiConvert, see section MobiConvert Manual).
2. Start the build process by running `build.sh`, which is in the build folder of the sample Trusted Application (in case you did not do so before, you should adapt `setup.sh` to your environment).

Note:

The `TA_KEYFILE` parameter of `makefile` shall not be changed as Trusted Application containers are preinstalled.

4.3.3 Running a Trusted Application in a Development Environment

Whenever a CA is requesting a session to a Trusted Application specified by a UUID, the Kinibi Driver looks for a Trusted Application with this UUID on the file system and loads it. Therefore, the following steps are necessary to run a Trusted Application.

1. Make sure that the output binary file of the Trusted Application build process is named after your selected UUID. For example, if the UUID is `{1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8}` the file named becomes `"01020304050607080102030405060708.tlbin"`.
2. Using ADB, push the Trusted Application to the `/data/misc/mcRegistry` folder on the device, so that the Trusted Application can be found and loaded by the Kinibi Driver.

If `mcOpenTrustlet()` is used, the Trusted Application does not have to be in `/data/misc/mcRegistry`.

3. Develop a CA that uses the Trusted Application (see section 7).

4.3.4 Debugging a Trusted Application

Kinibi is a secure runtime environment and thus there are limited debugging options on the target platforms. In fact, debugging the SWd may not be allowed at all on end user platforms.


```

<6>[ 24.805570] Trustonic TEE: 101|r4=0x00100000, r5=0x00000027, r6=0x00000000, r7=0x00000000
<6>[ 24.813793] Trustonic TEE: 101|r8=0x0000401c, r9=0x00000000, r10=0x00000001, r11=0x00000000
<6>[ 24.822042] Trustonic TEE: 101|r12=0x03d0113b, sp=0x0000c348, lr=0x0000164b, pc=0x000010b4
<6>[ 24.830312] Trustonic TEE: 101|UUID=05010000000000000000000000000000

<6>[ 27.859405] Trustonic TEE: 101|EXCEPTION in 501, thread 0x10005, cpsr=0x60000030 [USER,TCZ]
<6>[ 27.875738] Trustonic TEE: 101|cause=0x4 (TRAP_UNDEF_INSTR), meta=0x164f
<6>[ 27.882536] Trustonic TEE: 101|IFSR=0x205 [TranslationL1,LPAE], AIFSR=0, IFAR=0xbe000e00 [valid]
<6>[ 27.891279] Trustonic TEE: 101|r0=0x00000000, r1=0x00000000, r2=0x00000004, r3=0x00000000
<6>[ 27.900036] Trustonic TEE: 101|r4=0x00100000, r5=0x00000027, r6=0x00000000, r7=0x00000000
<6>[ 27.907673] Trustonic TEE: 101|r8=0x0000401c, r9=0x00000000, r10=0x00000001, r11=0x00000000
<6>[ 27.919892] Trustonic TEE: 101|r12=0x03d0113b, sp=0x0000c348, lr=0x00001625, pc=0x0000164e
<6>[ 27.930573] Trustonic TEE: 101|UUID=05010000000000000000000000000000

<6>[ 30.890947] Trustonic TEE: 101|EXCEPTION in 501, thread 0x10005, cpsr=0x20000010 [USER,C]
<6>[ 30.897783] Trustonic TEE: 101|cause=0x7 (TRAP_INSTR_FETCH), meta=0x30
<6>[ 30.904135] Trustonic TEE: 101|IFSR=0x207 [TranslationL3,LPAE], AIFSR=0, IFAR=0x30 [valid]
<6>[ 30.912314] Trustonic TEE: 101|r0=0x00000023, r1=0x00000032, r2=0x00000004, r3=0x00000000
<6>[ 30.920530] Trustonic TEE: 101|r4=0x00100000, r5=0x00000027, r6=0x00000000, r7=0x00000000
<6>[ 30.928761] Trustonic TEE: 101|r8=0x0000401c, r9=0x00000000, r10=0x00000001, r11=0x00000000
<6>[ 30.936972] Trustonic TEE: 101|r12=0x03d0113b, sp=0x0000c348, lr=0x0000165b, pc=0x00000030
<6>[ 30.945317] Trustonic TEE: 101|UUID=05010000000000000000000000000000

```

Kinibi-410a, with ASLR offsets:

```

<6>[ 188.134599] Trustonic TEE: 101(3)|FAULT in 701, thread 0x10007, UUID=cf27d291-427b-535c-a33d-80c48bf3753b
<6>[ 188.144215] Trustonic TEE: 101(3)|EXCH: trapType=0x2 (TRAP_SEGMENTATION), trapData=0x50001c, cpsr=0x60000030 [USER,TCZ]
<6>[ 188.155088] Trustonic TEE: 101(3)|EXCH: DataFault: DFSR=0x206 [TranslationL2,read], ADFSr=0, DFAR=0x50001c [valid]
<6>[ 188.165482] Trustonic TEE: 101(3)| r0=0x00000000, r1=0x00000000, r2=0x00000033, r3=0x00000000,
<6>[ 188.174417] Trustonic TEE: 101(3)| r4=0x00500018, r5=0x00001777, r6=0x001de1a0, r7=0x001d2488,
<6>[ 188.183328] Trustonic TEE: 101(3)| r8=0x00300010, r9=0x00300008, r10=0x00500008, r11=0x001c87f9,
<6>[ 188.192253] Trustonic TEE: 101(3)|r12=0x00500000, sp=0x001d2300, lr=0x00300010, pc=0x001ca51c
<6>[ 188.201079] Trustonic TEE: 101(3)|EXCH: ASLR offsets: task=0x001c7000, mcLib=0x07f1b000

```

From this trace we can see:

- < Kinibi was executing on core 3, see (3) in the traces
- < A fault happened in TA with session ID 701, corresponding to thread ID 10007.
- < The UUID of the TA.
- < The crash type, crash address and register dump.
- < The PC is at 0x001ca51c
- < The TA code was relocated to address 0x001c7000.
- < The McLib was relocated to address 0x07f1b000.

From this information we can deduce:

- < PC was in TA code, not in McLib (PC smaller than mcLib offset)
- < Original PC is 351c = 0x001ca51c - 0x001c7000 = PC_DUMP - TASK_OFFSET
- < LR seems invalid, in other cases LR should be in TA code or McLib code
- < If PC or LR are in McLib code, then McLib offset needs to be subtracted.

4.3.4.4 Logging

Trusted Applications can use printf-like logging. The log is written to the Linux kernel log and can be retrieved via adb shell dmesg or using the serial port of your Arndale board. The function `tlApiLogPrintf(...)` writes a formatted string to the Linux log. The helper function `tlDbgPrintf(...)` can be used for logs that are only included in the TA binary in debug builds.

Apart from that, TAs can return error codes and error strings in the parameters of a command.

4.3.4.5 Debugging with ASLR

When debugging an application, it can be useful to temporarily disable the ASLR. To do so, the developer can specify in the TA makefile:

```
TA_PIE := NON_PIE
```

However, in any case, MLib code is always loaded to randomized address.

Another option could be to run and debug the same application on a Kinibi version without ASLR.

Beware of so-called information-leaks:

When using printf-based debugging, it is often useful to print addresses with %p. Note that this can help an attacker to defeat the ASLR of the TA. Printing addresses using %p or %x should be done only in the debug version of the TA, using tIDbgPrintf and the debug version must not be used in production.

For diagnostic purposes, developers often leave traces in the TA also in the production version, for example by using tApiLogPrintf. A developer must not use %p inside tApiLogPrintf or in any other way output an address of the TA code or derived address like TCI or WSM in a production version of a TA.

4.3.4.6 Development Platforms

Special development versions of the target platform and generic development platforms exist. They allow full JTAG access, so that any code can be debugged in detail. However, halting the system or manipulating certain registers may change the runtime behavior of the system significantly. This may even lead to an unstable system due to timing issues.

Furthermore, Kinibi is a multi-tasking system which uses the MMU. Every task runs in a separate virtual address space. JTAG access usually does not bypass the MMU, thus breakpoints are set on virtual addresses only.

Usually, a debugger is not aware of different tasks and address spaces by default. Thus manual checks of the current task are necessary each time a breakpoint is hit.

Hint: Please refer to the manual of your debugger for details about process- or task-aware debugging.

4.4 COMPILING AND TESTING CLIENT APPLICATIONS

The following chapter describes how to set up your development PC appropriately and gives you the basis of CA development using the Android NDK tool chain.

Building a CA depends a lot on the platform you are developing for.

In the case of Android, the Native Development Kit (NDK) provided by Google is best suitable to compile and build executable, shared- and static libraries.

Although it is possible to directly use the TEE Client API from your Android App, allowing you to write your CA logic purely in Java code, this chapter focuses on developing a CA in C code, so you are able to generate portable code for different platforms.

4.4.1 Client Application Structure

The Client Application is responsible for connecting the Android App with the Trusted Application. It provides the interface to the Trusted Application functionality on the Java layer.

Therefore a CA is composed of the following components:

- ◀ One or more generic CA methods using the TEE Client API to communicate with the Trusted Application in the SWd.
- ◀ A shared library written in C/C++ wrapping the generic CA methods and providing their functionality to the Java layer.
- ◀ A Java class including the shared library and calling the library methods using Java Native Interface (JNI).
- ◀ An Android App including the Java class and the shared library.

The subsequent sections will provide a more detailed view on how to create these components.

4.4.2 Android Compilation

4.4.2.1 Compile a CA with the Android NDK

The Android NDK is used to build the NWd's native code, namely the CA consisting of the shared library and an (optional) binary for testing purposes.

4.4.2.1.1 Android NDK Overview

The NDK is a stripped down version of the Android-Kernel tree. It comes equipped with the following components:

- ◀ The ARM cross compiling development tool chain
- ◀ The stable Android API's for native code development
- ◀ Android's Bionic C library, providing lightweight wrappers around kernel facilities
- ◀ A static library of the GNU libstdc++

Hint: Bionic is not binary-compatible with any other Linux C library. This means that you cannot build something against the GNU C Library headers and expect it to dynamically link properly to Bionic later.

More detailed information can be found in the "docs" folder of the NDK.

The NDK build system is based on Make, but encapsulates the generic build settings internally. It has been designed to meet the following goals:

1. Simple build files to describe the project. Basically you only need to list the C/C++ files and include directories.
2. The Android tool chain deals with many important details:
 - ◀ Proper tool chain selection and invocation (compiler + linker flags)
 - ◀ Android API level / platform / project support
 - ◀ Multi-ABI code generation
 - ◀ Native debugging setup

You need to define two Make files, which may reside in your projects "code" base folder:

- ◀ Application.mk. This is the main build file defining the modules required to build the project, which is NDK's naming for Make targets. You may define global variables here and need to point to the second Makefile.
- ◀ Android.mk. Here you need to define what each module is depending on:
 - Path to header file folders
 - Path to source files
 - (external) libraries
 - Type of output (at the end of a module's section):
 - Shared libraries:

```
include $(BUILD_SHARED_LIBRARY)
```

- Static libraries:

```
include $(BUILD_STATIC_LIBRARY)
```

- Executables:

```
include $(BUILD_EXECUTABLE)
```

See the best practice samples or the NDK's "docs" folder for more details on writing Make files.

Once you defined the Make files, the build can be invoked calling:

```
$ <path-to-ndk>/ndk-build \
  -B \
  -C <path-to-project> \
  NDK_DEBUG=1 \
  NDK_PROJECT_PATH=<path-to-project> \
  NDK_APPLICATION_MK=Application.mk \
  NDK_MODULE_PATH=<path-to-libMcClient.so> \
  NDK_APP_OUT=<path-to-output-dir> \
  APP_BUILD_SCRIPT=Android.mk
```

For a detailed description of the parameters, have a look at documentation.html in your NDK's base folder.

If you need to debug your build setup, add "V=1" as parameter for verbose output.

After a successful build the binaries can be found in `<path-to-output-dir>/local/armeabi/`.

If you defined an executable as one of your build targets, you can now push it from there to the device's `/data/app/` folder and execute it using ADB.

4.4.2.2 Android APK

This guide does not go into details on how to turn your application into an Android APK, and our Product Package does not have support for Java APIs.

This being said, there are some points to consider when writing an APK, especially with regards to Authentication. Indeed, release 310C makes it possible for phone makers to add an authentication mechanism that will filter which applications can connect to the Kinibi TEE.

As it stands today, the Authentication requires APKs to provide two files named `license.bin` and `issuercert.bin` in their assets directory. These two files are provided by Trustonic using a specific channel which we will not elaborate on here. It also requires a specific permission to be added to the `AndroidManifest.xml` file in the APK, like so:

```
<manifest (...)>
    <uses permission android:name="com.trustonic.license" />
</manifest>
```

4.4.3 Linux Compilation

CMake is used for the compilation of the Client Applications. Only Native C or C++ is supported.

4.4.3.1 CMake overview (<https://en.wikipedia.org/wiki/CMake>)

CMake is cross-platform free and open-source software for managing the build process of software using a compiler-independent method. It is designed to support directory hierarchies and applications that depend on multiple libraries.

The build process with CMake takes place in two stages. First, standard build files are created from configuration files. Then the platform's native build tools are used for the actual building.

Each component contains a `CMakeLists.txt` file in every directory that controls the build process. The `CMakeLists.txt` file has one or more commands in the form `COMMAND (args...)`, with `COMMAND` representing the name of each command and `args` the list of arguments, each separated by white space. While there are many built-in rules for compiling the software libraries (static and dynamic) and executables, there are also provisions for custom build rules. Some build dependencies can be determined automatically. Advanced users can also create and incorporate additional makefile generators to support their specific compiler and OS needs.

In `CmakeLists.txt` file you find the rules to build your CA, example of `TlcAes` application:

- Define include directories and files:

```
# By scanning folder :
file(GLOB_RECURSE LIST_HEADERS_TOP *.h)

foreach (_headerFile ${LIST_HEADERS_TOP})
    get_filename_component(_dir ${_headerFile} PATH)
    list(APPEND INCLUDE_DIRS ${_dir})
```

```
endforeach

include_directories(
  ${INCLUDE_DIRS}
  ${COMMON_DEP_INCLUDE_DIRS}
  ${COMP_PATH_TlAes}/Public
  ${COMP_PATH_TlSdk}/Public/MobiCore/inc
)
```

- Define your source code by scanning a specific folder:

```
file(GLOB RECURSE LIST_CPP_TOP "*.cpp")
SET(SRCS
  ${LIST_CPP_TOP}
)
```

-
- Create a binary based on source code :

```
ADD_EXECUTABLE(tlcSampleAes ${SRCS})
```

- Create a Shared library:

```
ADD_LIBRARY(TlcSampleAes SHARED ${SRCS} )
```

- Create a Static library:

```
ADD_LIBRARY(TlcSampleAes STATIC ${SRCS} )
```

- Define a install Path:

```
INSTALL(TARGETS tlcSampleAes DESTINATION bin)
INSTALL(TARGETS TlcSampleAes DESTINATION lib)
```

- Link option:

```
TARGET_LINK_LIBRARIES(TlcSampleAes
  ${COMMON_DEP_LIBRARIES}
  ${COMMON_DEP_LDFLAGS}
)
```

See [cmake website](#) for more details on writing CMakeLists.txt.

In the build folder of the sample Client Application, you can find a build.sh which uses cmake for the build.

```
cmake \
  $BASE/Locals/Code/ \
  -DCMAKE_INSTALL_PREFIX=$BASE/$OUT_DIR \
  -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake \
  -DCOMMON_DEP_INCLUDE_DIRS=$COMP_PATH_MobiCoreClientLib_module/Public/ \
  -
DCOMMON_DEP_LIBRARIES=$COMP_PATH_MobiCoreClientLib_module/Bin/armgnuabi-
v7a/Debug/ \
  -DCMAKE_BUILD_TYPE=$MODE \
  -Wno-dev

make -j $CORES

make install
```

After a successful build the binaries can be found in <cmake-install-dir>/ armgnueabi-v7a/.

If you defined an executable as one of your build targets, you can now push it from there to the device's `/usr/bin/` folder and execute it using ADB.

4.4.4 Run your Client Application

4.4.4.1 Connect to the Device via USB

Connect your development device via USB and check with ADB if you can see the device:

```
$ adb devices
List of devices attached
3532C8CB5C0A00EC device
```

If you don't see a device ID, but:

```
List of devices attached
????????????? no permissions
```

You are missing permissions. Restart ADB with root privileges:

```
$ adb kill-server && sudo <path_to_SDK>/platform-tools/adb devices
```

4.4.4.2 Connect to the Device via Ethernet

Depending on your device, they may be required each time you restart the device. In case you connect the device directly to the host via USB, you can skip to the last point.

1. Obtain the IP address of the target. It can be found by adding "ip=dhcp" in the bootargs, which will obtain and print the IP automatically during boot. Alternatively, you can enable the Ethernet port and obtain an IP address via DHCP running the following commands in a terminal connected to the device once the platform has booted:

```
# netcfg eth0 up
# netcfg eth0 dhcp
```

2. Using the command below you can verify that the board did obtain an IP address:

```
# netcfg
```

3. On the host, perform the following (every time you reboot the device or create a new connection):

```
$ adb kill-server
$ adb connect <ip-address-of-device>:5555
```

4. Ensure that connection is working by running:

```
$ adb shell
```

You should see a command prompt of the target on your host.

Verify this by running `ps` or similar commands.

Exit the ADB shell by typing exit.

Other useful ADB commands are:

\$ adb logcat	displays logging output of Android programs.
\$ adb devices	shows available connected devices
\$ adb shell	connects a shell to the device
\$ adb shell <command>	executes a command on the device
\$ adb connect <ip>:<port>	connects to a network connected device. Default port is 5555
\$ adb kill-server	restarts adb server (on client) in case of problems (e.g. when getting "error: device offline")
\$ adb push/pull <src> <dest>	uploads/downloads files to/from the device
\$ adb install <file.apk>	installs an application to the device

Table 5: ADB commands

For a detailed description of all ADB commands see <http://developer.android.com/tools/help/adb.html>

4.4.4.3 Upload and Test

In order to test your CA make sure the Kinibi Driver Kernel module is loaded and the Kinibi Driver daemon is running:

1. If not done yet, set up an ADB connection to the device:

```
$ adb connect <ip>:5555
```

Now you can upload and test your Trusted Application and CA as described below:

2. Upload the Trusted Application to the device:

```
# For Android:
$ adb push <path-to-tl>/<your-tl> /data/misc/mcRegistry
# For Linux
$ adb push <path-to-tl>/<your-tl> /usr/share/trustonic_tee/registry
```

3. Upload and run the CA:

```
# For Android
$ adb push <path-to-tlc>/<your-tlc> /data/app/
$ adb shell /data/app/<your-tlc>
# For Linux
$ adb push <path-to-tlc>/<your-tlc> /usr/bin/
$ adb shell /usr/bin/<your-tlc>
```

Depending on your CA you should now see your debugging messages in the command shell.

4.4.4.4 Shared Libraries on the Device

By default the Android linker will search in `/system/lib` for shared libraries. In order to provide your own libraries for applications you need either to make sure that you can write to this folder (root the device and remount the partition) or extend the `LD_LIBRARY_PATH` variable with a directory with sufficient access rights (e.g. `/data/app`).

The following command extends the library path:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/data/app
```

You need to set this each time the platform is restarted or the ADB shell is reconnected.

By default on Linux, libs are located in `/usr/lib` (default Linux path). In order to provide your own libraries for applications you need either to make sure that you can write to this folder (root the device and remount the partition) or extend the `LD_LIBRARY_PATH` variable with a directory with sufficient access rights (e.g. `/usr/share/trustonic_tee`).

The following command extends the library path:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: /usr/share/trustonic_tee
```

4.4.4.5 Client Application usage from within your Android App

For the Java part of your development we recommend using Eclipse in connection with the Android Development Tools plugin.

Please refer to <http://developer.android.com/guide/index.html> for more information on developing the Java part of an Android App.

Good places to start are also the tutorials and sample code available at <http://developer.android.com/training/index.html>.

For the Android App to utilize the CA's functionality it needs to import the shared library you created with the Android NDK. See the documentation and samples on <http://developer.android.com/tools/sdk/ndk/index.html> for a guideline on how to do that.

4.4.5 Debug

This section should give you an overview of all relevant debugging resources and strategies available on an Android device.

Below you can find a summary of some useful debugging commands for Linux system debugging to be executed from an Android shell:

Command	Description
dmesg	Displays Kernel debugging messages (kernel module output).
cat /proc/interrupts	Shows number of interrupts and their associated kernel module.
cat /proc/version	Displays the current Kernel version, when, from whom and which tool chain was used to build it.
ps (-t)	Prints current running processes (incl. threads if parameter "-t" is set) with their process ID (PID) and current program counter (PC) position.
cat /proc/"<PID>"/maps	Investigates memory mapping of a process.
cat /proc/meminfo	Shows current memory usage.
cat /proc/misc	Lists miscellaneous drivers registered on the miscellaneous major device.
lsmod	Lists all modules loaded into the kernel.
objdump -x <binary> grep NEEDED	List the required libraries of a binary or library. The libraries must be found under /system/lib directory on the device.

Table 6: Linux system debugging commands

4.4.5.1 Segmentation Faults

In case you see something like this on the command line:

```
[1] Segmentation fault /data/app/tlcSampleRot13
```

Have a look at the output of logcat, e.g.:

```
I/DEBUG ( 678): *** *** *** *** *** *** *** *** *** *** *** *** *** ***
*** *** ***
I/DEBUG ( 678): Build fingerprint:
'generic/generic/generic/:2.2/MASTER/eng.robert.20100629.090756:eng/test-
keys'
I/DEBUG ( 678): pid: 2071, tid: 2071 >>> /data/app/tlcSampleRot13<<<
I/DEBUG ( 678): signal 11 (SIGSEGV), fault addr 00000000
I/DEBUG ( 678): r0 0001a4e4 r1 be882ca8 r2 00000003 r3 00000000
I/DEBUG ( 678): r4 00018670 r5 40009008 r6 00000000 r7 00000000
I/DEBUG ( 678): r8 00000000 r9 00000000 10 00000000 fp 00000000
```

```
I/DEBUG ( 678): ip 00018708 sp be882ca0 lr 00012401 pc 00012404
cpsr 00000030
I/DEBUG ( 678): #00 pc 00012404
/system/data/app/tlcSampleRot13
I/DEBUG ( 678): #01 lr 00012401
/system/data/app/tlcSampleRot13
```

The interesting line is:

```
I/DEBUG ( 678): #00 pc 00012404
/system/data/app/tlcSampleRot13
```

It displays the binary/library which created the segfault and the content of the PC.

4.4.5.2 GDB

For debugging your C/C++ CA code, the GDB (Gnu Debugger) can be used. The setup is based on a gdbserver running on the device and an ARM aware GDB (arm-eabi-gdb) on the host machine.

While only the gdbserver and the arm-eabi-gdb binaries are required to debug native code on Android, you can also integrate debugging with Eclipse. The following section describes a method of debugging the NDK based applications from within Eclipse.

4.4.5.2.1 Debug stand-alone binaries

This section describes how to debug a stand-alone CA binary (without the Java app on top).

For full GDB debugging functionality use Android 2.3 or later as your debugging platform.

ADB and device configuration:

1. set up an ADB connection to the device:

```
$ adb connect <ip>:5555
```

2. forward port 5039 to port 5039 on the device:

```
$ adb forward tcp:5039 tcp:5039
```

3. start GDB server with the program to debug on the device:

```
$ adb shell gdbserver :5039 /path/to/program
```

A detailed description to the last command can be found on http://www.kandroid.org/online-pdk/guide/debugging_gdb.html

Eclipse configuration:

First of all, Eclipse needs to find gdb:

- ◀ Make sure the Android SDK tools and the NDK binary folder are in your system path. For Linux, add to `/home/<user>/.profile`:

```
PATH="$PATH:<path-to-NDK>/build/prebuilt/linux-x86/arm-eabi-4.4.0/bin/"
```

Configure a new C/C++ Application Debug configuration for your project.

Go to Debug-View, then Debug Button, then configure a new C/C++ Application Debug configuration for your project:

1. Main tab
 - a. C/C++ Application must point to your binary.
2. Select debugger
 - a. On the bottom: Using GDB (DSF) Standard Create Process Launcher → Select other
 - i. Use GDB (DSF) Remote system Process Launcher
 - ii. This will change existing tabs
3. Debugger tab
 - a. Stop on startup at: can be set to your entry method (e.g. "main")
 - b. In the subtab Main set GDB debugger to the arm-eabi-gdb
 - c. In the subtab Connection set TCP, localhost and 5039

You need to restart the gdbserver after each debug run.

4.5 INIT ENTRY POINT FOR TRUSTED APPLICATIONS

It is possible to define an optional `_init()` entry point which is called before the `tlMain()` or `TA_CreateEntryPoint()` entry point.

```
void _init(void)
{
    // do your startup tests
}
```

This entry point can be used, for example, to perform “power-on” tests. If the power-on tests fail, the Trusted Application can call the `tlApiExit()` function to stop the execution.

5 MOBI CONVERT MANUAL

MobiConvert is Trustonic's file conversion tool that creates various binary objects required for deployment of secure services. Its main purpose is to take a service executable image (ELF file of a Trusted Application or driver) as <in-file> and to create a Kinibi loadable service image as <out-file>. Depending on the service type, it is either signed with a private key contained in a PEM-format file or protected with a symmetric key stored in XML format file. On success, a converted loadable image is created and the exit code is 0.

Other simple modes of operation are the generation of a UUID, of a RSA key pair, to generate a UUID from a RSA key and to display file information of converted files. Other advanced modes are the encryption of System TAs, the re-signing of converted service images using an HSM and the editing of service properties.

5.1 HELP OUTPUT

```
MobiConvert V1.6 t-base-Hikey-Android-410a-v002-20180730_190511_53063
  Copyright (c) 2013-2018 TRUSTONIC LIMITED. All rights reserved.

usage: java -jar MobiConvert.jar [mode] [options]
  -?,--help                Show help for specified command, e.g.
                           --help -gk
  -encrypt,--encrypt       Encrypt a System TA or Driver
  -g,--genuuid4            Generate a version 4 UUID according to RFC
                           4122 and exit
  -gk,--genkey             Generate 2048-bits RSA key pair
  -gp,--gp_prop <mode> <args> Properties editor for GP TAs:
                           read/write/help
  -h,--header <file>     Show the header of the file given as
                           argument
  -pu,--printuuid         Print UUID generated from RSA key to
                           standard output
  -s,--servicetype <int> Convert and sign a TA of type:
                           1: Trusted Driver
                           2: Service Provider Trusted Application
                           3: System Trusted Application
  -sign,--signraw         Sign a tlbins/tabins/drbins (signed or
                           --nosign) using a key file or HSM

For more information, read the 'MobiConvert Manual' in the Developer's
Guide.
```

5.2 RSA KEY GENERATION

To encrypt Trusted Applications of the Service Provider type, a RSA key is required. You can generate a RSA key pair using MobiConvert:

```
java -jar MobiConvert.jar --genkey
```

Example:

```
The generated RSA key is:
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAg7vDwXYn5MElF5Ny+JeOH2DFY/JAcDNwG3xXT8n7jgmWg/Sc
ctYM/xu6x/QtKCKUquaV9BiRW8o1ZyPeMZ83+K1lpFXZl12gChejItKgyifVx03h
XsBY+G2rGGQH64kk2dkjtfnzRxUkTdlg5flMEjnd04HLi+Sm6iXjfsr1YkAl9TQ
+J0QZQ7T18RzCs7w2Pfu4QIrU2zFfeS1AG/en3U+Iiu278i54U4dZn3zJp1SFX0K
WIQiozCU0ZFv+xBm+xvzv3+kOpDgFmjeTvcV8n17Ywo6+Mu55IvsAAwQkFX2rMv4
rGyURaFtRS5gdHdCWSR2s42Ozi05WwYz1bk4mQIDAQABAOIBAQCcoqDfBSPQ3AcUW
2VWBdP48LMLOvHyYdKH2LHBnSPvHa/0pTNNVKIkNBcOzWkhFMUi7mB4oKQpHGcN
Wz19TFwg2tJGyZVvxaBEkJJjwa3tu5+FJCRg9NCr8rCkvKDWnhLV7B3ZO0fEGKxV
2EOwt9wQzknfzcoEcqPGsz0wKgK7qzuh8/6KmCfjLBgbxvkYZB+3NYLyNibzTx47
fhaCVqwdgbnUy4YQ2PIEEO+uNYIqBv7QQBWEClFrHd8/5JNEZAw4XUzXA0XOBqN0
P5jWZntfAqFfr1aQAOHs5fiGi2a4w40cg5+sJBeCrt1rWcpW8Zmm2KU6F0bNM1Q5
xhQ4Sx2xAoGBAN49lQ1Y2mhOW2bGZqjZ5xug6G4k/AJoQNdvFYaVMEMKImC94Pp6
jMVv5Z+abhVuPAoKmkRvChW02QKTI6DbK49x3vAwlnN7bEwo8L4PuW8OtRm08S+U
rza6iTujsnMVfEh9j/P/6ewfFJyXCOyMQxlNmQQkaIWk8d5mv1xZV2T1AoGBAMXS
FDeF5XMHP6RCYk8GL+A3HO8fGQUs7zAyPzZdAatTRmLgZw9gzldTe2rEuhE2XhiI
k8xHhymNC3Yj5GRQhVZRwD+tDzgt5qGbQmLsw2sKCb0XoundJagiofzZTj0WAVT+
fvY5iheRSofcRJLQFLA+lgXxu0KZ8rGXJMxBs96VAoGBAMumlObelC1W+Gzii/pY
y23G8pbUL1apYBnKgmG0V+hm5f/On9YH7O2Tz1CE/DGJNV1iP+FL+2rOsTmPybFC
hdVJ3Kgvbf7e7+uObKVN1XgOeyfWZllan4DASLctF35cBuqKnRpTvXERPhsMUDIr
ieUq9XgVQO6OqtFJSDwA5pPtAoGBALAGO08cggstDAhRucCvtLJC2FA+z7i3Py8X
xwWVcwLMWv1ozMv2TCWQd2WOIDNouVoDTeCcVT038FbzoStSKxOgMv12NPC8h1iO
GwiDvW/lwryr559J1VRDXPjtNj10k2jZ/IeEs8g81KEH80zgM0iQqFYpv4OIEVjN
MUU/wZnxAoGAHSnx3KM8x1D6orUkPqsxJWh9uUfW4VPYqxspJe1RRaAxTq12SjLX
kaBL094pUB5XlyVsc1eJe8Hb+E+VPdsBmTZ+SAP49L7Ax9YS3x1T/QhcrstREOW
zxSMu4peXV7AewV6RFD/1uX9Ga3ftr8bm199mN4urrHJnIMtw55wM+I=
-----END RSA PRIVATE KEY-----
```

You can copy this text into a file at Locals/Build named pairVendorTltSig.pem.

Note that if an HSM is to be used to sign a System TA, the key must be generated and held in a JCA-accessible key store. Please refer to the HSM vendor documentation to accomplish this task.

5.3 UUID GENERATION

5.3.1 For GlobalPlatform Trusted Applications

For GlobalPlatform TAs, the UUID is derived from a RSA key pair. Generate a key pair using MobiConvert and store the key under Locals/Build/pairUUIDKeyFile.pem.

Then call MobiConvert again to derive the UUID from this key file:

```
java -jar MobiConvert.jar --printuuid --uuidkeyfile  
Locals/Build/pairUUIDKeyFile.pem
```

Example:

```
java -jar MobiConvert.jar --printuuid --uuidkeyfile uuid.pem  
cb89f66f-f771-5614-a2b1-3f0bc3229e83
```

Remove the slashes and copy this UUID into your services Makefile as TA_UUID.

Note, you need to specify the UUID key file also when converting your Trusted Application.

5.3.2 For legacy services and drivers

To generate a RFC 4122 UUID for your new legacy service, use the following command-line:

```
java -jar MobiConvert.jar --genuuid4
```

Example:

```
The generated uuid (type 4) is  
ea846ca0bf9c425ebbe7b4600f5a3189
```

You can copy this UUID into your services Makefile as TA_UUID.

5.4 HEADER MODE

The header mode provides a way to display the content of the header of a service that was converted.

The name of the file provided as argument does not need to respect the "uuid.tlbin/drbin" format.

Usage :

```
java -jar MobiConvert.jar --header <file>
```

Example of using the Header mode:

```
java -jar MobiConvert.jar -h 04010000000000000000000000000000.tlbin
MODE = Show Trusted Application/Driver header and information
```

```
##### System Trustlet header: #####
Magic                = MCLF
Version              = 2.5
Flags                = New layout
Flags                = Position Independent TA
MemType              = external
Service type         = System Trustlet
NumberOfInstances    = 16
UUID                 = 04010000000000000000000000000000
DriverID             = 0
NumberOfThreads      = 1
CodeAddr             = 0x1000
CodeLength           = 26132
DataAddr             = 0x9000
DataLength           = 1056
BssLength            = 344
Entry Addr.          = 0x10C5
Interface Version    = 0.0
permittedHwCfg       = Encrypted Service
permittedSuid        = ffffffff0000000000000000
GPLLevel             = 1
attestationOffset    = 27232
CryptoParamsOffset   = 27711
#### Text header: ##
textHeaderVersion    = encrypted
textHeaderLen        = encrypted
InitHeapSize         = encrypted
MaxHeapSize          = encrypted
#### Cryptograms: ##
File size            = 27743
Executable size      = 27188
Crypto param size    = 32
System TA signature size = 523
```


-in,--signedbin <filename.tlbin>	Input file name, containing a converted (and optionally signed) service (alias to -b option)
-iv,--interfaceversion <major.minor>	Give the TA version in the format major.minor. The range of major and minor shall be each between 0-65535.
-k,--keyfile <file.pem>	Key file for the operation: .pem file for Key Pair to sign System TAs and Secure Drivers. .xml file for Symmetric Key for Service Provider TAs and System TA encryption
-m,--memtype <2>	2 = use only external memory, 1 = use only internal memory, 0 = prefer internal memory
-mh,--maxheapsize <integer>	Maximum size in bytes of the main heap
-n,--numberofthreads <integer>	Number of threads (1 for Trusted Applications, between 1 and 8 for Trusted Drivers)
--nosign	Convert without signing. Extension .raw is appended to output file <UUID.tlbin.raw>
-o,--output <UUID.tlbin>	Output file name UUID is 32 char [0-9a-fA-F], i.e. b230723bcfe75f8581122dc1f8e52f4d, 04010000000000000000000000000306 File Extension: .tlbin for legacy Trustlets, .tabin for GP Trusted Applications, .drbin for Secure Drivers
-rL,--relaxedlen	Optional, to skip checks on the size of the Trusted Application
-rO,--relaxedout	Optional, to skip checks on the output file name
-s,--servicetype <int>	Convert and sign a TA of type: 1: Trusted Driver 2: Service Provider Trusted Application 3: System Trusted Application
-sh,--initheapsize <integer>	Initial size in bytes of the main heap
-uk,--uuidkeyfile <file.pem>	RSA key pair that proves UUID ownership, default to Locals/Build/pairUUIDKeyFile.pem

For more information, read the 'MobiConvert Manual' in the Developer's Guide.

5.6 SIGN MODES FOR OEMS

An OEM can split the conversion step in two parts:

1. Convert .AXF file to .tlbin.raw file (--servicetype mode)
2. Sign .tlbin.raw file using a key into a .tlbin file (--sign mode)

The MobiConvert --sign mode supports two input formats:

1. .tlbin.raw: a raw tlbin file without signature
2. .tlbin: a tlbin file that is already signed. MobiConvert will remove the signature and re-sign)

The MobiConvert --sign mode supports two key formats:

1. Using an HSM-backed key store, using -hsm and related options
2. Using an XML key file, using -k option.

5.6.1 Signing using an HSM-backed key store

To sign a System TA using an HSM, the ELF file must first be converted into an unsigned Kinibi loadable format image.

Follow the example given above (section 5.5.4), supplying the additional "--nosign" argument to avoid doing the signature operation. Goal of this first action is to inject in Trusted App ELF binary the standard required parameters (needed flags, specify memory types...)

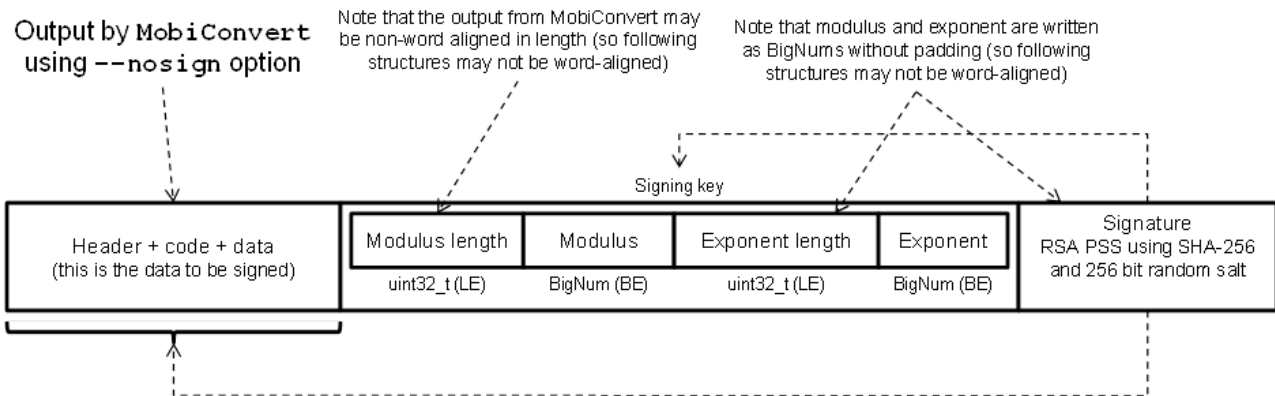


Figure 7 Illustrating the System TA signature structure

The resulting image must then have a signature appended, as illustrated in **Error! Reference source not found.**

To do that signature operation independently, MobiConvert provides a mode of operation that accepts the previously pre-configured/generated unsigned image.

The following syntax illustrates:

```
java -jar MobiConvert.jar
    -sign,--signraw --hsm
    -in,--rawfile <path to raw TA image>
    -o,--output <path to signed TA output file>
    --keystore <path to JCA key store>
    --keystorepassphrase <JCA key store passphrase>
    --keyname <name of signing key in JCA keystore>
```

```
--keystoretype <JCA key store type (default: JKS)>
--cryptoprovider <Java JCA provider class>
```

Vendors wishing to sign System TAs using alternate means (other than using `MobiConvert`) may simply follow the illustrated structure (**Error! Reference source not found.**) and generate the signed TA image accordingly.

5.6.2 Signing using a key file

The command is almost the same as when signing with an HSM.

The following syntax illustrates:

```
java -jar MobiConvert.jar
    -sign
    -in <UUID.tabin.raw>
    --output <UUID.tabin>
    --keyfile Locals/Build/pairVendorTltSig.pem
```

5.6.3 Re-signing a signed tbin

`MobiConvert` allows re-signing an already signed tbin. It works either with a keyfile or with HSM.

To select re-sign mode, the extension of input file should be `.tbin`, `.tabin` or `.drbin`.

The following syntax illustrates:

```
java -jar MobiConvert.jar
    -sign
    -in <UUID.tabin >
    --output <UUID.tabin>
    --keyfile Locals/Build/pairVendorTltSig.pem
```

5.6.4 Help output

```
MobiConvert V1.6 t-base-Hikey-Android-410a-v002-20180730_190511_53063
Copyright (c) 2013-2018 TRUSTONIC LIMITED. All rights reserved.

usage: java -jar MobiConvert.jar [mode] [options]
    --cryptoprovider <name>           Optional, name of the JCA
                                       provider, default to
                                       BouncyCastleProvider
    --hsm                             Use an HSM via a JCA key store,
                                       instead of a keyfile.
                                       It requires --keystore,
                                       --passphrase and --keyname,
    -in,--rawfile <filename.tlbin>    Input file name, containing a
                                       converted (and optionally signed)
                                       service
    -k,--keyfile <file.pem>          Key file for the operation:
                                       .pem file for Key Pair to sign
```

```

--keyname <name>           System TAs and Secure Drivers.
--keypassphrase <passphrase> .xml file for Symmetric Key for
                             Service Provider TAs and System TA
                             encryption
--keystore <filename>      Name of the key in the key store
--keystorepassphrase <passphrase> Optional, passphrase for the key
                             in the key store
--keystoretype <jks>       Name of the JCA key store
                             containing the signing key
-o,--output <UUID.tlbin>  Passphrase to unlock the JCA key
                             store
                             UUID is 32 char [0-9a-fA-F], i.e.
                             b230723bcfe75f8581122dc1f8e52f4d,
                             0401000000000000000000000000000306
                             File Extension:
                             .tlbin for legacy Trustlets,
                             .tabin for GP Trusted
                             Applications,
                             .drbin for Secure Drivers
--sign,--signraw          Sign a tlbin/tabin/drbin (signed
                             or --nosign) using a key file or
                             HSM

```

For more information, read the 'MobiConvert Manual' in the Developer's Guide.


```

--keyname <name>           Service Provider TAs and System TA
--keypassphrase <passphrase> encryption
                             Name of the key in the key store
--keystore <filename>      Optional, passphrase for the key
                             in the key store
                             Name of the JCA key store
--keystorepassphrase <passphrase> containing the signing key
                             Passphrase to unlock the JCA key
                             store
--keystoretype <jks>       Optional, type of the JCA key
                             store
-o,--output <UUID.tlbin>   Output file name
                             UUID is 32 char [0-9a-fA-F], i.e.
                             b230723bcfe75f8581122dc1f8e52f4d,
                             040100000000000000000000000000306
                             File Extension:
                             .tlbin for legacy Trustlets,
                             .tabin for GP Trusted
                             Applications,
                             .drbin for Secure Drivers

```

For more information, read the 'MobiConvert Manual' in the Developer's Guide.

5.8 PROPERTIES EDITOR MODE

GlobalPlatform allows a TA developer to specify properties for a TA. The MobiConvert gp_prop mode allows reading and writing GlobalPlatform TA configuration properties into TA AXF files from an XML file.

Usage :

```
java -jar MobiConvert.jar -gp_prop help
MobiConvert V1.5 Build by lukhan01@DE0015 09/20/16 16:32:38
  Copyright (c) 2013-2016 TRUSTONIC LIMITED. All rights reserved.

usage: GP TA properties
  --gp_prop help                Display this help
  --gp_prop read <GP TA binary.tabin>  Display GP TA properties
                                       from GP TA binary

  --gp_prop write <GP TA binary.axf> <config file .xml>
                                       Write GP TA properties
                                       from .xml to GP TA binary
```

Sample XML file :

```
<?xml version="1.0"?>
<properties>
  <group name="gpd.ta">
    <property name="singleInstance"    type="BOOLEAN" value="false"/>
    <property name="multiSession"      type="BOOLEAN" value="false"/>
    <property name="instanceKeepAlive" type="BOOLEAN" value="false"/>
    <property name="description"       type="STRING" value="An example of
GP internal APIs"/>
  </group>
  <group name="com.trustonic.ta">
    <property name="name"              type="STRING" value="taSampleGP"/>
    <property name="vendor"            type="STRING" value="Trustonic"/>
  </group>
</properties>
```