OPEN MOBILE PLATFORM

Trusted execution of applications on a modern Linux-based smartphone: Secure Boot, ARM TrustZone, Linux IMA

Konstantin Karasev, TEE Architect
Dmitry Gerasimov, Software Engineer

October 5, 2019

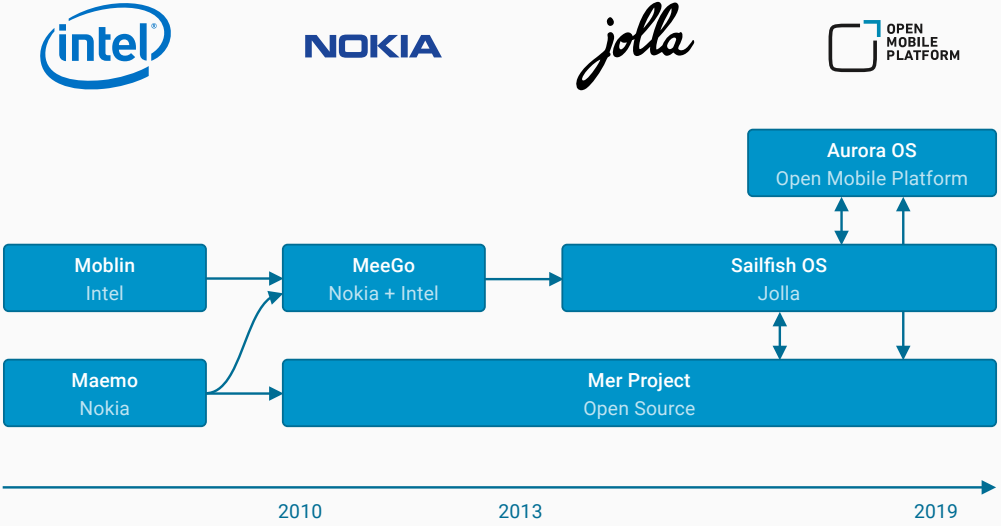
Open Mobile Platform

About us

Open Mobile Platform (OMP):

- Aurora OS, SDK and mobility services ecosystem development
- R&D centers in Innopolis and Moscow
- Strategic partner — Rostelecom

Aurora OS — Independent Linux-based mobile OS


Aurora OS
Open Mobile Platform
Moblin
Intel
MeeGo
Nokia + Intel
Sailfish OS
Jolla
Maemo
Nokia
Mer Project
Open Source
2010
2013
2019

Aurora OS vs Sailfish OS Highlights

- Focus on B2B and B2G market
  - Field staff terminals
  - Corporate office smartphones & tablets
- Security architecture
- Integration with enterprise infrastructure services (MDM, EMM)
- Federated and on-premise mobility services: updates, push, Appstore, etc
- API

Aurora OS is fully developed by Open Mobile Platform team and has own release cycle and product roadmap

Agenda

- Boot sequence of ARM-based smartphone
- Trusted boot of Linux kernel
  - Root and chain of trust
  - Bootloaders
  - TrustZone OS
  - Linux Kernel
- Linux IMA — trusted launch of applications
  - What is IMA?
  - How does it work
  - Integration experience
- Trusted boot mechanisms applicability in mobile OS

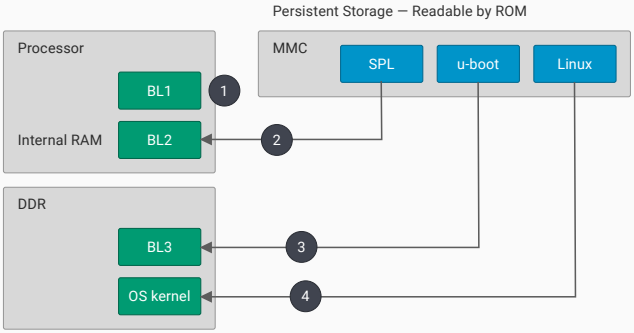
Boot sequence of ARM-based smartphone

Boot sequence of ARM-based smartphone

Usually there are 4 boot stages:

- BL1 — ROM code, BOOTROM, ...
- BL2 — SPL, preloader, ...
- BL3 — u-boot, aboot, lk, ...
- OS kernel — Linux, ...

1. BL1 loads BL2 from initialized persistent storage (e.g. MMC) to processor internal RAM
2. BL2 initializes external RAM (DDR) loads BL3 from MMC to DDR
3. BL3 loads OS kernel from MMC to DDR
4. OS kernel starts up



Trusted boot of Linux kernel

Trusted (Secure) Boot

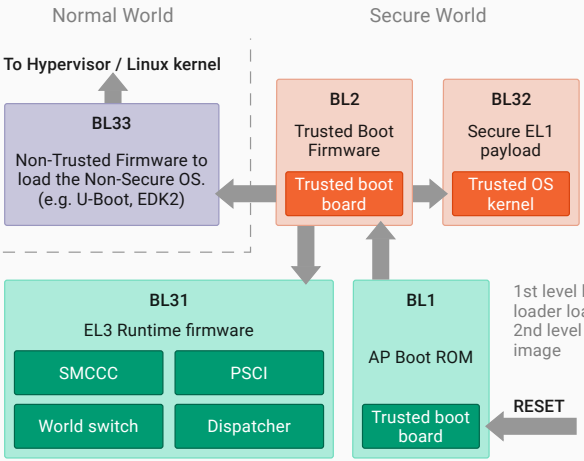
OPEN MOBILE PLATFORM

Trusted Boot — a boot sequence, where each consequent SW component is verified by preceding, already verified SW component. First SW component is HW-immutable and is considered as a trusted one.

- Root of trust
  - ROM code
  - Programmable ROM (e.g. EFUSE)
- Implements chain of trust from root to all critical (i.e. those requiring verification) read only SW components
  - Hash functions
  - Asymmetric cryptography (RSA)
- Allows verify SW components and block device booting if verification of SW component failed
- Until BL33 device is in Secure mode (details will follow)

Note: EFUSE — one-time-programmable memory inside SoC

Trusted (Secure) Boot



EL3 Execution

Secure EL1 Execution

Normal EL2/EL1 Execution

EDK2 — EFI Development Kit 2
EL — Exception Level
PSCI — Power State Control Interface
BL — Boot Loader
SMC — Secure Monitor Call

What is protected

- Bootloaders hierarchy (BL2, BL3x)
- Trusted OS (if present)
- Linux kernel and initramfs
- Read-only OS components
- Firmware (modems, ...)
- Rollback protection:
  - Write latest installed SW version to immutable (or tamper-evident) storage (e.g. EFUSE)
  - Write version to signed SW image
  - On boot check if image version $\geq$ version in EFUSE, otherwise block booting
  - On successful boot update SW version in EFUSE if needed

Chain of trust (1/3)

- Root of trust:
  - BOOTROM code and RSA key hash, written to EFUSE
  - Hardening: several keys, key inversion
- Certificate chain BL2:
  - Key certificate
    - Public Root key
    - Public Image key
    - Version
  - Image certificate
    - BL2 image hash
    - Version

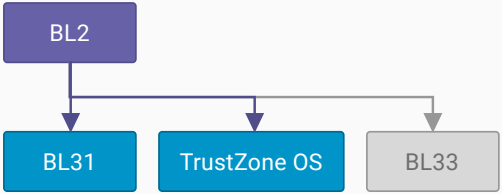
BOOTROM checks integrity and autheticity of Root key, then checks integrity and authenticity of BL2 image and pass control to BL2

Chain of trust (2/3)

BL2 checks and loads Trusted firmware (BL31), TrustZone OS (BL32) and consequently passes control to them for initialization

BL31 controls platform peripherals (e.g. via PSCI) and dispatches interrupts and requests to TrustZone OS

TrustZone OS (BL32) is an execution environment for trusted services



Usually key hash is written to EFUSE at device manufacturing stage.

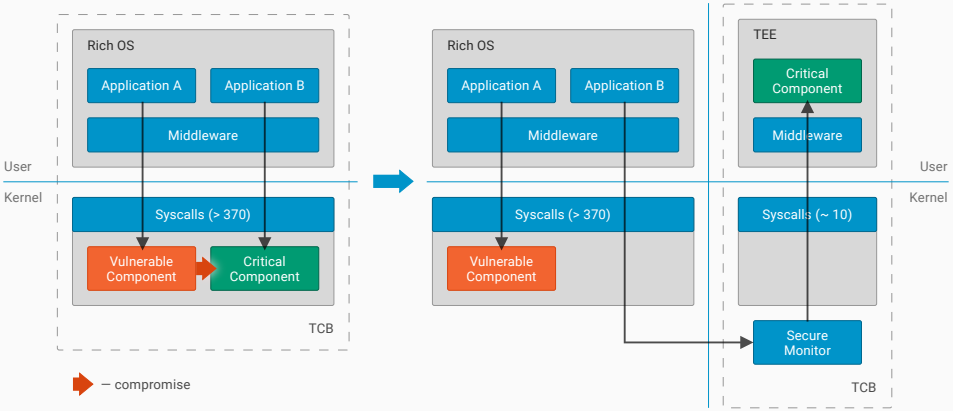
Further writes to EFUSE are also possible from BL1/BL2/BL31/TrustZone OS (BL32), reads (with limitations) are possible from BL33/Linux (usually via TrustZone OS services).

- TEE use cases and implementations
- ARM TrustZone
- Overview of TEE implementations in ARM TrustZone

What is TEE

TEE — Trusted Execution Environment:

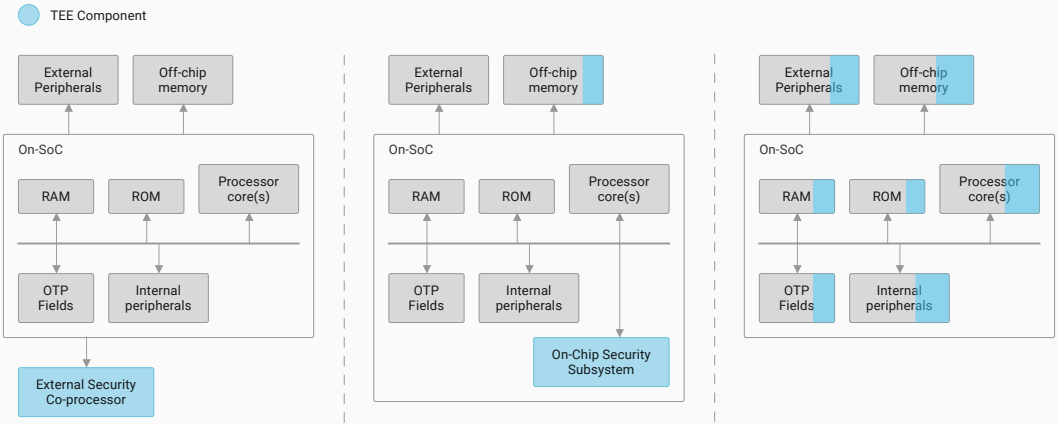
- HW-isolated execution environment
- Trusted Code Base (TCB) reduction
- Attack surface reduction



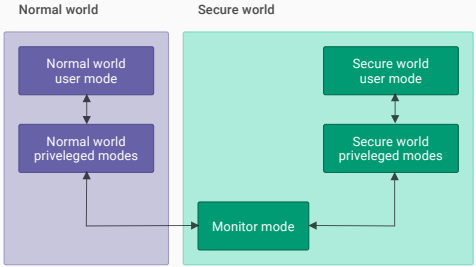
TEE use cases

- Cryptography
  - Keys isolation
  - Keys usage (encryption, certificates, ...)
- Biometric authentication
  - Fingerprint
  - Iris
  - Face
- Financial services
  - Mobile banking
  - Mobile payments
  - Mobile point of sales (MPOS)
- DRM-content playback
  - L3 — TEE is not involved
  - L2 — content decryption
  - L1 — decryption, processing and playback
- Healthcare
  - Heart beat
  - Blood pressure
  - Others

TEE implementations



ARM TrustZone



- *Normal World* (NWd, Rich Execution Env., REE) — CPU mode and SW; *NS bit is 1*
- *Secure World* (SWd, Trusted Execution Env., TEE) — CPU mode and SW; *NS bit is 0*
- *Secure Monitor* — CPU mode and SW running in it; NS bit is ignored on accesses (always 0)
- *Trusted application* (TA, trustlet) — a SW which implements a service in TEE
- *Client application* (CA) — a SW of TEE service client

Component	ARMv7	AArch64/32
Hypervisor	HYP mode	EL2
SWd privileged mode	SVC mode (NS = 0)	S-EL1
NWd privileged mode	SVC mode (NS = 1)	(NS-)EL1
SWd user mode	USR mode (NS = 0)	S-EL0
NWd user mode	USR mode (NS = 1)	(NS-)EL0

ARM TrustZone SW architecture variants

OPEN MOBILE PLATFORM

1. Trusted (Secure) OS
   - Parallel TA execution
   - Dynamic TA loading and updates
   - Address spaces isolation
2. Synchronous library
   - E.g. ARM Trusted Firmware components (PSCI and other OEM services)
3. Intermediate solutions
   - OS without source of interrupts (vulnerable to DoS)
   - A library with MMU-based address spaces isolation

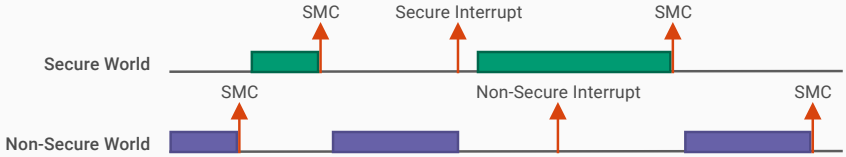
Switches between Linux and TZ OS

SMC

- ARM SMC Calling Conventions
- Fast or Standard (yielding)
- Bidirectional
- Similar to system call (SVC)

IRQ/FIQ

- Depends on GIC version and setup
  - GICv2 - FIQs: SWd / IRQs: NWd
  - GICv3 - IRQs: SWd / FIQs: NWd
- Works with Standard SMC only



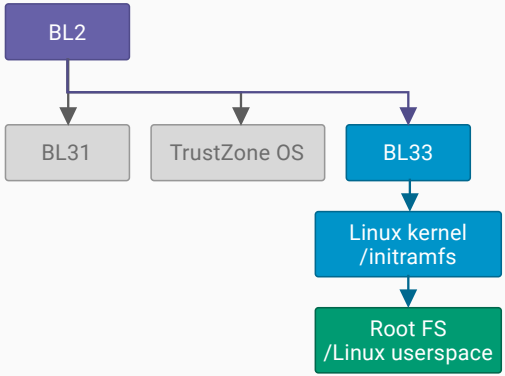
Some TEE implementations

OPEN MOBILE PLATFORM

Title	Vendor	License	Details
Kinibi (Trusted Foundation, MobiCore, t-base)	Trustonic	Proprietary	L4-based, GP certified, CPU boost
QSEE	Qualcomm	Proprietary	Monolithic, ASLR
TSEE (TKCORE, TrustKernel)	Shanghai Pingbo Information Technology Co.	Proprietary	Runtime data encryption, GP certified
ProvenCore	Prove&Run	Proprietary	Formally proven security, POSIX, microkernel
ISEE	Beijing Beanpod Technology	Proprietary	L4-based, GP certified
TEEGRIS	Samsung	Proprietary	Hybrid, GP certified, ASLR/KPTI, CPU boost
Yun OS TEE (Yunos TEE)	Taobao (China) Software Co., Ltd. (Alibaba Group)	Proprietary	A TEE for Yun OS, GP-certified
OP-TEE	Linaro	BSD	GP API centric, no shared libraries, no scheduler, monolithic, KPTI
TLK (Trusty, Google Trusty, Trusty Little Kernel)	Google (Nvidia according to Wiki)	BSD	Based on LK, no TA priorities, no GP API support, monolithic
GzOS	OpenTrustGroup	BSD	Zircon-based (Fuschia), capability-based security, Trusty and GP API support, microkernel
OMP-TEE	OMP	BSD/Proprietary	

Chain of trust (3/3)

- BL2 checks and loads BL33 (u-boot, others)
- BL33 checks images of recovery, logo, firmware and boot (Linux + ramfs)
- BL33 loads firmware and Linux and passes control to Linux
- After Linux booted chain of trust can be expanded to Rich OS components by using dm-verity, IMA, etc.



Linux IMA — trusted launch of applications

IMA (Integrity Measurement Architecture) — Linux kernel subsystem for controlling and enforcing system integrity.

History of enhancements in Linux kernel:

- Linux 2.6.30 — initial implementation
- Linux 3.3 — work with signatures
- Linux 3.7 — IMA-appraisal, IMA-audit
- Linux 3.12 — load only trusted keys
- Linux 3.19 — load certificates from kernel

IMA can be used to continue trusted boot chain in userspace.

Current implementation allows:

- generate table of file hashes on access
- verify file signatures
- log/prohibit execution of incorrectly signed files
- log/prohibit work of processes with incorrectly signed files

IMA calculates hash of each file before open (SHA-1/SHA-256). Measured values can be obtained from the virtual file:

```
/sys/kernel/security/ima/ascii_runtime_measurements
```

Example:

```
10 ddee...b59b ima-ng sha1:9797ed...4e45ee boot_aggregate
10 180e...8a52 ima-ng sha1:db8291...23cf3a /init
10 ac79...ea65 ima-ng sha1:f778e2...2af4db /bin/bash
10 0a0d...bb05 ima-ng sha1:b0ab2e...4359a7 /lib64/ld-2.27.so
...
10 4da8...8b27 ima-ng sha1:99a9c0...5f49de /etc/passwd
```

File signatures or reference hashes are stored in extended attributes:

```
$ getfattr -d -m ima /usr/bin/awk
# file: /usr/bin/awk
security.ima=0sAwIEBsbQzwCAuu7DwFGijSMm8...39LQrU=
```

Generally, signatures are installed by package managers. E.g. both `rpm` and `dpkg` have support for IMA.

Alternative approach is to set signature with `setfattr` or sign file in place with `evmctl`.

Key Retention Service allows to store cryptographic keys for further use by kernel. Enabled by option:

```
CONFIG_KEYS=y
```

One of the features of this service — combining keys into a *keyring* (list of keys).

Keyring used by IMA has a name «`.ima`».

Signatures are verified with certificates loaded into the `.ima` keyring.

A certificate can be loaded into .ima keyring either by kernel itself:

```
CONFIG_IMA_LOAD_X509=y
CONFIG_IMA_X509_PATH="/etc/keys/x509_ima.der"
```

Or it can be loaded from userspace (keyctl, evmctl):

```
description=$(keyctl describe %keyring:.ima)
keyring_id=${description%%:*}
evmctl import /etc/keys/x509_ima.der ${keyring_id}
```

Limitations

OPEN MOBILE PLATFORM

It's impossible to limit execution of incorrectly signed *scripts*:

```
# appraisal works, execve() --> ima_bprm_check()
$ ./hello.py
python: ./hello.py: Permission denied

# appraisal do not work, open() --> ima_file_check()
$ python ./hello.py
Hello, World!
```

Solutions:

- ~~do not use interpreters~~
- modify interpreters

How do we use it

OPEN MOBILE PLATFORM

We use IMA to

- control system integrity
- verify third-party application signatures

IMA is integrated into RPM: extended attributes are restored on package install.

The main problem we encountered on some devices when enabling IMA in Aurora OS — old kernels (3.10, 3.14) which do not suport all required features.

Questions?

Backup slides

Links

- https://source.android.com/security/verifiedboot
- https://android.googlesource.com/platform/external/avb/+/master/README.md
- https://source.android.com/devices/tech/ota/nonab/block
- http://www.virtualopensystems.com/en/services/arm-trusted-firmware-extended-services/
- https://source.android.com/devices/tech/ota/ab/ab_faqs#does-avb2.0-require-ab-otas
- http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf
- http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf
- https://globalplatform.org/specs-library/?filter-committee=tee
- https://habr.com/ru/post/309618/
- https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/firmware-design.rst
- https://optee.readthedocs.io/architecture/core.html

IMA setup example

Let's try to setup IMA on a simple example: *prohibit execution of any incorrectly signed files.* For this we have to:

- generate keypair *A* and root certificate based on it
- generate keypair *B* and certificate based on it to be loaded into .ima keyring
- sign all system executables using the *B* keypair
- build kernel with IMA support and built-in root certificate
- add several options to kernel command line
- apply IMA policy

IMA setup example: certificates, signatures

To generate certificates, you may use scripts that can be found in the ima-evm-utils package.

Generate a keypair and a root certificate, which will then be built into the kernel:

```
/usr/share/doc/ima-evm-utils/ima-gen-local-ca.sh
```

Next, generate a keypair for files signing and corresponding certificate (based on root certificate):

```
/usr/share/doc/ima-evm-utils/ima-genkey.sh
```

Then we need to sign all executables:

```
find / -fstype ext4 -type f -uid 0 -exec \
evmctl ima_sign -a sha256 --key privkey_ima.pem '{}'
```

IMA setup example: Kconfig, kernel command line

We need to put root certificate into kernel source tree and then build kernel with following options:

```
CONFIG_INTEGRITY=y
CONFIG_INTEGRITY_SIGNATURE=y
CONFIG_INTEGRITY_ASYMMETRIC_KEYS=y
CONFIG_IMA=y
CONFIG_IMA_APPRAISE=y
```

Also add several options to kernel command line:

```
ima_appraise=enforce ima_hash=sha256
```

IMA setup example: certificate loading

After booting a kernel with enabled IMA support, certificate needs to be loaded into the kernel `.ima` keyring:

```
description=$(keyctl describe %keyring:.ima)
keyring_id=${description%%:*}
evmctl import x509_ima.der ${keyring_id}
```

IMA setup example: policy

Minimal policy to check all executables signatures:

```
appraise func=BPRM_CHECK appraise_type=imasig
```

Policy must be written into a virtual file:

```
echo 'appraise func=BPRM_CHECK appraise_type=imasig' \
    > /sys/kernel/security/ima/policy
```

If the policy is successfully applied, file will disappear.

Usually, systemd is responsible for loading policy into the kernel.

IMA setup example: Hello, World!

Let's create a small script with following content:

```
#!/bin/sh
echo "Hello, World!"
```

and save it with a name `hello.sh` and `755` permissions.

Next, just try to execute it:

```
$ ./hello.sh
sh: ./hello.sh: Permission denied
```

That's it! We cannot execute an unsigned file.