NOKIA

Research Center

NRC-TR-2007-015

Mobile Trusted Module (MTM) - an introduction

Jan-Erik Ekberg, Markku Kylänpää

Nokia Research Center Helsinki, Finland
http://research.nokia.com
November 14, 2007

Abstract:

This paper provides a brief overview of the *Mobile Trusted Module* (MTM), its features and capabilities with respect to TPMs. The viewpoint is the device manufacturer's, i.e. the functionalities defined by the Remote Owner MTM (MRTM) profile of the specification. Additionally the paper presents a few use cases related to the technology, the implementation of a MRTM emulator and a proposal for an API to control MTM functionality.

Index Terms:

platform security
mobile phones

1 Introduction

The Mobile Trusted Module (MTM) is a security element and a newly approved TCG specification [1], [2] for use in mobile and embedded devices. Its origin lies in the TPM v. 1.2, but the mobile specification significantly differs from the original specification on a few issues:

1. The concept of *secure boot* is introduced. Many embedded devices and handsets in particular are subject to regulatory approval. That in turn motivates the need for enforced integrity protection of software in fielded devices, and secure boot, i.e., a boot sequence not only measured, but also aborted on any non-approved state transition, is a vital building block for this security service.

2. The specification explicitely supports implementation of the MTM as a functionality rather than as a physical implementation in hardware. This makes it possible for device manufacturers to add the MTM as an add-on to already deployed, proprietary security solutions.

3. In addition, the reference architecture takes into account the support of several parallel MTM instances in the same device. Some will be discretionary (MTM exposed to user applications) whereas e.g. the Device Manufacturer MTM by definition enforces security policy (mandatory access control).

This paper introduces the new features of MTM in the context of a reference implementation. MTM defines two interleaving profiles depending on the entity that holds ownership of the functionality - Mobile Remote Owner Trusted Module (MRTM) and the Mobile Local Owner Trusted Module (MLTM). We have focused on the former profile (MRTM), since the Remote Owner specification nicely contains all the new functionality of the MTM with respect to TPMv1.2. Intended to be used either by the device manufacturer or a carrier operator, the MTRM defines the security architecture and interfaces to implement an integrity-protected device.

The paper starts by introducing the main system components and interfaces that make up the secure boot concept in MTM - in particular the Reference Integrity Metric (RIM) certificates and their enforcement. Then we propose new additions to the TSS interface for managing system parameters and the secure boot concept in MTM. Our reference implementation is described in its own section. We conclude with a few general ideas on how to use RIM certificates for a variety of security-related purposes and finally sum up the lessons learned in a conclusion section.

2 MTM setup and state

The MTM standard is flexible enough to accommodate several different architectures for guaranteeing the integrity of the MTM in the scope of secure boot, and the mechanisms used to provide root secrets and immutable data to it. In this section we give a description of one setup that is acceptable within the context of the MTM standard, while at the same time describing the essentials of some MRTM functions and parameters. Please refer to the standards text for an exhaustive coverage of the secure boot phase.

2.1 Initial boot steps

For secure boot to be enforced, the system must have passed through two discrete states - the *Engine Reset* and *Engine Root-of-Trust Initialization* phases prior to activating the MTM. The reset state simply describes the fact that no software is running on the device prior to first MTM setup. The initialization phase in which the software MTM is set up is in the standards context defined by a set of "Roots of Trust", each of which describes a necessary security precondition to be satisfied in order for the MTM security to be complete. The *Root of Trust for Enforcement* (RTE) essentially states that platform-specific mechanisms must be used to guarantee the integrity and authenticity of the MTM code and its execution environment. Some form of device secret will be needed to establish a *Root of Trust for Storage* (RTS), and some immutable or similarly protected code will constitute the *Root of Trust for Verification* (RTV), an engine that makes the initial measurements to be added to MTM prior to the MTM being fully functional. A *Root of Trust for Reporting* (RTR) holds the secrets to sign PCR measurements for attestation purposes. An RTS with suitable statefulness guarantees can be considered to contain also the RTR and RTV.

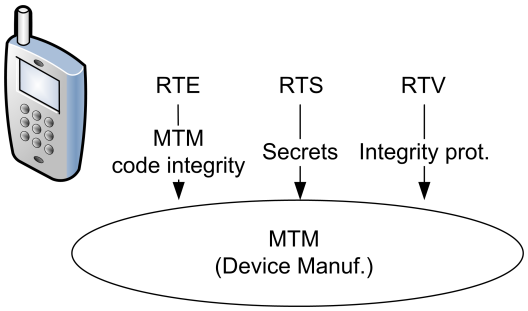

Figure 1: Our reference architecture

Figure 1 shows the reference model that we are targeting in this paper. The device has its own, proprietary way of ensuring boot integrity. That concept is used to protect the initial boot sequence up to the place where the MTM is loaded into a secure environment, e.g. to some on-processor memory. We also assume the presence of at least one device secret (a.k.a. RTS), passed to the MTM when loaded, to be used as a seed for confidentiality services in the MTM (e.g. the seed for the SRK). Additionally, immutable information for verifying verification keys and further RIM certificates, called the Root Verification Authority Information (RVAI) needs to be presented to the MTM. Furthermore the continued computational isolation of the MTM function with respect to e.g. the OS being loaded and run must be ascertained in order for the architecture to be secure.

2.2 MTM internal state

In comparison to a TPM, MTM requires some additional persistent state information. Most of this is stored in a new data structure specified by the standard, called *MTM_PERMANENT_DATA*.

Most parameters and attributes will be elaborated on in later sections, but a brief overview follows. The structure begins with a storage area for the attestation key AIK, used if it is pre-assigned to an MRTM during manufacuring. The boolean flag structure *verifiedPCRs* indicates the set of PCRs constrained to b extended only by means of RIM certificate. The value of the bootstrap counter is located in the permanent data, along with the unique identifiers of the two other mandatory counters. The bits of *loadVerificationKeyMethods* field specify what validation methods are supported when loading verification keys. The field *integrityCheckRootData* is used to store a value that can is used to bind the value of a root verification key to the MTM. The value is proprietary and can e.g. be a hash of the root verification key. The field *internalVerificationKey*, typically a HMAC key, is used to sign and validate internal RIM certificates. The *verificationAuth* value is used for authorization of the *MTM_InstallRIM* command.

An additional bit, the *loadVerificationRootKeyEnabled* is also part of the global state, but located in the *MTM_STANY_FLAGS* structure.

2.3 Initial keying

For a MRTM the concept of "Taking Ownership" of the functionality cannot be allowed if secure boot is used. This option can therefore be disabled, and the SRK can be generated at manufacturing time and inserted into the device.

Also, in a typical TPM, the process of enrolling identities to it is achieved through the EK and AIK keys. As privacy is not necessarily a consideration e.g., for an MRTM, the identity is allowed to be pre-set during manufacturing. In other words the EK is not necessary included in the MTM, and the AIK and related certificates are pre-installed and cryptographically bound to the device before reaching the customer.

Secure boot relies on a third semi-fixed configuration - the definition of the PCR types. Each PCR can be restricted to be updatable only by means of an update matching and validated by a RIM certificate. The bitfield defining this restriction for the individual PCRs can be updated only by the owner of the MTM (which in the MRTM case is the manufacturer, so the PCR updates in this aspect are "remotely

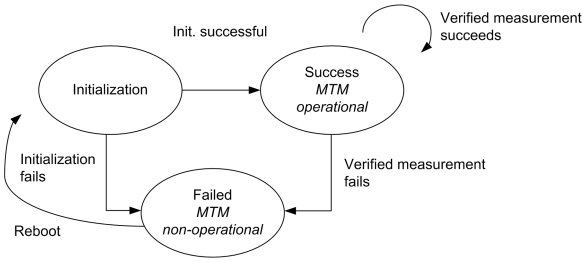

Figure 2: Secure boot operation (from spec.)

controlled"). This feature will be used as the basis for secure boot after the initial MRTM has been activated.

2.4 MTM operation and RIM certificates

At a high level of abstraction the MTM runs in one of three states during the boot phase. Figure 2 shows that the device boots into an initialization state, governed by the (non MTM) security of the platform. On successful initialization the operation moves to a *success* state where it stays as long as the PCR updates done using RIM certificates match a known state as given by the RIMs. On failing to reach such a state the MTM moves to a *failed* state, becomes inoperational, and stays that way until a device reboot. Additionally, when disabling the MTM the device may forcefully reboot, lock up, disable some hardware interfaces or take any other approach necessary for preserving device security and integrity.

3 MRTM - the Device Manufacturer's MTM

To best illustrate the scope of the MTM from a device manufacturer's perspective, the minimal functionality of "a software" MRTM is next presented. As an illustration, the secure boot process is explained embedded into the discussion.

During manufacturing, let's assume that there is no incentive for a device manufacturer to enroll new identities to a deployed unit, so no endorsement key *EK* is installed in the device. Instead, the device manufacturer decides on the *AIK*, *SRK* and *RVAI* keys, and configures them into the MRTM, e.g., by means of a secure storage defined by the secure bootstrap. Also the *verifiedPCRs* list of PCRs that only can be updated using RIM certificates is defined, and the *loadVerificationRootKeyEnable* flag is set to *false* - this disables updates to verifiedPCRs list and the *RVAI*. Thus, during MRTM operation, certain, predefined PCRs can only be modified using RIM certificates. The certificates are bound to so called *integrity keys* positioned in a key hierarchy under the installed *RVAI*.

Let us assume that the MRTM is software. Thus, the device is assumed to contain the secure bootstrap defined as *RTE*, and *RTS* can be extracted as part of the same bootstrap. The first part of *RTV* boots in this context (e.g. from a ROM), launches the *MRTM* and its associated state. Then *RTV* according to the specification makes diagnostic measurements regarding itself and the roots-of-trust, and feeds this information (using e.g. *MTM_VerifyRIMCertAndExtend*) into PCRs 0 and 1. It also measures and extends into PCRs the next code to be run, this presumably being some form of bootloader.

The secure boot illustration (figure 3) shows the boot steps in a graphical form. An important feature of the RTV is the termination of execution (or the forceful execution of some other action) if any invocation of VerifyRIMCertAndExtend fails. This is the essence of the secure boot feature, and essentially what sets the MTM apart from a traditional TPM.

3.1 Counters

Counters are included in the MTM specification to achieve freshness guarantees (rollback protection) for the secure boot procedure. No specific counters for data/key protection are mandatory.

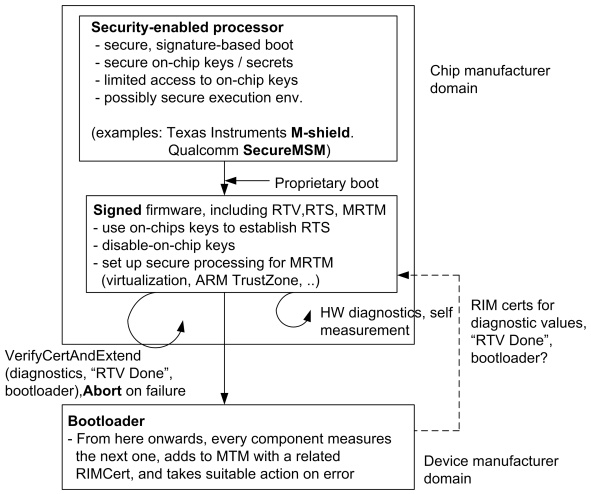

Figure 3: Secure boot

Three counters need to be managed by the underlying platform architecture. The *bootstrap* counter is initialized to 0, and runs up to a minimum of 31 steps. This counter is intended to protect the MTM bootstrap, and there must be a one-to-one correspondence between a RIM certificate for the bootstrap (the "firmware" in fig. 3) and a specific code version. It is critical for the integrity of firmware upgrades in secure boot systems that this correspondence holds and the small range of values makes it possible to implement the counter as (31) one-time programmable bits in hardware.

The implementation of a separate *RIMProtect* counter (running up to a minimum of 4095 steps) is also required. Even if any RIM certificate can be bound to the bootstrap version, binding certificates to RIMProtect enables a higher resolution of (software) upgrades. Still, the RIMprotect counter is singular, i.e. any security-relevant upgrade needed by any functionality bound to this counter induces the need for renewal of all other RIM certificates bound to RIMprotect, an issue that is partially solved by the introduction of RIM internal certificates.

A third mandatory counter, the StorageProtect counter, is intended to serve as state-protection for the RTS-protected storage. It is not addressed in any MTM interface function, and its minimal size (4095 bytes) implies restrictions in use frequency.

The RIMProtect counter is updated by the possibly user-authenticated *TPM_IncrementCounter*, whereas the Bootstrap counter can only be updated by the *MTM_IncrementBootstrapCounter* command, its use governed by a RIM certificate. This is in line with the estimated use of the Bootstrap counter - it is used when a security breach in the bootstrap is found whereby the authorization secret of a *TPM_IncrementCounter* command could have been compromized. In addition to RIM certificates, verification keys can also be bound to the abovementioned counters.

To be noted is that the MTM specification is somewhat lax on the platform support needed for counter protection, but intends to be more strict in future standard versions. For implementations of the v 1.0 specification, the statefulness guarantees provided by the counters will vary between devices.

3.2 RIM Certificates

The trust bindings related to the RIM certificates is multi-faceted. A certificate is typically bound to a counter and one or more PCR values. An *external* certificate is signed by a *verification* key, whereas *internal* ones are essentially tickets for the same data structure, produced by the MTM instance itself for future reference regarding the validity of the certificate. These two mechanisms are examined in the following:

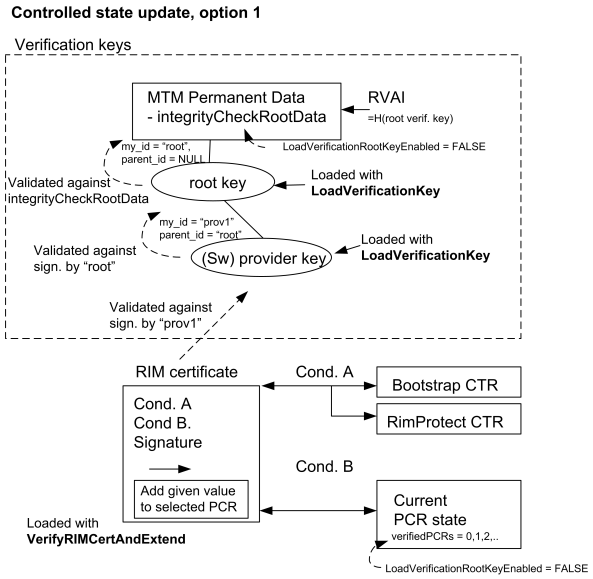

Figure 4: State update with external RIM certificates

3.2.1 External RIM certificates

The verfication keys form a hierarchy with respect to the MTM - the root key of the hierarchy is bound to the root verification authority identifier (RVAI), which depending on the MTM implementation can vary, but typically would be a hash of the root verification key. The RVAI can be made unmodifiable by the *MTM_LoadVerificationRootKeyDisable* command, turning the internal *loadVerificationRootKeyEnabled* flag to *false*. The same internal state bit disables the *MTM_SetVerifiedPCRSelection* command, fixing the set of PCR registers only modifiable by the use of RIM certificates. Neither command will ever be needed in an MRTM that is factory-initalized with pre-set values in the *RVAI* and *verifiedPCRSelection* parameters and the *loadVerificationRootKeyEnabled* bit permanently set to *false*.

When loading a verification key (using *LoadVerificationKey*) either a signature that is part of the verification key structure is validated against the key indicated by the *parent key handle* of the verification key being loaded, or, in case of the loaded key being the root key, against the RVAI in any appropriate manner. Thus every successfully loaded verification key is guaranteed to be part of a verification key hierarchy.

When a PCR update is initiated by a *MTM_VerifyRIMCertAndExtend*, the signature on the external RIM certificate is checked by a loaded verification key indicated by a key handle in the command. If the signature matches, and the preconditions, counter value equivalence and PCR contents match, a given PCR is updated with a value stated in the RIM certificate. A managerial function *MTM_VerifyRIMCert* is included to check a certificate against a verification key and the fact that the counter constraint of the certificate satisfies either the current or any future counter value for the referred counter. In the same manner, a RIM certificate is also used for the authorization of an *MTM_IncrementBootstrapCounter* command, which sets the bootstrap counter value to the value indicated in the RIM certificate on the condition that the verification key referenced by the certificate is a key that by parametrization is allowed to control bootstrap counter updates.

3.2.2 Internal RIM certificates

External certificates can be "internalized", i.e., rather than validating the certificate against a verification keys on every load, the validation can be done once, whereafter the MTM can convert the external certificate into an internally validated certificate. The constraining relations of an internally validated certificate are similar to an external one, but the signature is a HMAC on a key only known to the MTM, present for the singular purpose of signing internal certificates. The conversion process is executed using the *MTM_InstallRIM* command. Internal certificates can only be tied to the *RIMprotect* counter.

A couple of details regarding certificate internalization are worthy of explanation. First, the authorization of the InstallRIM command is not based on verification keys, but on a TPM authentication token on a key *verificationAuth*, assumed to be placed into the MRTM at the time of manufacture - there is no way to modify or change this symmetric key. Additionally, the verification process is assumed to be completed by an MTM-external component (knowing the verificationAuth) validating the key hierarchy needed for the external certificate to be internalized. The MRTM can be used as "an oracle" for this - on accepting integrity keys, their validity is assured. Also, the validity check of the external certificate can make use of the *MTM_VerifyRIMCert* command, and deduce correctness from return parameters. Only after these checks, the external party is assumed to produce the authenticated *MTM_InstallRIM* command for RIM certificate internalization. Thus the validity of the certificate installation is not guaranteed by the MTM itself, but rather by any entity with knowledge of *verificationAuth*.

A second detail regarding the internalization of certificates is that the internal certificate will have a counter binding of the current RIMProtect counter value plus one, independently of the counter reference value of the external certificate. Omitting counter checks during the install command is in line with the trust model of the command, and setting the counter reference to one more than the current counter value makes it possible to install many certificates and make them all usable with one update of the RIMProtect counter.

The process of installing RIM certificates into MTM can be governed by a revocation mechanism built around *RIM_Auth Validity Lists* and *RIM Validity Lists* [1]. The first one is a fresh list of currently valid verification keys that are signed by a given, higher-level verification key. If a key is not on the list, it must be assumed to be revoked. A similar construct is available for the actual certificates in the form of the RIM validity lists, maintained by the authorities behind the verfication keys that sign the certificates. These lists are based on the notion of universal time, and it is assumed that the device managing the validity lists has a reasonably accurate and secure clock to support the decisions based on these lists.

4 Supporting multiple MTMs in a device

The reference architecture [1] strongly brings forward the idea of interlocked MTMs in a single device, based on three intehrity-protected lists - the device manufacturer's mandatory engines, the device owner's mandatory engines, and the device owner's discretionary engines. The basic assumption is that all mandatory engines should be securely activated as part of a proper boot, and failure to do so should be treated as a boot error. The integrity-protected lists would be controlled by the device manufacturer and owner respectively. For software implementaitions, it is also to be assumed that the local security (RTV, RTS) of all engines are boostrapped from the device MRTM whereas the identity-related roots-of-trust may and probably should vary between engines. This architecture makes it possible to have separate MTM service domains, where unique engines intended to serve different stakeholders and applications are co-located in the same device. In a general sense, the same architectural ideas have already been presented in the TPM domain, a good example being research conducted around virtual TPMs, e.g., by IBM Research [7].

5 Other MTM commands

The MRTM inherits a fair bit of mandatory functionality from TPMv1.2. The basic storage functions related to binding and sealing are all supported, and since the PCRs defined in *verifiedPCRs* also can be used for the PCR binding, a new (higher) dimension of configurability for services like secure storage is achieved. Signing and key certification are also supported. Verification keys can be evicted using *TPM_FlushSpecific*. For PCR updates in the non-verified set the basic *TPM_Extend* is available, and for reporting at least *TPM_Quote* is present in all MTMs. However, especially in MRTMs there may be no EK (no support for provacy CA:s) and AIKs might be pre-installed, so the attestation function might be limited to one certifying authority. In general, delegation, timing, non-volatile memory services and migration, as well as most administrative maintenance commands are declared optional in the context of MTMv1.

6 MTM implementation

For validating the MTM specification we have added MTM specific functions to an open source TPM emulator [4] and made some modifications to existing TPM routines as specified in [2]. The code implements the mandatory command set of an MRTM, as well as optional MRTM and MLTM functions inherited from [4].

6.1 TSS API

TCG has not defined any new TSS functions for the MTM-functionality. The main reason for this is that the MTM engines are primarily targeted for embedded environments and for the booting phase (MRTM), where consistent and standardized APIs are of less relevance than for TPMs, which primarily are intended to be used by applications. Even so, a consistent API is valuable also in closed environments if it represents a known good design, and in the context of MLTMs RIM certificates might also be used by applications. We here propose an API for this purpose, which has emerged from the need to test our own MTM implementation.

We have extended the TCG standardized TSS API with necessary functions to access MTM functionality. This could be a starting point for defining mobile specific TSS (MTSS) or merging these functions to TSS. In this work, we have intentionally avoided to define new types in TSS level, instead MTM specific types (RIM certificates and verification keys) are transferred as raw buffers, the contents reflecting the byte structures defined by the MTM specification.

The following new TSS functions have been defined:

```
TSS_RESULT Tspi_MTM_InstallRIM
(
 TSS_HTPM         hTPM,               // in
 UINT32           ulRimCertSize,      // in
 BYTE *           rimCertData,        // in
 UINT32 *         outCertSize,        // out
 BYTE **          outCertData         // out
 );

TSS_RESULT Tspi_MTM_VerifyRIMCert
(
 TSS_HTPM         hTPM,               // in
 UINT32           ulRimCertSize,      // in
 BYTE *           rimCertData,        // in
 UINT32           hVerificationKey    // in
);

TSS_RESULT Tspi_MTM_VerifyRIMCertAndExtend
(
 TSS_HTPM         hTPM,               // in
 UINT32           ulRimCertSize,      // in
 BYTE *           rimCertData,        // in
 UINT32           hVerificationKey,   // in
 TCPA_PCRVALUE *  pPcrValue           // out
);

TSS_RESULT Tspi_MTM_LoadVerificationKey
(
 TSS_HTPM         hTPM,               // in
 UINT32           hParentKey,         // in
 UINT32           verificationKeySize,// in
 BYTE *           verificationKeyData,// in
 UINT32 *         hVerificationKey,   // out
 BYTE *           loadMethod          // out
);
```


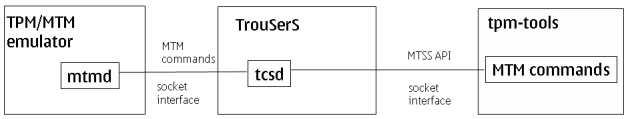

Figure 5: Architecture of the MTM emulator implementation

```
TSS_RESULT Tspi_MTM_LoadVerificationRootKeyDisable
(
 TSS_HTPM         hTPM                // in
);

TSS_RESULT Tspi_MTM_SetVerifiedPCRSelection
(
 TSS_HTPM         hTPM,               // in
 TCPA_PCR_SELECTION * selection       // in
);

TSS_RESULT Tspi_MTM_IncrementBootstrapCounter
(
 TSS_HTPM         hTPM,               // in
 UINT32           ulRimCertSize,      // in
 BYTE *           rimCertData,        // in
 UINT32           hKey                // in
);
```

The respective purpose of each command is self-explanatory, as the commands reflect MTM functions in a one-to-one manner. Error values or command success is indicated by the *TSS_RESULT* parameter. For each command above, a corresponding tpm_tools user command has been written. These are *tpm_setverifiedpcrselection*, *tpm_incrementbootstrapcounter*, *tpm_installrim*, *tpm_loadverificationrootkeydisable*, *tpm_loadverificationkey* as well as *tpm_verifyrimcert* that also executes the extend operation conditional to an input parameter.

6.2 Implementation details

The original open source TPM emulator [4] is a Unix daemon process that can be controlled using either Unix domain socket or by accessing /dev/tpm device. The device interface is provided mainly for backwards compatibility. The emulator has been written using C programming language. The software has its own small cryptographic library that is based on open source MP library [5].

The original code contains a few Linux dependencies (macros from Linux kernel source) and there is an option to build a Linux kernel module that provides the device /dev/tpm. In our implementation all Linux dependenices have been removed and the modified MTM/TPM emulator can also use Internet domain sockets, as shown in fig 5.

MTM specific data structures have been added alongside TPM permanent data structures in the TPM emulator. Additionally utility functions to marshal and unmarshal MTM specific data structures were developed and command handlers have been extended to include the new MTM functionality. Test utilities have been developed to create, sign and display *TPM_VERIFICATION_KEY* and *TPM_RIM_CERTIFICATE* structures. There are also test utilities for MTM specific commands.

The emulator has been connected to a modified TrouSerS [6] TSS stack using a socket interface. Unit test tools and commands have been written in the spirit of the TrouSerS *tpm-tools* test commands for testing and using the new MTM interfaces and functions in the emulator. The implementation has been validated in Linux, but porting the emulator to other POSIX-compliant architectures is straight-forward.

As the MTM specification contains lots of optional functionality and features, a policy file has been added for configuring the overall MTM functionality of the emulator for any given purpose or to match a given architecture.

6.3 Example architecture for use-cases discussion

The following listing contains a simple example session. The MTM emulator *mtmd* is first started together with the TrouSerS daemon *tcsd*. The MTM emulator is connected to tcsd using socket interface and the tpm-tools commands connect to tcsd using a socket interface as well. PCR 9 is included in the *verifiedPCRs* list. Thus an attempt to extend this PCR with *tpm_extend* fails. The command *tpm_loadverificationkey* is used to load one *TPM_VERIFICATION_KEY* into MTM. The command returns a handle. The command *tpm_verifyrimcert* is used to verify a RIM certificate using the verification key handle as a parameter (-r option) and to extend a PCR register (-e option). The PCR register to be extended is specified in the RIM certificate.

```
$ service start mtmd
$ service start tcsd
$ tpm_readpcr -n 9
PCR[09]: 0xf443319695979917a03b717049235423874d2716
$ tpm_extend -n 9
    -e 0xae97c310289ce1eb417f6edbfe47ba091eda4467
PCR extend failed.
$ tpm_loadverificationkey -i test.key
Handle is 0x123
Verification key installed
$ tpm_verifyrimcert -i test.cert -r 123 -e
RIM certificate successfully verified.
PCR extended.
$ tpm_readpcr -n 9
PCR[09]: 0x2b289e32d928a75e08fc01e730dec3135dc8c680
```

7 Examples and Use Cases

By inclusion of the RIM certificates, the MTM is more capable than a securely booted TPM. This section motivates this statement by introducing a few examples related to an imaginary embedded device depicted in figure 6 - something akin to a mobile phone. The device deploys secure boot with a trusted domain consisting of at least a bootloader, whereafter a customer might deploy his or her own OS, although the device comes with a certified OS and applications as well. The example is mirrored to use the underlying security architecture of Texas Instruments M-shield [3], but similar services can be found with many (mobile) core and chip vendors.

An important part of a mobile chip(set) is the HW part of the radio stack - sometimes little more the A/D converters but typically a fair bit of algorithms and state-machine logic as well. If the radio (frequency) is regulated, the access to this functionality should be restricted to the software certified for the purpose. This can be achieved by

1. wiring the radio enable/disable operation (I/O addresses) to be part of the secure processing environment address space (and thus be visible only for programs in that space)
2. defining the extension of one predefined PCR to also trigger the activation of the radio (as the MTM is software to be run in the secure environment, this is trivially achieved)
3. locking the PCR in question to be in the *verifiedPCRs* set and
4. issuing a RIM certificate from the aggregate PCR set of the approved software stack (up to OS and application) to extend the PCR in question

The example shows that enforcement is trivial to achieve - one simple hardware constraint (item (1)) which is reasonable to implement with or without MTM, and a "non-standard" modification to the MTM implementation (item (2)). All software can use the MTM interfaces as is, and the enforcement policy is software agnostic in the sense that OS upgrades and old-version revocation using the MTM counters can be defined in MTM terms over MTM interfaces.

The same logic can be transported to a different aspect of hardware configuration. In contemporary manufacturing, the price per unit drops as quantities go up. On the other hand, market segmentation

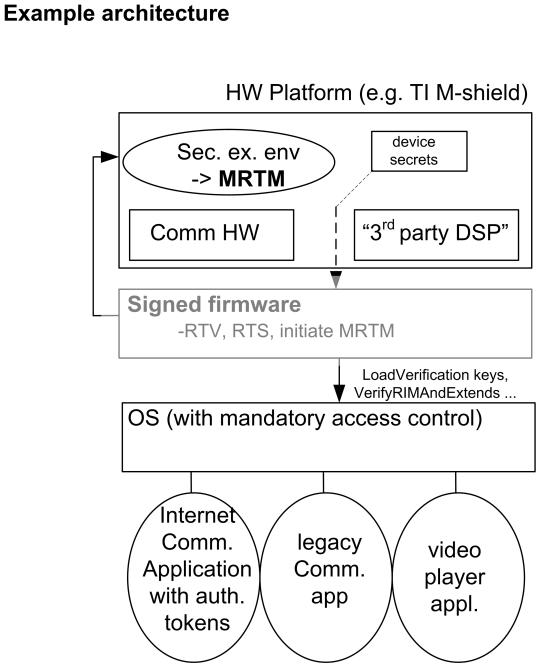

Figure 6: Example architecture

- having specific device models for specific user groups - is important. If we consider e.g. a video DSP, it might be beneficial for all stakeholders to include the DSP in all hardware, but enable it (and pay license costs to the designer of the DSP) based on RIM certificate activation in those devices where it is used. Even users could be allowed to upgrade their (restricted) devices in this manner. The logic above would be exactly the same, except that some unique device identifier (HW serial number) should always be added to some PCR to be able to produce RIM certificates dedicated to specific devices.

Another example, given in the specification ([2]), is to use PCRs, governed by RIM certificates, as a basis for supporting multi-stakeholder booting. Each stakeholder can use a different PCR to address its state, and stakeholders can address each others "result PCRs" as a condition for their own loading. This decouples the boot integrity mechanism from the versioning and update process related to the specific code of each individual stakeholder.

All previous examples also highlight the benefit to dedicate PCR registers for specific purposes in a given MTM. As the possibility for software implementation exists given the right hardware environment, it is feasible to include many more PCRs than the bare minimum required by the specification. With numerous PCRs available, constrained by the RIM certificate mechanism, a system can be made to resemble a property-based architecture. This in turn allows for greater flexibility in other TPM operations, i.e. providing keying support or sealing data for known (property-defined) states or setups.

8 Conclusions

The authors feel that the MTM technology introduces some powerful new concepts to the TPM research arena, and this paper intends to highlight these for the "TPM-aware" reader in a concise package.

We have also augmented the tpm emulator by Mario Strasser as a validation exercise to also include MTM version 1 functionality. APIs and unit test interfaces are also included, and this fundament can serve as a basis for further experimentation and development in the scope of MTM evolution and/or service development on MTMs.

9 Abbreviations

A/D Analog-to-Digital

AIK Attestation Identity Key

API Application Programming Interface

DSP Digital Signal Processor

EK Endorsement Key

HMAC keyed-Hash Message Authentication Code

HW Hardware

I/O Input/Output

MLTM Mobile Local-owner Trusted Module

MRTM Mobile Remote-owner Trusted Module

MTM Mobile Trusted Module

MTSS Mobile TCG Software Stack

OS Operating System

PCR Platform Configuration Register

RIM Reference Integrity Metric

ROM Read-Only Memory

RTE Root of Trust for Enforcement

RTR Root of Trust for Reporting

RTS Root of Trust for Storage

RTV Root of Trust for Verification

RVAI Root Verification Authority Information

SRK Storage Root Key

TCG Trusted Computing Group

TPM Trusted Platform Module

TSS TCG Software Stack

References

[1] *TCG Mobile Reference Architecture*, Specification v. 1.0, revision 1, 12 June 2007, https://www.trustedcomputinggroup.org/specs/mobilephone/tcg-mobile-reference-architecture-1.0.pdf

[2] *TCG Mobile Trusted Module*, Specification v. 1.0, revision 1, 12 June 2007, https://www.trustedcomputinggroup.org/specs/mobilephone/tcg-mobile-trusted-module-1.0.pdf

[3] *Texas Instruments M-shield*, http://focus.ti.com/general/docs/wtbu/-wtbugencontent.tsp?templateId=6123&navigationId=12316&contentId=4629

[4] *Mario Strasser: TPM emulator, software implementation,* https://developer.berlios.de/projects/tpm-emulator/

[5] *GNU MP*, GNU Multiple Precision Arithmetic Library, http://gmplib.org/

[6] *TrouSerS*, TCG TSS implementation, http://trousers.sourceforge.net/

[7] *Virtual Trusted Platform Module*, IBM Research, software and architecture, www.research.ibm.com/ssd_vtpm/

