Trust in a Small Package

Minimized MRTM Software Implementation for Mobile Secure Environments

Jan-Erik Ekberg
Nokia Research Center
Helsinki, Finland
jan-erik.ekberg@nokia.com

Sven Bugiel
Nokia Research Center
Helsinki, Finland
sven.bugiel@nokia.com

ABSTRACT

In this paper we present a software-based implementation of a Mobile Remote Owner Trusted Module, using security extensions of contemporary System-On-Chip architectures. An explicit challenge are the constrained resources of such on-chip mechanisms. We expose a software architecture that minimizes the code and data size of the MRTM, applying some novel approaches proposed in recent research. Additionally, we explore alternatives within the specification to further optimize the size of MTMs. We present an analysis of specific new security issues induced by the architecture. Performance figures for an on-the-market mobile handset are provided. The results clearly indicate that a software-based MRTM is feasible on modern embedded hardware with legacy security environments.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*; L.4.0 [**Security/Trust**]: Security and Trust; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection*

General Terms

Security

Keywords

Mobile phones, mobile trusted module, secure hardware, trusted computing, platform security

1. INTRODUCTION

A Mobile Trusted Module (MTM) is conceptually a Trusted Platform Module (TPMv1.2) as defined in the Trusted Computing Group (TCG) specifications [20]. MTM, however, differs in some aspects from the design of a TPM [8]. For example, MTM mandates only a minor subset of the TPM commands, but on the other hand includes extra commands and control specifications needed for mobile phone specific use cases such as secure boot.

The TCG Mobile Phone Work Group (TCG MPWG) defines in its specification [17] two interleaving profiles for a MTM. Depending on the owner entity, the specification distinguishes between a Mobile Local Owner Trusted Module (MLTM) and a Mobile Remote Owner Trusted Module (MRTM). The local owner is the device user, i.e. the one with physical access to the device. A remote owner is a stakeholder without physical access to the deployed device. Such a remote owner may e.g. be a device manufacturer or a network service provider. To support several local and remote stakeholders, the reference architecture [18] presented by the TCG MPWG provides the opportunity of deploying parallel MTMs, with different owners, on one device. In this manner, every local and remote party has the possibility of ensuring the integrity and confidentiality of their own data on the mobile device.

To enable the parallelism and to make MTM deployable on contemporary hardware, the specification does not require the functionality to be implemented as a discrete hardware component. Instead, the MTM specification introduces new *trust roots* that mandate security properties in terms of isolation and integrity. These can still be met by implementing MTM as a separate chip, but as we will see, also processor security architectures like ARM TrustZone [4] or Texas Instruments M-shield [16, 14] can be used to meet these targets.

Further, the MTM supports *secure boot* in addition to *trusted boot*. This improvement accounts for the bulk of the new functionality introduced with MTM, compared to the TPMv1.2. In short, MTM mechanisms can be used to abort the boot-up sequence if required conditions in terms of system state are not met. This is consistent with the modus operandi with embedded devices in general, and especially with devices like handsets, on which regulatory and legislative requirements (and as a consequence liabilities) are imposed.

This paper reports on the activity of implementing the remote owner's M(R)TM logic on a contemporary handset platform with the TI-M-shield security architecture. To meet the trust root requirements, the code has to be run in an interleaved manner on System-On-Chip (SoC) secure memory, of which around 7kB (for code and data) is available at a given instance. For this reason, the implementation of the MRTM is divided into parts (collections), individually minimized in terms of code and data size. Additionally, we carefully consult the specification for options and approaches that are within the specification but still allows us to achieve the smallest possible footprint.

The main research contribution of this paper is the overall architecture by which the MTM footprint is adjusted to meet contemporary secure SoC environments, i.e. by which the MTM functions are run within on-chip memories and isolated by hardware features provided by the respective Trusted Execution Environment (TrEE). We also provide performance measurements for our implementation. To the best of our knowledge there exists no comparable implementation of an MTM today.

The rest of this paper is organized as follows. Section 2 examines the state-of-the art in mobile trusted computing, and presents a few architectural options deployed in this work. In section 3 we present the mapping between MTM trust roots and the underlying security platform. In section 4 we list a few typical use cases for an MRTM. These use cases form the context for which we make the implementation and decide on optimizations. An overview of our architecture is presented in section 5. In sections 6 and 7 we discuss optimizations made on MRTM internal structures and commands. Sections 8 and 9 provide measurements of the code size and the performance when deployed on a Nokia N96 handset. Section 10 elaborates on security threats specific to our architecture. As a side-effect of this work we have identified and reported a few errors in the MTM standard and section 11 outlines these. We conclude the paper with future work direction and conclusions.

2. RELATED WORK

Publicly available implementations of TCG technology include the TPM emulator by Mario Strasser [15] and the adjunct MTM emulator by Nokia Research Center Helsinki [9]. Both of these are provided as PC client applications and their binary size and heap consumption far exceed the constrained resources of our target devices.

Pioneering work for introducing MTMs as a platform concept includes [22], which describes an MTM in a kernel-level isolation domain, achieved by deploying Linux with (a slightly) extended SELinux.

An alternative approach to reach isolation is to deploy virtualization by means of a hypervisor over trusted hardware. In [13], Schmidt *et al.* present a virtualization-based deployment design based on the TCG MPWG reference architecture. A similar example is IBM's vTPM [5], based on Strasser's work, that implements virtual TPMs on desktop machines by means of a modified XEN hypervisor [3] and DRTM-enabled technologies like Intel's VT-x or AMD's Pacifica. Other approaches based on virtualization with XEN and ARM processors include [1] and [10]. All of the above mentioned approaches are susceptible e.g. to simple hardware attacks like memory-bus eavesdropping/injection.

Recently, the Institute for Applied Information Processing and Communications (IAIK) of the Technical University Graz published two approaches for MTM architectures, based on a software-based MTM on top of already deployed security hardware. Hence, these are very similar to our solution. The approach by Dietrich [6] focuses on Java Platform Micro Edition (J2ME) platforms. The MTM is implemented as an applet for an (on board) JavaCard runtime environment and provides support of trusted computing functionality for Java applications on mobile devices by exploiting the security features of the environment. Alternatively, Winter provides in [21] his solution for a Linux-based MTM implementation based on ARM TrustZone, relying especially on its virtualization capabilities. Winter's solution includes a virtualization framework and the paravirtualization of the Linux kernel, but as an architecture the work targets a software emulated MTM.

Our implementation requires the MTM program to be split up in smaller fragments, each a subset of the MTM commands, in order to comply with the limited resources of the secure environment. The idea of disembedding a TPM is presented in [11] by Kursawe and Schellekens. The authors describe an alternative design to current TPM architectures by externalizing larger parts of the TPM implementation, resulting in smaller trust boundaries, more agility towards specialized application requirements and reconfiguration of the TPM itself [7]. In contrast, we do not use a minimized TPM hardware as a trusted basis, but instead leverage the TrEE of the main ASIC. Our end result will differ in its security models, but the concept of "disembedding code" is rooted in the work of Kursawe and Schellekens.

A further solution in part introduced by Schellekens *et al.* [12] is to externalize the TPM state (i.e. persistent data) by extending the trust perimeter to external (insecure) memory using authenticated encryption and statefulness managed by the TPM. This is the typical and only way to manage larger quantities of volatile data within TrEEs such as M-shield and ARM TrustZone, and thus also deployed in our architecture.

3. TARGET ADAPTATION

In general terms, the MTM specification abstracts the device boot-up, and its relation to the *"Roots of Trust"* (RTs) in the following way: The initial boot step consists of an engine reset and an initialization of the trust roots, each of which describes a necessary security precondition to be satisfied in order for the MTM protection to be complete. The Root of Trust for Enforcement (RTE) asserts 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 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 could be considered to contain also the RTR.

The RTs are used to protect the initial boot sequence up to the place where the MTM is deployed. From this point onwards, the principle of booting is like in TPM where every component takes a measurement of the next component to be run. However, in MTM the measurement can be further validated against a certificate – a Reference Integrity Metric (RIM) – before the measurement is extended into the Platform Configuration Registers (PCRs). In case a verification during the bootstrap fails, the boot procedure is aborted, so it is not possible to boot into an untrusted system state.

In our target device, the native handset boot starts executing from an on-chip ROM, which in turns loads a signed bootloader and executes it on successful verification. The M-Shield architecture additionally includes on-chip RAM to be used by so-called protected applications. These can either persistently be present on an on-chip ROM, or be uploaded to on-chip RAM as signed binaries. The system implements a firewall/monitor entry point for executing these applications, and this firewall takes care of disabling or clearing all security-critical processor features (interrupts, DMA, VM) for the duration of the TrEE invocation. In this manner the system provides hardware-enforced isolation for the protected applications. As is also evident from [14], the on-chip protected applications have access to a limited amount of persistent secret data (like a device-specific symmetric key) and to cryptographic accelerator primitives. A further examination shows how these primitives can be used to fulfill the MTM RTs:

1. **RTE:** If the MTM code is run as a protected application, independently of whether it is made part of the ROM code in the secure environment or uploaded to the secure environment dynamically whereby its signature is checked, the platform asserts its integrity and authenticity. Its run-time immutability and isolation is conditional to what other applications are run as protected applications. However, the isolation of the MTM and its data from the OS (and system services) is enforced by hardware access control.

2. **RTS:** The storage requirement of MTM is easily achieved through a sealing mechanism making use (of a derivation) of the device secret key. The cryptographic primitive used for sealing should provide both confidentiality and integrity, as well as statefulness. Run-time statefulness can be asserted by the M-shield alone, between boot-ups some external secure memory, counter, or clock is needed to guarantee statefulness of the RTS (within the needs of the MTM specification).

3. **RTR:** The reporting trust root defines that the secret (private key) used to make attestation is kept secure. This is easily achievable using the RTS.

4. **RTV:** The correctness of the initial measurement of the MTM (provided by the RTV) implies that the RTV must be part of the secure boot chain. E.g. a setup where the MTM code + state is uploaded to the secure environment as a part of the bootloader, and where the initial measurement is measured and transferred to the activated MTM protected application from the same bootloader, can be considered secure, since the bootloader itself is verified prior to being activated.

In short, implementing the MTM on a processor with a TrEE conceptually satisfies the high-level security requirements of the MTM specification. A similar solution could be envisioned on architectures with secure boot (or late-launch validation like DRTM) combined with virtualization. For these setups the MTM code size and memory usage is no longer a pressing issue, but the protection of the MTM core against data eavesdropping/modification on the memory bus is. Memory bus attacks are easy enough to mount to be practical, especially if the reward may e.g. result in unlocking a locked device. The need for awareness in this matter can for example be seen in the selected MRTM use cases, described in the following.

4. MRTM USE CASES

We see the MRTM primarily as a vehicle for the integrator/manufacturer to realize the software setup of a trusted personal device. This notion is backed up by the specification itself – details like the optional omission of most management functions resonates well with the fact that any needed set-up of a software component can be done during manufacturing. Also, the optionality of both TPM/DAA and privacy CA support can be leveraged since there is little need for privacy when it comes to local software management – the user's privacy is not exposed or otherwise threatened by this activity. In this spirit we identify three main use cases for the validation of our adaptation of the MRTM:

4.1 Secure boot with stakeholders

A basic, signature-based secure boot chain is a working solution for locking software to a specific device. However, for integration management this is cumbersome – typically many stakeholders may contribute to the overall firmware of a device (some write user applications, other ones do communication stacks, drivers, etc). Today, the key management and software signing is in practice handled by the main integrator. As a result, all software patches and updates will also have to first gathered and signed by the integrator before being released to the field. The MRTM secure boot system with accompanying RIM certificates and verification keys provide the means for the different device stakeholders to operate more independently from each other (sign their code themselves) as the overall orchestration of the trust in the boot sequence can be handled by MTM in a standardized fashion.

4.2 Secure storage for applications

Even in securely booted devices applications typically are not provided with secure persistent flash storage for the application's own purposes. The MRTM specification, even in a minimal configuration, provides the tools to seal and bind keys and other information to the platform. Again, the service itself is not new and can e.g. on many handsets be realized in proprietary ways. However, an API for MTM might be a way for the industry to agree on a common interface for persistent secure storage of sensitive material like cryptographic keys and credentials. Whether this service is in the end visible through the MRTM or an accompanying MLTM is another matter.

4.3 Remote attestation

Attestation, even in a non-private form, may be important e.g. for platform software updates and licensing schemes. Clearly, attestation is one of the strong use cases of TCG specifications as a whole and may find its use in the mobile domain as well.

5. ARCHITECTURE AND IMPLEMENTATION

In this section, we present the architecture and implementation of our MRTM on an ARM 9 processor platform with TI M-Shield. More generally, our implementation requires technology that provides

1. ROM to store the program code or a mechanism by

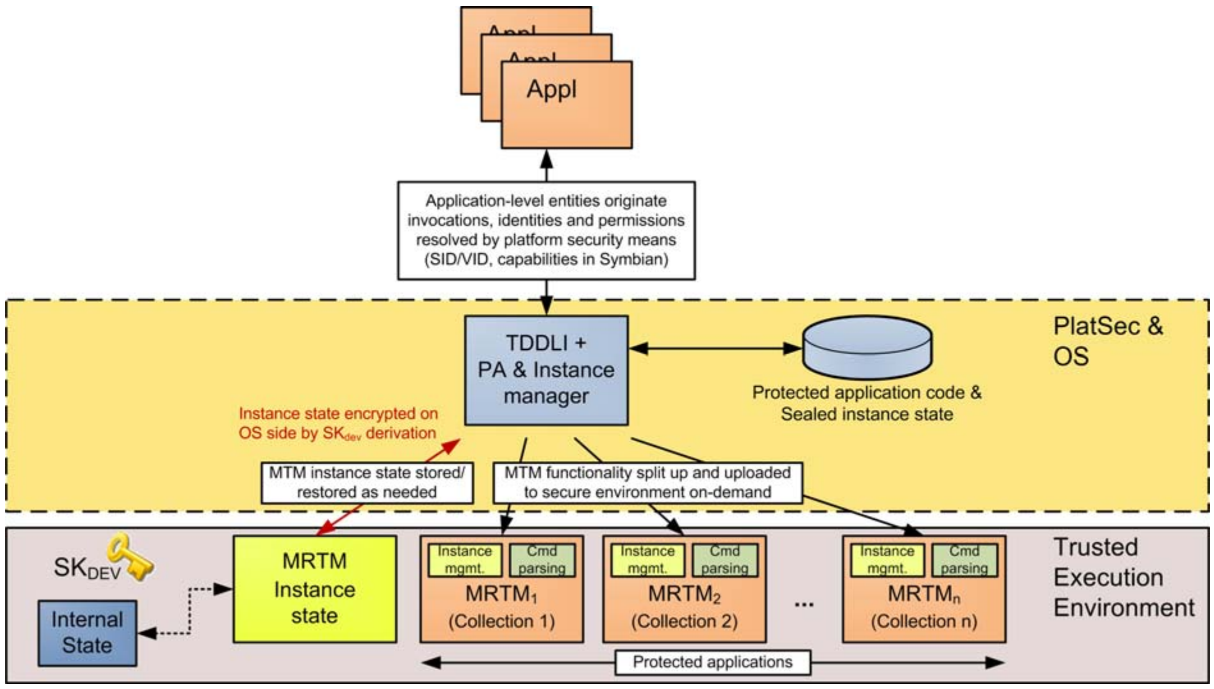
Appl
Appl
Appl
Application-level entities originate invocations, identities and permissions resolved by platform security means (SID/VID, capabilities in Symbian)
TDDLI + PA & Instance manager
Protected application code & Sealed instance state
PlatSec & OS
Instance state encrypted on OS side by SKdev derivation
MTM instance state stored/ restored as needed
MTM functionality split up and uploaded to secure environment on-demand
SKDEV
Internal State
MRTM Instance state
Instance mgmt.
Cmd parsing
MRTM1 (Collection 1)
MRTM2 (Collection 2)
...
MRTMn (Collection n)
Trusted Execution Environment
Protected applications

Figure 1: Minimized MRTM software architecture

which the integrity of code uploaded to the secure environment can be validated (code signing).

2. a shielded location (secure RAM) for the loaded state, as well as for run-time data.
3. an isolated execution environment (TrEE) for the program code with access to the shielded data location.
4. a device-specific, persistent secret to seed the RTS. The confidentiality (access control) of the secret can e.g. be bound to the secure environment itself.
5. a simple (I/O) library for use in the isolated environment, including cryptographic primitives and random number generation necessary for a MTM.[1]

Figure 1 shows the current design for our software MRTM. Due to size constraints of the secure environment in the targeted devices, both the software and the state are "disembedded" in the spirit of [11] and [12]. We have grouped the MRTM commands by size and function into 12 collections of 1-4 commands each. Depending on the command to be executed, one of these collections is loaded into the secure environment and executed. The integrity of the code collections is maintained by the underlying M-shield security architecture, by means of digital signatures. In a similar fashion, the state for the MRTM(s) is loaded (and returned) with every command invocation. The collection code is in the current implementation responsible for the handling of the state blob, which in the current implementation is a single, confidentiality and integrity protected 2kB entity. Only run-time version-control information of the state is persistently kept inside the on-chip secure RAM. When powercycling this information is lost, old state blobs are not reusable any more, and the MRTM will restart from a fixed, known state.

[1]With the exception of randomness, this is not an absolute requirement, but one that reflects size and performance. In measurements given in this paper, e.g. SHA-1, RSA, AES are provided by the platform.

An operating system component, the Instance Manager (implemented in Symbian OS) handles the selective loading of command collections and the state into the secure environment based on the command to be executed. We argue that this extended TDDL driver [19] does not need to be part of the trusted computing base (TCB), and motivate this within the informal security analysis in section 10. This architecture by design lends itself well to multi-tasking between several M(R)TMs as well as with other security services running within the device security.

To achieve maximum portability and deployability the MRTM collections (the command implementations) are implemented in ANSI C. For minimizing the overall code size, we strictly restrict the MRTM to the mandatory commands – the architecture provides easy extendability if needed. We do not support an EK (optional for an MTM) and by preinstalling the AIK, SRK, and the Root Verification Authority Information (RVAI), assigning the loadVerificationRootKeyEnabled flag with FALSE and presetting the verified PCRs, we end up with 33 compulsory commands, whereby the MTM_SetVerifiedPCRSelection and MTM_LoadVerificationRootKeyDisable commands are dummy implementations. Many other commands can be narrowed down, e.g. by removing locality checks and dispensable logic in the context of our assumptions presented in next sections.

In order to save memory in the secure RAM, especially on the secure stack, we inline code due to high subroutine invocation costs (in terms of memory space). For the same reason (function size minimization) global variables and structures are used extensively.

Listing 1: Structure for symmetric SRK

```
// 128 bits key length
#define SRK_KEYLENGTH 16

typedef struct tdTPM_KEY_SRK {
  TPM_STRUCT_VER ver;
  TPM_KEY_USAGE keyUsage;
  TPM_KEY_FLAGS keyFlags;
  TPM_AUTH_DATA_USAGE authDataUsage;
  TPM_KEY_PARMS algorithmParms;
  TPM_SECRET usageAuth;
  UINT32 PCRInfoSize;
  TPM_PCR_INFO pcrInfo;
  BYTE symKey[SRK_KEYLENGTH];
} TPM_KEY_SRK;
// Size of TPM_KEY_SRK in bytes: 123
```

Listing 2: Structure for loaded asymmetric key

```
typedef struct tdTPM_KEY_LOADED {
    TPM_STRUCT_VER ver;
    TPM_KEY_USAGE keyUsage;
    TPM_KEY_FLAGS keyFlags;
    TPM_AUTH_DATA_USAGE authDataUsage;
    TPM_KEY_PARMS algorithmParms;
    UINT32 PCRInfoSize;
    TPM_PCR_INFO pcrInfo;
    TPM_STORE_ASYMKEY keyData;
} TPM_KEY_LOADED;
// Size of TPM_KEY_LOADED in bytes: 551
```

6. SIZE REDUCTIONS RELATED TO KEY MANAGEMENT

Public-key structures consume a lot of space, and in a default implementation they do take up the bulk of the MTM state. Consequently, optimizations to these structures are required to make the state vector more suitable for e.g. the constrained memory resources of the M-shield TrEE. In this section, we present a few optimization techniques for MRTM that we apply in order to compress the key structures and hence the overall state.

6.1 Symmetric Storage Root Key

The Storage Root Key (SRK) is used to protect data (keys and seals) locally for storage. For this kind of use there is no reasonable compatibility requirement with external devices, since only the same MTM/TPM that produces the encrypted data needs to be able to decrypt it.

On a traditional TPM, the SRK is generated as part of the TPM_TakeOwnership command invocation. According to the specification, this command defines the SRK to be a RSA key with a key length of at least 2048 bits, supporting OAEP encryption with SHA1 and MGF1 [20, Part 3: Commands, p. 23]. Even so, the SRK is used solely for TPM internal purposes – e.g. to wrap new keys.

On an MRTM, however, the support for taking ownership is optional, and clearly not motivated by the notion of remote ownership. Thus the SRK is in practice pre-installed during manufacturing. For pre-installed SRKs, the TPM specifications only require that *"[a]ll storage keys MUST be of strength equivalent to a 2048 bits RSA key or greater. The TPM SHALL NOT load a storage key whose strength less than that of a 2048 bits RSA key"*[20, Part 1: Design Principles, p. 18, l. 629–631].

One way to decrease the SRK size requirement is to use a more size-efficient asymmetric primitive like Elliptic Curve Cryptography. However, in the absence of such alternatives it is possible to implement the SRK of an MRTM as a symmetric key with a sufficient key strength, e.g. AES with a 128 bit key length [2]. As a necessary precaution, the command TPM_GetPubKey must consequently be hindered from reading the non-existing SRK public part. This is achieved by assigning FALSE to the permanent flag readSRKPub.

To exploit the memory savings resulting from introducing a symmetric SRK, the TPM_KEY data structure has to be partially adapted. We introduce a new MRTM internal data structure called TPM_KEY_SRK for a symmetric SRK with 128 bits key length, shown in listing 1. With this change, we can replace the structure TPM_KEY of size 829 bytes with a TPM_KEY_SRK of only 123 bytes. The symmetric SRK structure thus saves us 706 bytes of heap memory or 85.16% compared with an asymmetric SRK.

Deploying a symmetric SRK is not without drawbacks. By introducing a symmetric SRK, we effectively make (the optional) migration impossible. Also, an authenticated encryption mode must be used when the SRK is applied, i.e. the support for such an algorithm must be arranged. For our implementation such a mode is provided within the secure environment.

6.2 Keyslot minimization

As already mentioned, RSA keys consume high amounts of space. Even with the SRK optimization, an MTM still needs signing keys and verification keys. While accepting this fact, our code provides the option to further minimize the required space for a loaded key and also the necessary number of simultaneously loaded keys in such places where the optimizations neither endanger interface compatibility nor the security of the solution.

For a minimization of the key size we modified the command TPM_LoadKey2 (TPM_LoadKey is not required for an MRTM), such that it loads only the private part of the key together with the accompanying management information. For internal keyslots, we introduced a lightweight version of the TPM_KEY structure, the TPM_KEY_LOADED (listing 2). In it, we omit the structure member for the public key. As a result, we traded run-time for saved memory, since now whenever the public part of a loaded RSA key is required, it has to be re-computed from the private component. The savings are nevertheless notable: The TPM_KEY_LOADED structure is 278 bytes or 33.5% smaller than the TPM_KEY (551 bytes) structure.

The second approach is to limit the number of keyslots. To verify or certify a loaded key with an earlier loaded key, the MTM would require a minimum of two keyslots. But for an MRTM no user keys necessarily need to be supported. Consequently, we constrain the key hierarchy rooted at the SRK. In our implementation, we declare the (symmetric) SRK as the only possible parent key for new keys created by the command TPM_CreateWrapKey. This restricts the key hierarchy depth to one, and opens up the possibility to implement only one keyslot, since the SRK is supposed to be always present in the MRTM.

Limiting ourself to only one keyslot causes a few additional problems. The compulsory command TPM_CertifyKey in an abstract sense requires two keys to be simultaneously loaded – the one key being certified as well as the certifying key. Here, we move the computational steps related to the certified key from TPM_CertifyKey to TPM_FlushSpecific – the command used to evict the keys after operating on them. In more detail, every time a key is flushed, we check if this specific key is eligible to be certified. If it is, we create the certification information of this key according to the specification of the command TPM_CertifyKey[20, Part 3: Commands, p. 130] and store this information in the state along with the key handle and usage authorization – information that is needed for the actual certification.

The side-effect of this will be that from an interfacing perspective the following ordered command sequence has to be executed to get a key certified:

1. Load the key which shall be certified into the MRTM
2. Flush this specific key
3. Load the certification key into the MRTM
4. Call TPM_CertifyKey

The process is counter-intuitive, but it does not modify any interfaces. The memory trade-off is 156 bytes vs. the size of a second keyslot (551 bytes). The code size is not affected, because we simply move the corresponding logic from one command to another. The overall run-time increases slightly, since we generate the certification information every time an eligible key is evicted.

Verification keys are also loaded into the same keyslots inside the MRTM as TPM keys and hence the constraint to one keyslot affects them as well. The design of the MRTM implies that the sole keyslot needs to be occupied by a verification key, when a new verification key should be loaded into the MRTM (with the exception of the root of the validation hierarchy, which is validated by the RVAI). In order to maintain a verification key hierarchy that is more than one level deep, we implemented an implicit substitution of a loaded verification key for one of the next-level keys during the operation of the MTM_LoadVerificationKey. As the parent key in this instance is implicitly evicted, we added a vendor specific bit to the usageFlags of the TPM_VERIFICATION_KEY structure, that indicates whether a key can be implicitly evicted in this manner. Of course the decryption of the key data (when necessary) and verification of the child key by means of the parent key are conducted inside the secure environment before the parent key will be substituted by its descendant.

6.3 Sealing versus binding on an MRTM

As was mentioned, a storage key requirement is to be equivalent in strength to a 2048-bit RSA key. However, sealing can only be done with keys of that type. This causes a dilemma, since accommodating for even a single 2048-bit secret key in the state adds in the order of 1kB to the memory consumption. On the other hand, if we do not allow for the generation of 2048-bit TPM storage keys, the only allowed seals are such that are done with the (symmetric) SRK itself. We chose the latter option, with the following motivation:

Both sealing and binding operations are used to encrypt data asymmetrically. While the binding is performed outside the MTM with the public key of a binding or legacy key, the sealing of data is achieved inside the MTM by an invocation of the TPM_Seal command with a storage key as the only possible key type parameter. Conceptually the main difference between sealing and binding is that the sealed data includes a statistically unique value (*tpmProof*) bound to the specific MTM, device and current owner. The binding includes no such parameter.

In our MRTM implementation, we do not support (the optional) migration, i.e. all keys are implicitly bound to the MRTM. Further, for us every binding key is a "leaf key" in a hierarchy rooted at SRK. In the absence of TPM_TakeOwnership, the SRK has to be pre-installed and will never change for this MRTM. In this setting, the binding functionality is close to the sealing functionality. Both TPM_Unbind and TPM_Unseal are only possible on the same MRTM with the same owner.[2] Thus, a service may use binding as a close approximation for sealing (i.e. to use keys other than the SRK), especially as it has the possibility to "seal" with keys of length 1024 bits using the PKCS#1 v1.5 encryption scheme.

6.4 Separating keys from state

In addition to minimizing the available keyslots, a further approach involves the evacuation of the remaining permanent keys in the MTM, including the AIK and the abovementioned symmetric SRK. Wrapping of these keys is not addressed in the specifications and hence neither AIK nor SRK has a parent key to encrypt their private key when or if stored outside the MTM.

In our implementation of the MRTM, the outset is to make use of an underlying secure environment as an operating environment. We turn to this environment for the RTS anyway, and we now separate the AIK from the state – the state vector only contains a binding to the right AIK, which is separately sealed for the platform and only brought into the secure environment when needed. In essence, we partially split up the MTM state, where the AIK part is being used only with a few commands.

To provide the binding between the state and the permanent keys we overload the state parameter integrityCheckRootData to include all needed digests in an ordered sequence, i.e.

```
SHA1(SHA1(AIK) || SHA1(RVAI) || SHA1(SRK))
```

When loading any of the related keys, the calling driver will upload the needed key along with the hash values of the other two. The secure-side code will re-calculate and validate the integrityCheckRootData. If the pre-image resistance of the hash function holds, the overloading is no less secure than the original use of integrityCheckRootData with only the `SHA1(RVAI)` as its content.

6.5 Symmetric verification keys

As an option, we also support symmetric verification keys. That entails certain restrictions to the verification key chain. The specifications define, that the keyData field of a verification key *"MUST contain a cryptographic key for a cryptographic primitive of strength comparable to at least 3DES CBC-MAC. [...] IF this key is a symmetric key THEN the confidentiality AND integrity of the structure MUST be protected."* [17, p. 26, l. 1–4]. We achieve the integrity and

[2] A subtle difference is also that for sealing operation the data can be bound to a specific platform configuration, i.e. certain PCR values at the time of the sealing or some other point of time. The bind command enforces no such parameter, instead platform state enforcement can be defined for the wrapping key.

confidentiality by encrypting the verification key data with the key data of its parent key. Because the verification key structure contains either a symmetric key or the public part of an asymmetric key, the private part is not immediately available for a loaded asymmetric verification key and must be provided from outside the MTM. Clearly, the decryption of a symmetric key with an asymmetric parent key would require kilobytes of work-space in the secure environment, so we disallow asymmetric parent keys for symmetric verification keys. This constraint can be stated as "the parent key of a symmetric verification key *must* be symmetric" and implies that symmetric verification keys are only possible in the beginning of a verification key chain where the RVAI binds a symmetric verification key. The integrityCheckData of a symmetric verification key is computed as a 3DES CBC-MAC with the key of its parent.

7. OTHER SIZE OPTIMIZATIONS

7.1 MRTM internal state size minimization

In addition to the keyslots, we also minimize the rest of the state by implementing only state data that is actually used in an MRTM. Compared to the definitions in the MTM and TPM specifications we can achieve significant savings. In particular, most of the data required for changing ownership and all data related to locality is unnecessary. For an MRTM the owner is a remote party without physical access to the device, and locality is forbidden by the specification if verified PCRs are deployed. We only accommodate for two simultaneous authorization sessions[3] to the TPM — the minimum needed to support the mandatory commands – and 16 PCRs. We implement only two counters (CounterRIMProtect and CounterBootstrap) and omit the CounterStorageProtect, since it is not used by any command and a storage statefulness feature can be assumed to be present on the underlying platform. With this counter set, the command TPM_IncrementCounter is dedicated to the CounterRIMProtect and hence its implementation becomes smaller. Appendix B lists the most compressed internal (non-key) structures.

7.2 TTP for issuing internal RIM certificates

A "Trusted Third Party" with a knowledge of the verificationAuth of an MRTM is able to create internal RIM certificates for that specific MRTM. E.g. a manufacturer is most likely such an entity, if it sets up the MRTM state at the time of device manufacture. These internal RIM certificates are equivalent to those generated by the TPM_InstallRIM command on that MRTM. The system could be used e.g. to issue RIM certificates for devices in the field or to increment the bootstrap counter during firmware updates. As a consequence, this concept makes the TPM_InstallRIM command on the device superfluous.

8. CODE SIZE MEASUREMENTS

The commands and sub-routines total 17840 bytes (ARM compilation). Only 2290 bytes are required for the MRTM state. This small footprint is achieved through the structural improvements and modifications described in section 6 through 5, and the optimizations of the command logic. We believe that the code is still to most parts compliant with the MTM specification v.1.0 – if we accept the fact that unnecessary but mandatory commands are left unimplemented and that some commands must be executed in a predefined order to reach a given end goal.

Table 8 lists the sizes of the individual MRTM commands in descending order. Common subroutines, which are partly implemented as inline functions, are omitted from this table. The commands that require asymmetric cryptography clearly constitute the major part of the aggregated size, even though the cryptographic primitives themselves are provided by the environment. The command TPM_ChangeAuth "suffers" from the fact that the involved encryption key can be either symmetric or asymmetric. Further, the command involves both decryption and encryption routines. The same argument is valid for commands related to key management and sealing – all involve several cryptographic functions like key generation, (asymmetric) encryption and decryption, and operations related to digital signatures.

9. PERFORMANCE

The usability consequences of MTM performance cannot be overlooked. With the introduction of support for secure boot, the speed of the trusted module will be directly mirrored in device boot-up times. For our design, the critical issues in terms of performance are a) the invocation times of the secure environment including the internal validation of the collection and b) the penalty caused by the need for state protection implemented by the collection. Both are issues not typically present in TPMs implemented as standalone ASICs. Otherwise, the internal speed of the implementation was from the start not believed to be an issue (and our measurements confirm this fact), since the logic in our design by definition is executed at the same cycle speed as the main ASIC.

The performance of our design is measured on an off-the-shelf Nokia N96 device, running an ARM9 core at 332 MHz. The core includes the M-shield security architecture. Time measurements for our design, as well as the ones for the TPMs added for comparison, were done at the TDDL layer in the OS driver – for the use case of secure boot we believe this to be the right reference point. In the Nokia N96 the OS was Symbian, and, in other cases Linux. The TPM-enabled PCs were machines with processors in the GHz range (2.3GHz, 1.6GHz).

Table 9 summarizes our comparison findings in terms of execution time. All measurements were done as averages over 10000 trusted module invocations. Measurements of our design secured the entire MTM state using a platform-optimized AES-CBC with an embedded SHA1 integrity check. Using a size-optimized, authenticated encryption mode (AES-EAX) for state protection, bumped the baseline time for entry + exit from around 11 ms to 160 ms, clearly indicating the dominance of the state protection algorithm in terms of achieving speed. Overall, the results show that the disembedded implementation approach, using an integrated secure "domain", is competitive in terms of performance, even on a device with a comparably slow application processor. The measurements also indicate that the TPM/MTM logic itself is not the bottleneck in terms of performance, rather communication channels and in our case I/O design approaches are the likely causes for slow MTM/TPM interaction. Re-visiting the secure boot use

[3]Each session can be either object-independent (OIAP) or object-specific (OSAP).

Command	Size (bytes)	Command	Size (bytes)
TPM_ChangeAuth	1774	TPM_IncrementCounter	514
TPM_CertifyKey	1606	TPM_FlushSpecific	454
TPM_CreateWrapKey	1534	MTM_VerifyRIMCert	432
TPM_LoadKey2	1430	MTM_IncrementBootstrapCounter	410
MTM_LoadVerificationKey	1422	TPM_GetPubkey	394
TPM_Seal	1112	TPM_Extend	292
TPM_Unseal	980	TPM_ReadCounter	224
MTM_VerifyRIMCertAndExtend	898	TPM_OIAP	212
TPM_Quote	792	TPM_PCRRead	144
TPM_Sign	772	TPM_GetRandom	138
TPM_OSAP	580		

Table 1: Sizes of the implemented commands

	TPM_OIAP	TPM_OSAP	TPM_PCRRead	TPM_Extend
NRC MRTM/M-Shield	11.6ms	12.0ms	11.5ms	11.6ms
Ubuntu Linux 9.04 with 2.6.24-19-generic kernel				
Atmel AT97SC3201	12.02ms	25.00ms	11.36ms	11.29ms
Atmel AT97SC3202	35.94ms	35.99ms	35.41ms	35.42ms
Broadcom BCM5755	24.01ms	24.00ms	23.34ms	23.30ms
Ubuntu Linux 9.04 with vanilla 2.6.24 kernel				
Atmel AT97SC3201	6.01ms	20.94ms	6.11ms	8.05ms
Atmel AT97SC3202	17.99ms	23.99ms	17.26ms	17.06ms
Broadcom BCM5755	12.58ms	12.59ms	12.00ms	12.00ms

Table 2: Performance comparison

case, the VerifyRIMCert command executes in 12ms on the Nokia N96 platform, showing that the use of MTM is not prohibitively expensive, when used as part of secure boot.

10. SECURITY ANALYSIS

The disembedded approach for implementing the MTM will open up some new attack vectors when compared to a straight-forward monolithic software implementation. We informally list the identified issues and outline the applied solutions.

1. **Code integrity:** The integrity of the MRTM is governed by the RTV. In the disembedded design, the RTV extends over the lifetime of the MRTM, in that the code collection to be executed will be validated not only once, but prior to every execution.

2. **Data integrity and confidentiality:** In the MRTM implementation, the state is protected either by authenticated encryption (AES-EAX) with a platform specific key and a 16-byte random IV, or alternatively with an AES-CBC encryption with a random IV, integrity protected with an embedded SHA-1 in prefix-suffix mode [4].

3. **Data state freshness:** An MRTM-specific vector of latest-used random IVs is maintained within the secure environment. Only these IVs are acceptable for state decryption, guaranteeing state freshness.

4. **Binding session state/MRTM selection:** The state uniquely defines an MRTM instance in case an authenticated session, i.e. OIAP or OSAP, is used. The instance manager may address non-authenticated commands (like TPM_PCRRead) to the wrong instance, but for these commands, no authenticity guarantees are given anyway.

5. **Binding command and collection:** All collections are treated equal by the secure environment (they are run, if the signature validates). Thus the instance manager could potentially activate the wrong collection for a given command. By mandating (in the collection) the matching of the command parameter in the request to the invoked function set, this threat is eliminated.

6. **Completeness of RTM:** Like indicated above, our current secure environment will accept any properly signed binary. This is a common issue to many deployed security environments and the implication is that the RTM is partly undefined – the device manufacturer or integrator can in principle sign some new code for the underlying architecture that potentially violates the integrity and isolation property of the MRTM. For now, the manufacturer has to be trusted not to issue such a signed piece of code.

Also, a concern with the storage hierarchy and the sealing being based on a symmetric primitive is that the amount of data, i.e. wrapped keys and seals, directly encrypted with the SRK may grow enough to weaken the confidentiality of the SRK. This bound is algorithm-specific, and can be counterbalanced e.g. by the MRTM limiting the data encrypted with the SRK.

11. SPECIFICATION SHORTCOMINGS

While implementing our MRTM, we encountered a few logical inconsistencies in the MTM v.1.0 specification [17].

[4]The reason for including the AES-CBC is speed, since the mode is hardware-optimized on the target device.

First, when the MTM_IncrementBootstrapCounter command is called with a RIM certificate, whose integrityCheckData verifies successfully, but whose counter reference is not consistent with the platform, the specified return value is TPM_Success, even though CounterBootstrap has *not* been incremented. A more logical return error value would be, for example, TPM_BadCounter.

Second, the MTM_VerifyRIMCert computes the reference value for the integrityCheckData of a RIM certificate in a different way than the command MTM_InstallRIM – the former creates the integrityCheckData with integrityCheckSize set to zero and the latter does not. As a consequence, a RIM certificate created with MTM_InstallRIM can *not* be verified with MTM_VerifyRIMCert. Consistency can be achieved by setting the value to zero in both commands.

The findings have been reported back to the TCG MPWG.

12. FURTHER WORK

We still foresee further optimization opportunities. In the current architecture, the MRTM state (except AIK) is handled as a single encrypted and integrity-protected blob. This is inefficient, since a single command (collection) typically addresses only a small subset of the state. Additionally, a significant part of the state is/can be static. Finally, maintaining confidentiality is irrelevant for most of the information stored in the state. These aspects clearly indicate room for further optimization related to state handling, with gains to be expected both in terms of memory consumption and execution speed.

Improved or better utilized hardware support within the TrEE can further minimize the MRTM footprint. Counters can make use of persistent counters in HW, flags and PCRs could be realized as hardware registers. Immutable parts of the state can be made part of a secure ROM dedicated for the MRTM.

For devices, where strict interoperability is not an issue, further trade-offs in favor of smaller footprint can be done. One such issue is byte-ordering. Initial tests indicate that 6.4% of the compiled code in a little-endian device (ARM) is used to accommodate the specification-imposed network byte-order. By sacrificing compliance, the savings are immediately available. Also, from our key-size optimization choices presented in earlier sections it is evident that removing the support for asymmetric keys could be done with quite limited impact on functionality. This could be a choice for implementing MTMs on classes of even more limited and less expensive embedded devices. Many low-end programmable controllers today include security features like embedded keys and symmetric cryptography, but cannot still accommodate e.g. RSA within the given price range.

In order to reach production quality, the MTM code still needs to pass stringent testing and validation. If this affects code-size, the disembedding architecture allows us to adjust the collection count and thereby increase the “available code space”. The same argument allows us to later augment the command coverage of our implementation to also incorporate optional and MLTM-specific commands.

13. CONCLUSIONS

We described a software implementation of a minimal MRTM that runs in hardware-enforced isolation inside the trusted execution environment of a Nokia N96 handset. Even will all optimizations active, the code is with a few minor exceptions compliant with the MTM v1.0 specification, and as a monolithic compilation it can execute in 20kB of RAM encompassing both code and data. We achieved this small footprint by reducing the data structures to a specification compliant minimum and optimizing the command logic to comply with the highly specialized demands of an MRTM. Our architecture makes use of and adapts recent research for disembedded TPM implementations to achieve the necessary size and performance parameters. Moreover, we described the use of symmetric keying for MTM – for now as an optimization feature, but also as a possible further direction for highly specialized trusted modules in the embedded domain. We proved the viability of the selected architecture by performance measurements.

14. REFERENCES

[1] Embedded XEN. `http://sourceforge.net/projects/embeddedxen/`.

[2] Keylength.com - Cryptographic Key Length Recommendation, `http://www.keylength.com`.

[3] XEN Hypervisor. `http://xen.org/`.

[4] ARM. TrustZone-enabled processor. `http://www.arm.com/pdfs/DDI0301D_arm1176jzfs_r0p2_trm.pdf`.

[5] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[6] Kurt Dietrich. An integrated architecture for trusted computing for java enabled embedded devices. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 2–6, New York, NY, USA, 2007. ACM.

[7] Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Dries Schellekens, and Marko Wolf. Reconfigurable trusted computing in hardware. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 15–20, New York, NY, USA, 2007. ACM.

[8] Jan-Erik Ekberg and Markku Kylänpää. Mobile Trusted Module (MTM) - an introduction. `http://research.nokia.com/files/NRCTR2007015.pdf`.

[9] Jan-Erik Ekberg and Markku Kylänpää. MTM implementation on the TPM emulator. `http://mtm.nrsec.com/`.

[10] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261, January 2008.

[11] Klaus Kursawe and Dries Schellekens. Flexible μTPMs through disembedding. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 116–124, New York, NY, USA, 2009. ACM.

[12] Dries Schellekens, Pim Tuyls, and Bart Preneel. Embedded Trusted Computing with Authenticated Non-volatile Memory. In *Trust '08: Proceedings of the*

1st international conference on Trusted Computing and Trust in Information Technologies, pages 60–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Andreas U. Schmidt, Nicolai Kuntze, and Michael Kasper. On the deployment of Mobile Trusted Modules, 2007.

[14] Jay Srage and Jerome Azema. M-Shield Mobile Security Technology, 2005. TI White paper. `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`.

[15] Mario Strasser and Heiko Stamer. A Software-Based Trusted Platform Module Emulator. In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 33–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] Harini Sundaresan. OMAP platform security features, July 2003. TI White paper. `http://focus.ti.com/pdfs/vf/wireless/platformsecuritywp.pdf`.

[17] Trusted Computing Group. Mobile Trusted Module (MTM) Specification. Version 1.0 Revision 6, 26 June 2008, `http://www.trustedcomputinggroup.org/resources/mobile_phone_work_group_mobile_trusted_module_specification_version_10`.

[18] Trusted Computing Group. TCG Mobile Reference Architecture Specification. Version 1.0 Revision 1, 12 June 2007 `http://www.trustedcomputinggroup.org/resources/mobile_phone_work_group_mobile_reference_architecture`.

[19] Trusted Computing Group. TCG Software Stack (TSS). Specification Version 1.2 Level 1 Errata A, 7 March 2007, `http://www.trustedcomputinggroup.org/resources/tcg_software_stack_tss_specification`.

[20] Trusted Computing Group. Trusted Platform Module (TPM) Main Specification. Version 1.2 Revision 103, 9 July 2007, `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[21] Johannes Winter. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, New York, NY, USA, 2008. ACM.

[22] Xinwen Zhang, Onur Aciiçmez, and Jean-Pierre Seifert. A trusted mobile phone reference architecturevia secure kernel. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*, pages 7–14, New York, NY, USA, 2007. ACM.

APPENDIX

A. ABBREVIATIONS

AIK	Attestation Identity Key
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
DAA	Direct Anonymous Attestation
DRTM	Dynamic Root of Trust Measurement
EK	Endorsement Key
HW	Hardware
I/O	Input/Output
IV	Initialization Vector
MLTM	Mobile Local-owner Trusted Module
MRTM	Mobile Remote-owner Trusted Module
MTM	Mobile Trusted Module
MPWG	Mobile Phone Work Group
OS	Operating System
PCR	Platform Configuration Register
RAM	Random Access Memory
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
SoC	System-On-Chip
SRK	Storage Root Key
TCG	Trusted Computing Group
TDDL	Trusted Software Stack Device Driver Library
TPM	Trusted Platform Module
TrEE	Trusted Execution Environment
TTP	Trusted Third Party

B. COMPRESSED STRUCTURES

```
typedef struct tdTPM_PCR_ATTRIBUTES {
  BOOL pcrReset;
} TPM_PCR_ATTRIBUTES;
// Size of TPM_PCR_ATTRIBUTES in bytes: 1

typedef struct tdTPM_PERMANENT_FLAGS {
  TPM_STRUCTURE_TAG tag;
  BOOL disable;
  BOOL FIPS;
  BOOL readSRKPub;
} TPM_PERMANENT_FLAGS;
// Size of TPM_PERMANENT_FLAGS in bytes: 5

typedef struct tdTPM_STCLEAR_FLAGS {
  TPM_STRUCTURE_TAG tag;
  BOOL deactivated;
} TPM_STCLEAR_FLAGS;
// Size of TPM_STCLEAR_FLAGS in bytes: 3

typedef struct tdTPM_STANY_FLAGS {
  TPM_STRUCTURE_TAG tag;
  BOOL postInitialise;
} TPM_STANY_FLAGS;
// Size of TPM_STANY_FLAGS in bytes: 3

#define TPM_SESSIONS 2
typedef struct tdTPM_STANY_DATA {
    TPM_STRUCTURE_TAG tag;
    TPM_SESSION_DATA sessions[TPM_SESSIONS];
} TPM_STANY_DATA;
// Size of TPM_STANY_DATA in bytes: 98

#define TPM_NUM_COUNTER 1
#define TPM_NUM_PCR 16

typedef struct tdTPM_PERMANENT_DATA {
  TPM_STRUCTURE_TAG tag;
  BYTE revMajor;
  BYTE revMinor;
  TPM_NONCE tpmProof;
  TPM_KEY srk;
  TPM_COUNTER_VALUE monotonicCounter[
        TPM_NUM_COUNTER];
  TPM_PCR_ATTRIBUTES pcrAttrib[TPM_NUM_PCR];
} TPM_PERMANENT_DATA;
// Size of TPM_PERMANENT_DATA in bytes: 173

typedef struct tdTPM_STCLEAR_DATA {
  TPM_STRUCTURE_TAG tag;
  TPM_COUNT_ID countID;
  TPM_PCRVALUE PCR[TPM_NUM_PCR];
} TPM_STCLEAR_DATA;
// Size of TPM_STCLEAR_DATA in bytes: 326
```