Advanced Remote Firmware Upgrades Using TPM 2.0

Andreas Fuchs, Christoph Krauss, Jürgen Repp

Advanced Remote Firmware Upgrades Using TPM 2.0

Andreas Fuchs, Christoph Krauß, and Jürgen Repp

Fraunhofer Institute for Secure Information Technology SIT
Darmstadt, Germany
{andreas.fuchs | christoph.krauss | juergen.repp}@sit.fraunhofer.de

Abstract. A central aspect for securing connected embedded systems are remote firmware upgrades to deal with vulnerabilities discovered after deployment. In many scenarios, Hardware Security Modules such as the Trusted Computing Group's Trusted Platform Module (TPM) 2.0 are used as a security-anchor in embedded systems. In this paper, we discuss the benefits of TPM 2.0 for securing embedded systems and present a concept for advanced remote firmware upgrade of an embedded system with enforcement of Intellectual Property Rights and Privacy protection of device-resident data (i.e., data that remains on the device during the flashing process). This concept utilizes unique features of TPM 2.0. Furthermore, a prototypical implementation using a hardware TPM 2.0 and the TPM Software Stack 2.0 low-level System API is presented as a proof-of-concept.

1 Introduction

Information Technology (IT) is one of the main drivers for innovations in our everyday private and work life. Especially, the use of highly connected embedded systems is increasingly growing, e.g., to enable new (safety-critical) applications in areas such as automotive, aerospace, energy, healthcare, manufacturing, or entertainment. An example is a connected car which communicates with other cars or the infrastructure to increase traffic safety and efficiency.

However, the increased connectivity (even to the Internet) and physical accessibility of the embedded systems also introduces new threats with regard to IT security and privacy. Successful attacks may have serious consequences - even to life and limb of people.

One approach for securing embedded systems is the use of Hardware Security Modules (HSMs) as a security-anchor. An HSM provides secure storage for cryptographic keys, a secure execution environment for (cryptographic) operations, and additional mechanisms such as secure boot to ensure the integrity of the platform. One promising approach is the use of the latest version of the Trusted Platform Module (TPM) 2.0, an international open standard developed by the TCG. TPM 2.0 and the corresponding TPM Software Stack (TSS) provide a high flexibility which is of great value to cost-efficiently secure different types of embedded systems on the basis of one common standard. TPM and TSS 2.0

can be realized in different specialized profiles. For example, the TCG already specified a TPM 2.0 Automotive Thin Profile [22] which is suitable for small resource-constrained electronic control units (ECUs) in a car such as an engine control unit. For more powerful embedded systems, other profiles such as PC Client might be appropriate. Also the TPM 2.0 itself provides a high flexibility. In contrast to TPM 1.2, a TPM 2.0 can be realized not only as an additional hardware chip (which provides high security but also introduces high costs) but also as a System on Chip (SoC) or Firmware TPM. Thus, one general approach can be applied to different types of embedded systems.

A central aspect to ensure security is the support for remote firmware upgrades to deal with vulnerabilities discovered after deployment. A great challenge for remote upgrades is the protection of device-resident data, i.e., data that remains on the device during the flashing process. Device-resident data are usually large data sets which are not changed for the upgrade and contain intellectual property of the manufacturer or data created on the device during operation containing person-related data. It must be ensured that only new upgrades by the original firmware manufacturer can be installed and downgrade attacks or attempts to install malicious firmware upgrades are prevented.

In this paper, we first discuss the benefits of integrating TPM / TSS 2.0 in an embedded system. Next, we present a concept for advanced remote firmware upgrade of such an embedded system using the TPM 2.0 as a trust anchor for realizing an enhancement to the classical secure boot. Our concept additionally includes the enforcement of intellectual property rights (IPR) and privacy protection for device-resident data. As proof-of-concept, we implemented our concept for an automotive head unit using a hardware TPM 2.0 and the TPM Software Stack 2.0 low-level System API.

This paper is organized as follows. In Section 2, we present background on TPM / TSS 2.0 and related work. Our concept and prototypical implementation is presented in Section 3. Finally, Section 4 concludes the paper.

2 Background and Related Work

In the last years, the security of embedded devices has been intensively investigated. The attacks on firmware upgrades go back to hacks against consumer electronics like the Playstation Portable [1] in the mid 2000s all the way to most recent attacks against safety critical systems such as automotive ECUs [8, 16].

In order to prevent completely arbitrary firmware to run, different vendor solutions and standards have been implemented. On x86 systems UEFI introduced Secure Boot in 2013 [26]. Gaming consoles include solutions of signed software since Playstation 2 and XBox 360 [27]. IPhones and Android phones included Secure Booting features from the beginning [2,3]. Automotive ECUs provide Secure Boot via SHE and EVITA [9, 10, 25]. All these basic Secure Boot schemes, however, target primarily the integrity of the base firmware. The security of device-resident data during such firmware upgrade in terms of confidentiality

as well as integrity is not targeted by any of these solutions; especially not the binding of these against trustworthy and fresh basic firmware images.

In order to provide the security for device-resident data in our proposed scheme, the functionalities of the Trusted Platform Module (TPM) 2.0 are used. Alternative solutions such as Intel TXT, TrustZone, Security Fuse Processor, or Secure Elements [4,6,11], can usually not directly provide these feature without further extensions. Intel SGX and some GlobalPlatform Trusted Execution Environment [7,12] provide similar capabilities, but only for applications running within the shielded environment, not the Rich Operating System. The means for retaining the security of the resident data during upgrade needs to be manually implemented within each application. Alternative HSMs such as TPM 1.2, SHE or Evita [10,17,25] cannot provide the specific feature that are required within this proposed concept. Specifically, Enhanced Authorization of TPM 2.0 is required in order to provide the security guarantees for this paper.

2.1 Background on TPM 2.0 and TSS 2.0

The second iteration of the Trusted Platform Module (TPM), namely the TPM 2.0 Library Specification [18] has been released by the Trusted Computing Group (TCG) in October 2014. It provides a catalog of functionalities that can be used to build TPMs for different platforms. Since then, a so-called TPM profile has been released for PC Clients [21] in January 2015, but also a TPM profile for Automotive Thin [22] in March 2015. In June 2015, the TPM 2.0 was also approved by ISO as successor to TPM 1.2 in ISO/IEC 11889:2015 [23].

Accompanying the TPM specifications, the TCG has engaged with the specification of a TPM Software Stack (TSS) 2.0 for this new generation of TPMs. It consists of multiple Application Programming Interfaces (APIs) for different application scenarios. A so-called Feature Level API has been published for review [19] in November 2014 and is currently still under development. It targets high-level application in the Desktop and Server domain. For lower-level applications, such as embedded, the so-called System API specification [24] was released as a final version in January 2015 in conjunction with the TPM Command Transmission Interface (TCTI) for IPC module abstraction. The System API and TCTI were designed such that embedded applications could leverage on TPM 2.0 functionalities whilst minimizing requirements against the platforms. As such, they can for example be used in heap-less scenarios where only stack-based memory allocation are possible.

2.2 Difference of TPM 2.0 to TPM 1.2

Compared with TPM 1.2, TPM 2.0 is a complete rewrite. Many of the new features were not compatible with the data type and function layout of TPM 1.2. The main purpose of TPM 2.0 remained to provide Roots of Trust for Storage and Reporting. In combination with a Root of Trust for Measurement, this allows the provisioning of reliable identities, securely stored secrets, and

reporting and enforcement of software integrity. The new features of TPM 2.0 in comparison to TPM 1.2 include:

- Cryptographic Agility: Whilst TPM 1.2 was restricted to RSA $\leq$ 2048 bit and SHA1, TPM 2.0 merely provides placeholders for cryptographic algorithms that can be filled up with from the TCG's Algorithm Registry [20]. The PC-Client TPM Profile for example requires RSA, ECC, SHA1, SHA256 ...
- Symmetric Algorithms: TPM 1.2 was bound to use RSA for all encryptions. This was time consuming and inefficient. TPM 2.0 also supports symmetric cryptography such as AES and HMACs.
- Enhanced Authorization: TPM 1.2 was very limited on authorization mechanisms. In general, it used SHA1 hashes of passwords. In addition, some functions further had the possibility to include bindings to a single set of PCR values (such as key usage and sealing). With TPM 2.0 the concept of Enhanced Authorization was included, which allows the forming of arbitrary policy statements based on a set of policy commands. This provides a high flexibility requiring multiple factors to be fulfilled but also to allow different paths for policy fulfillment.
- Non-Volatile Memory: With TPM 2.0 the capabilities of the TPM's integrated non-volatile memory (NV-RAM) were enhanced. They can now be used as counters, bitmaps, and even extended PCRs.
- Many more enhancement were made. A lot of those are outline in [5].

2.3 Platform Configuration Registers

One of the core functionality of TPMs is the capability to store the status of the software/firmware that was started on a device. In order to do so, each software component – starting from the pre-BIOS as Root of Trust for Measurement – calculates a hash of the next component in the boot-chain and sends this hash to the TPM. These hashes are called configuration measurements and stored within the TPM's Platform Configuration Registers (PCRs). For space efficiency, those measurements are not stored in plain, but as a hash-chain, such that a PCR's value represents a hash-chain of the measurements of the boot-chain.

The TPM's PCRs can be used to report or protect the device's integrity. The most well-known case is the concept of Remote Attestation, where a signature over a selection of PCRs is transferred to a remote entity. This entity can then determine the current configuration of a device and assess its trustworthiness. Another use case is local attestation, where certain actions of the TPM – such as data sealing (cf. Section 2.4) – can be restricted to specific device configuration states via PolicyPCR (cf. Section 2.6).

2.4 Data Sealing

The Trusted Platform Module (TPM) 2.0 has the capability for sealing data similar to the TPM 1.2. The purpose of sealing is to encrypt data with the TPM and to ensure that only this TPM can unseal the data again. In order to unseal the data, an authentication secret can be provided or a policy session that follows the scheme for Enhanced Authorization (cf. Section 2.6) can be used.

2.5 NV-RAM Counters

A TPM comes with an internal non-volatile memory. This memory can be used to make keys of the TPM persistent but can also be allocated by applications. The set of TPM2_NV_* commands provides ways to allocate memory in so called NV-Indices and perform read and write operations.

With TPM 2.0 additional classes of NV-Indices were introduced: Extendable NV-Indices and NV-Counters. The latter can be used to define strongly monotonic 64 bit counters that may only be incremented but not directly written, reset, or decremented. In order to prevent attackers from merely freeing and reallocating an NV-counter with a smaller value, any NV-counter is initialized to the highest value that has ever been present in any NV-counter of the TPM. This way, decrementation of NV-counter is always prevented. One of the main purposes of NV-counters is the use in Enhanced Authorization via TPM2_PolicyNV against a maximal value.

2.6 Enhanced Authorization

With TPM 2.0 a new concept for authorizing functions of objects was introduced under the name Enhanced Authorization. Any object that requires authorization can either be authorized using a secret value assigned during creation (similar to TPM 1.2) or using a policy following this scheme.

Enhanced Authorization consists of a set of policy elements that are each represented via a TPM command. Currently, eighteen different policy elements exist that can be concatenated to achieve a logical *and* in arbitrary order and unlimited number. Two of these policy elements – PolicyOr and PolicyAuthorize – act as logical *or*. Due to implementation requirements, policy statement are, however, neither commutative nor distributive. Once defined they need to be used in the exact same order. In this paper, we use the following notation:

$$Policy_{abc} := PolicyX_1() \wedge PolicyX_2() \wedge \ldots PolicyX_n()$$

where $Policy_{abc}$ is the "name" for this policy, such that it can be referred to from other places and $PolicyX_i()$ describes the n concatenated TPM2 Policy commands that are required to fulfill this policy.

Out of the eighteen currently defined Policy commands of the TPM 2.0 library specification, the following four policies are utilized in our concept described in Section 3. They are now explained in more detail.

TPM2_PolicyOr The PolicyOr element allows the definition of logical *or* of up to eight policy branches. Each of these policy branches itself can be an arbitrary combination of policy elements of unlimited length. In order to satisfy a PolicyOr, one of the branches needs to be satisfied before calling the PolicyOr command. To add further branches, multiple PolicyOr elements can be combined. In this paper, we represent a PolicyOr as

$$Policy_{abc} := PolicyOr([Branch_1] \vee [Branch_2] \vee \ldots)$$

TPM2_PolicyAuthorize PolicyAuthorize allows the activation of policies *after* the definition of an object. In order to achieve this, a public key K^{pub} is registered with a policy. This policy element then acts as a placeholder for any other policy branch that is signed with the corresponding private key K^{priv}. In order to satisfy such a policy, a branch needs to be satisfied first, and then PolicyAuthorize is called with the cryptographic signature that authorizes this branch as allowed. If such a signature for the currently satisfied branch can be provided, then the PolicyAuthorize element is also fulfilled. Logically, a PolicyAuthorize can be viewed as an *or* for all branches that were signed with the corresponding private key.

$$Policy_{abc} := PolicyAuthorize(K^{pub}_{abc})$$
$$Authorization_1 := Sig(K^{priv}_{abc}, [Branch_1])$$
$$Authorization_2 := Sig(K^{priv}_{abc}, [Branch_2])$$

After signatures have been provided for certain branches, we say:

$$Policy_{abc} := PolicyAuthorize(K^{pub}_{abc}, [Branch_1] \vee [Branch_2])$$

TPM2_PolicyPCR The PolicyPCR element allows functions on an object inside the TPM to be restricted to a specific combination of PCR values and thereby software configurations. A similar but less flexible capability also existed with TPM 1.2 with the keyUsageInfo for keys and digestAtRelease for sealed data. With TPM 2.0 this capability can now be applied to any authorization policy. It can further be targeted at certain operations only. In combination with PolicyOr and PolicyAuthorize it is possible to also authorize several software versions.

$$Policy_{abc} := PolicyPCR(SoftwareVersion)$$

TPM2_PolicyNV The PolicyNV element provides the possibility to include NV-indices into the evaluation of a policy. This can be used to switch between different modes of operation, by selectively enabling and disabling certain branches of a PolicyOr and PolicyAuthorize. Furthermore, it can be used to invalidate branches of a PolicyOr and PolicyAuthorize when combined with an NV-counter, as this paper presents.

$$Policy_{abc} := PolicyNV(NVindex, operation, value)$$

An example could be the comparison of the NV index NV_{abc} against a maximum allowed value of 20. If and only if the number stored within this NV_{abc} is smaller or equal to 20 can this policy element be fulfilled. Otherwise, it would fail.

$$Policy_{abc} := PolicyNV(NV_{abc}, \leq, 20)$$

3 Remote Firmware Upgrades Retaining IPR and Privacy

A Trusted Platform Module 2.0 can be used for a multitude of application cases. One of those cases is the realization of secure remote firmware upgrades with the protection of IPR-related data material and privacy sensitive information stored in device-resident data. The remainder of this paper will focus on this use case.

3.1 Scenario

Firmware upgrades are a necessity of all software based systems to fix bugs and vulnerabilities. During such a software upgrade, not all data stored on a device shall be replaced by the incoming firmware upgrade. The two categories of data not included in a software upgrade are large unchanged data sets and data created on the device during operation.

The requirements that come along with this is that consecutive software versions need to have access to this data, whilst it needs to be stored inaccessible to malicious firmware images. Also this data must become inaccessible to old versions of the firmware after an upgrade in order to prevent so called "downgrade attacks". In a *downgrade attack*, an attacker installs an outdated version of the firmware that has known vulnerabilities in order to exploit them. Such attacks have for example been used for breaking the first generation of PlayStation Portable. The classic realizations of sealing, as employed by TPM 1.2 [17], however, is not applicable in these scenarios, because they do not allow the "updating" of referenced PCR values from the sealed blobs.

Classically, this would have to be done by allowing the updater to "reseal" the data for the state after upgrading, which requires the upgrade module to be privileged. Furthermore, it was impossible to disallow usage of the old seal for accessing the data. With the framework for "Enhanced Authorization" (EA) in TPM 2.0, it is possible to achieve this use case respecting the circumstances and requirements outlined above. In the following, the set of requirements is listed, the concept described, and a prototypical implementation outlined.

3.2 Requirements

The requirements for securing large data sets and personal information during a firmware upgrade can be summarized as follows:

1. Provide confidentiality of IPR and user data.
2. Only allow original manufacturer firmware to read / write data.
3. Prevent access by old firmware after an upgrade to new firmware.
4. Do not require "privileged mode" such as updaters to unseal the data.

3.3 Concept

Our concept for providing these requirements can be divided into three phases: Provisioning, Firmware Upgrade, and Firmware Usage.

Provisioning After provisioning of the device, the very first thing should be the definition of an NV index that represents the device's currently required minimal firmware version. This has to be done first, in order to ensure that the NV index is initialized with a value of 0 (read Zero), cf. Section 2.5 for details. This counter is initialized as a single 64 bit unsigned integer value and then incremented once in order to be readable:

$$\begin{aligned} &NV_{Version} := TPM2\_NV\_DefineSpace(counter) \\ &TPM2\_NV\_Increment(NV_{Version}) \end{aligned}$$

The IPR and user data is stored in an encrypted container or partition on the devices flash that lays outside of the actual firmware binaries. The key to this encrypted container is then sealed with the TPM to the following policy:

$$Policy_{Seal} := TPM2\_PolicyAuthorize(K^{pub}_{Manu})$$

This policy allow the manufacturer as the owner of K^{priv}_{Manu} to issue new policies in the future for accessing the stored data. For the initial firmware of version $v1$, the manufacturer will provide the following policy:

$$\begin{aligned} Sig\big(K^{priv}_{Manu}, \big[&TPM2\_PolicyPCR(Firmware_{v1}) \wedge \\ &TPM2\_PolicyNV(NV_{Version} \leq v1)\big]\big) \end{aligned}$$

The PCRs that represent the integer state of $Firmware_{v1}$ may be used until the nv index $NV_{Version}$ that represents the currently required minimal version exceeds the value of $v1$.

$$\begin{aligned} Policy_{Seal} := &TPM2\_PolicyAuthorize\big(K^{pub}_{Manu}, \\ &\big[TPM2\_PolicyPCR(Firmware_{v1}) \wedge \\ &TPM2\_PolicyNV(NV_{Version} \leq v1)\big]\big) \end{aligned}$$

Note that the signed policy cannot be part of those components of the firmware that is measured into the representing PCRs. The reason is that this would lead to a cyclic relation that cannot be fulfilled.

Firmware Upgrade Whenever a firmware upgrade is issued by the manufacturer, it will be accompanied by a newly signed policy, that (similar to the original policy) grants access to the encrypted container based on the PCR representation of that firmware. This access again is only granted until the NV index representing the minimum required version exceeds this firmware's version:

$$\begin{aligned} Sig\big(K^{priv}_{Manu}, \big[&TPM2\_PolicyPCR(Firmware_{v2}) \wedge \\ &TPM2\_PolicyNV(NV_{Version} \leq v2)\big]\big) \end{aligned}$$

This leads to $Policy_{Seal}$ being:

$$\begin{aligned} Policy_{Seal} := & TPM2\_PolicyAuthorize\big(K^{pub}_{Manu}, \\ & \big[TPM2\_PolicyPCR(Firmware_{v1}) \wedge \\ & \ TPM2\_PolicyNV(NV_{Version} \leq v1)\big] \\ & \vee \big[TPM2\_PolicyPCR(Firmware_{v2}) \wedge \\ & \ TPM2\_PolicyNV(NV_{Version} \leq v2)\big]\big) \end{aligned}$$

During the update process, the updater mode stores the firmware including its policy on the device's flash drive. However, it does not require access to the encrypted container.

A similar concept with TPM 1.2 would have required that the updater mode would have required access to the encrypted container and to reseal the secret, or the manufacturer would have to have known the secret and bind it for the new PCR values. This is one of the main benefits of this TPM 2.0 based approach.

Firmware Runtime Whenever a legitimate firmware version starts, it can unseal the necessary data and read/write to these storage areas. In order to invalidate access by outdated firmware versions, during each start, the firmware will check the currently stored minimal required firmware version inside the TPM counter and increment it to its own firmware version. This invalidates the usage of PolicyAuthorize branches for previous firmware version, since they require a lower value for the $NV_{Version}$ counter. Firmware should only perform this increment when it successfully completed its self test and started up correctly, since a recovery of the previous version is impossible afterwards. Instead the issuing of a new firmware version would be required. Of course a manufacturer may choose to issue a certain recovery firmware version, or multiple such version by, e.g., encoding those as the "odd version" vs. regular firmware as "even versions".

We write the increment of the NV counter as

$$TPM2\_NV\_Increment(NV_{Version}, v1)$$

which invalidates any older policies and thereby any outdated firmwares. This leads to:

$$\begin{aligned} Policy_{Seal} := & TPM2\_PolicyAuthorize\big(K^{pub}_{Manu}, \\ & \big[TPM2\_PolicyPCR(Firmware_{v1}) \wedge \\ & \ TPM2\_PolicyNV(NV_{Version} \leq v1)\big]\big) \end{aligned}$$

A similar concept with TPM 1.2 could only have removed the sealed blobs for older versions from the flash drive, but in case of a readout of the flash from an earlier state, there would have been no possibility to actually disable these older policies with the TPM. This is another main benefit of this TPM 2.0 based approach.

3.4 Security Considerations

The presented concept relies on the correct cryptographic and functional execution of a TPM 2.0 implementation for the encryption and correct handling of the policy tickets respectively. Furthermore, the scheme relies upon a set of specific assumptions in order to function properly:

- The private ticket signing key of the vendor must be kept confidential.
- The provisioning needs to use the correct ticket public key for verification.
- The scheme does not protect against runtime-attacks against software.

3.5 Prototypical Implementation

The described concept was implemented in a proof-of-concept demonstrator for a typical automotive Head Unit. An Intel NUC D34010WYK in combination with Tizen In-Vehicle Infotainment (IVI) [14] using Linux kernel 4.0 was chosen for this implementation. These 4th generation NUCs are one of first commercial off-the-shelf (COTS) devices equipped with a TPM 2.0 implementation. Our demonstrator is shown in Fig. 1.

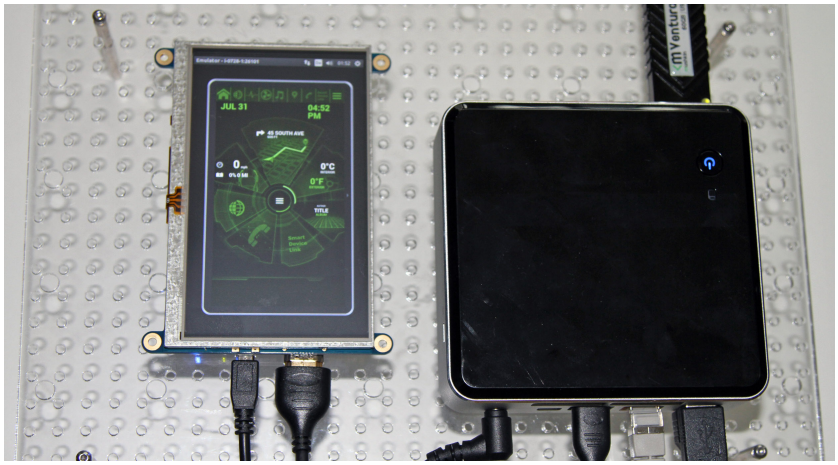

Fig. 1. TPM 2.0 Head Unit Demonstrator

To protect IPR and privacy sensitive data, the Linux LUKS implementation (Linux Unified Key Setup) is used to create and to open an encrypted container for storing this data. The key for the encrypted container in turn is protected by TPM 2.0's enhanced authorization mechanisms as described in Section 2.6.

For ease of demonstration the representation of firmware versions in the PCR values was simplified – namely not calculations of the overall firmware hashes. Instead of measuring the complete firmware at boot and extending a PCR with

this measurement, we measure a single file into the PCR called $version\_file$. A TPM monotonic counter is used to represent the version number.

Note that the standard Linux Integrity Measurement Architecture (IMA) [15] could also not be used in practice for this scenario. Due to the event-driven startup mechanisms under modern Linux systems, the exact order of PCR extension can vary, which renders them unusable for sealing. Instead, the complete image would have to be measured from within the initial ramdisk. A future publication will demonstrate a possible approach based on e.g. [13].

In Algorithm 1 and 4, we assume that the TPM 2.0 equivalent of a Storage Root Key (SRK) – a TPM Primary Key under the Storage Hierarchy – is already computed. If the NV counter NVC is already defined, it is used directly. Otherwise NVC gets defined.

Provisioning When the device is started for the first time, the following steps shown in Algorithm 1 are executed for provisioning the protected storage. It consists of the definition of the NV version counter, the instantiation of the policy, the creation of the LUKS container and the sealing of the LUKS key.

Algorithm 1 (Provisioning)

$/*$ *NV Creation* $*/$
if $not\ defined(NVC)$ ***then***
  $NVC := TPM2\_NV\_DefineSpace(counter)$
$/*$ *Policy Initialization* $*/$
$pubkey := TPM2\_LoadExternal(pub\_key\_manu)$
$policy := TPM2\_PolicyAuthorize(pubkey)$
$/*$ *LUKS creation* $*/$
$S := generate\_random\_key()$
$create\_crypto\_fs(S)$
$/*$ *Sealing* $*/$
$SRK := get\_srk()$
$enc_{SRK}(S) := TPM2\_Create(S, SRK, policy)$
$save(enc_{SRK}(S))$

To ease readability, we wrapped some of the details within simplified function calls. All functions prefixed with $TPM2\_$ are equivalent of corresponding TPM functions. $not\ defined(NVC)$ will check, whether the NV index has been defined by performing a $TPM2\_NV\_Read$ and checking the resulting value. The public key pub_key_manu of the manufacturer, loaded into the TPM, is bound to the object S encrypted by SRK via the policy calculateded by $TPM2\_PolicyAuthorize$. The corresponding private key to pub_key_manu now can be used to alter policies necessary for unsealing the encrypted key S.

Firmware release Algorithm 2 shows how the manufacturer can produce signatures for new firmware versions with a corresponding TPM 2.0 policy without using a TPM. The function $compute\_policy$ calculates the policy based on the

PCR value, after extending a PCR by the firmware digest, and the version of the firmware. This computation is performed by the manufacturer according to the TPM2.0 specification. This policy will be signed with the private key of the manufacturer. The device later must be able to associate the policy and the signature with the version to be installed.

Algorithm 2 (Firmware release)

$policy(version) := compute\_policy(digest(version\_file), version))$
$signature(version) := sign(policy(version), priv\_key\_manu)$

Upgrade Algorithm 3 shows the steps executed to install a new firmware version signed by the manufacturer. The new version will be active after the next reboot (see Algorithm 4). The version of a signed firmware to be installed must be greater or equal to the current NV counter because this demonstrator should be able to execute the provisioning process several times and the hardware TPM's monotonic counter cannot be decremented again. To be more precise, even if the counter is undefined by $TPM2\_NV\_UndefineSpace$ it is not possible to reset the counter to a smaller value because for the next definition the counter will be initialized to be the largest count held by any NV counter over the lifetime of the TPM. In practice it would be possible to store the initial policy directly in the firmware image for version 1.

Algorithm 3 (Upgrade)

$version := get\_latest\_signed\_version\_number()$
$signature(version) := get\_signature(version)$
$policy(version) := get\_policy(version)$
$save(version\_file, version)$

In order to perform the simulated upgrade of the firmware binary, we increment the value encoded within the firmware representing version file.

Runtime The steps described in Algorithm 4 are executed in the boot process to mount the encrypted container. They consist of the PCR extension with the digest of the version file, the satisfaction of the policy, using this policy for unsealing the container key, the opening of the encrypted container and then the invalidation of old firmware by incrementing the NV version counter.

Algorithm 4 (Mount encrypted file system)

/ Satisfying the policy */*
$version := load(version\_file)$
$TPM2\_PCR\_Extend(digest(version\_file))$
$approved\_policy := load\_policy(policy(version))$
$signature := load\_signature(signature(version))$
$pubkey := TPM2\_LoadExternal(pub\_key\_manu)$
$ticket := TPM2\_VerifySignature(signature, pubkey, approved\_policy)$

$session := TPM2\_StartPolicySession()$
$session.TPM2\_PolicyPCR(PCR)$
$session.TPM2\_PolicyNV(NVC, <=, version)$
$policy := TPM2\_PolicyAuthorize(pubkey, ticket, approved\_policy)$
/ Unsealing the container key and mount */*
$SRK := get\_srk()$
$enc_{SRK}(S) := load()$
$TPM2\_Load(enc_{SRK}(S))$
$S := TPM2\_Unseal(enc_{SRK}(S), session)$
$mount\_crpyto\_fs(S)$
$firmware\_self\_test()$
/ Invalidate old firmwares */*
while $TPM2\_NV\_Read(NVC) < version$ ***do***
    $TPM2\_NV\_Increment(NVC)$

The functions *load_policy* and *load_signature* will load the policy respective the signature associated by the manufacturer with the version stored in the version file. The $session.TPM2\_Policy$ function represents an execution of those policy commands on the TPM, referring to the policy session *session*. For the signed approved policy, a ticket derived from this policy and the public key of the manufacturer is computed by $TPM2\_VerifySignature$, if the signature verification is successful. The current policy value of the session will be compared with the approved policy and the TPM then validates that the parameters to $TPM2\_PolicyAuthorize$ match the values used to generate the ticket. After the crypto file system is mounted, the device should perform a self test $firmware\_self\_test$ and the NV counter will be incremented until the value which corresponds to the current firmware version is reached. Thus, the object $enc_{SRK}(S)$ can't be unsealed by firmware versions less than *version*, providing protection against downgrade attacks.

4 Conclusion

In this paper, we discussed the benefits of integrating a TPM 2.0 (and the corresponding TSS 2.0) as HSM in an embedded system and presented a new concept for advanced remote firmware upgrades of such an embedded system while additionally enforcing intellectual property rights (IPR) and privacy protection for device-resident data. This new approach is only possible by using new features introduced by TPM 2.0 such as NV-RAM counters and Enhanced Authorization. Our approach secures device-resident data by ensuring that only new firmware upgrades of the manufacturer can be installed and downgrade attacks or attempts to install malicious firmware upgrades are prevented. In our prototypical implementation for an automotive head unit, we protected device-resident data of the manufacturer (i.e., navigation maps) and of the car user (i.e., contacts and preferred navigation destinations) against unauthorized access before, during, and after an upgrade. Our general concept can be applied in different application scenarios using different TPM and TSS 2.0 profiles.

References

1. How to Downgrade a PSP, http://www.wikihow.com/Downgrade-a-PSP
2. Android: Verified boot, https://source.android.com/security/verifiedboot/
3. Apple: iOS Security. Tech. rep., Apple (2015), page 5
4. ARM Security Technology: Security technology – building a secure system using trustzone technology (2009)
5. David Challener and Will Arthur: A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security. Apress (2015)
6. Freescale: Secure Boot – For QorIQ Communications Processors (2011), http://cache.freescale.com/files/32bit/doc/white_paper/QORIQSECBOOTWP.pdf
7. Global Platforms: Trusted execution environment specifications (2011), http://www.globalplatform.org/specificationsdevice.asp
8. Greenberg, A., Miller, C., Valasek, C.: Hackers Remotely Kill a Jeep on the Highway - With Me in It (2015)
9. Henniger, O., Apvrille, L., Fuchs, A., Roudier, Y., Ruddle, A., Weyl, B.: Security requirements for automotive on-board networks. In: 9th International Conference on Intelligent Transport Systems Telecommunications (ITST 2009). IEEE (2009)
10. Hersteller Initiative Software (HIS) AK Security: SHE–Secure Hardware Extension, version 1.1 edn. (2009)
11. Intel: Trusted Execution Technology
12. Intel: Software guard extensions programming reference (2014), https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf
13. Jonathan Corbet: dm-verity, https://lwn.net/Articles/459420/
14. Linux Foundation, Tizen Association: Tizen, https://www.tizen.org/
15. Linux Weekly News: The Integrity Measurement Architecture, https://lwn.net/Articles/137306/
16. Miller, C., Valasek, C.: A Survey of Remote Automotive Attack Surfaces. In: Blackhat (2014)
17. Trusted Computing Group: TPM Main Specification, Level 2 Version 1.2, Revision 116 edn. (March 2011)
18. Trusted Computing Group: Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.16 edn. (October 2014)
19. Trusted Computing Group: TSS Feature API Specification, Family 2.0, Level 00,Revision 00.12 edn. (November 2014)
20. Trusted Computing Group: Algorithm Registry, revision 01.22 edn. (2015)
21. Trusted Computing Group: PC Client Platform TPM Profile (PTP) Specification, Family 2.0, Revision 00.43 edn. (January 2015)
22. Trusted Computing Group: TCG TPM 2.0 Library Profile for Automotive-Thin, version 1.0 edn. (March 2015)
23. Trusted Computing Group: Trusted Computing Group TPM 2.0 Library Specification Approved as an ISO/IEC International Standard (June 2015)
24. Trusted Computing Group: TSS System Level API and TPM Command Transmission Interface Specification, Family 2.0, Revision 01.00 edn. (January 2015)
25. Weyl, B., Wolf, M., Zweers, F., Gendrullis, T., Idrees, M.S., Roudier, Y., Schweppe, H., Platzdasch, H., El Khayari, R., Henniger, O., et al.: Secure on-board architecture specification. Evita Deliverable D3.2 3, 2 (2010)
26. Wilkins, R., Richardson, B.: UEFI Secure Boot in Modern Computer Security Solutions. Tech. rep., UEFI Forum (2013)
27. van Woudenberg, J.: 20 ways past secure boot, https://www.riscure.com/documents/10_ways_past_secure_boot_v1.0_jvw_shakacon.pdf