Runtime Firmware Product Lines Using TPM2.0

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. Runtime firmware product lines enable the generation of unified firmware images, i.e., a single firmware with several features can be used on several models. The device itself "decides" whether to unlock a feature or not. However, an attacker could alter their model and upgrade it to a higher-level model. In this paper, we propose an approach for secure runtime firmware product lines. Unified firmware images can be provisioned to a whole series of products while preventing unauthorized feature activation. Our approach is based on a Trusted Platform Module (TPM) 2.0, acting as security anchor using several new TPM 2.0 functionalities. The feasibility is shown in a proof-of-concept implementation.

1 Introduction

The development of Information Technology (IT) products has shifted drastically from separate and designated designs per model to unified hardware architectures with different software versions. Different products of similar kind nowadays only differ by their casing and firmware – and sometimes additional interfaces. For example, an Original Equipment Manufacturer (OEM) may produce routers and firewalls that only differ in the casing, firmware, and number of network ports. The firmware, however, differentiates a router from a firewall.

To improve the software production process, *software product lines* [11] were introduced for generating similar software versions at compile-time from a shared set of software components using conditional compilation techniques, e.g., `#ifdef` in the C programming language or flags such as `--enable-feature[=arg]` in the GNU Autoconf package [17]. *Firmware product lines* are used for generating unified firmware source code for (embedded) systems, e.g., by using the OpenEmbedded software framework [7] to generate a customized Linux distribution.

The next iteration of device product lines will go even beyond this and we refer to this step as *runtime firmware product lines*. Instead of compiling and packaging different firmware images from the same code collection, a unified firmware image will be delivered for a whole product series. The product customization will happen on the device itself. Hence, one firmware image contains several features and the device itself "decides" whether to unlock a certain feature or not. We use *overlay mounting* for the unlocking of features. This approach is particularly attractive due to its cost efficiency and simplifies the maintenance

S. De Capitani di Vimercati and F. Martinelli (Eds.): SEC 2017, IFIP AICT 502, pp. 248–261, 2017.
DOI: 10.1007/978-3-319-58469-0_17

of devices drastically. However, it poses the risk that attackers could alter their model and upgrade it to a higher-level model.

In this paper, we propose an approach for *secure runtime firmware product lines* enabling the provisioning of unified firmware images to a whole series of products while preventing unauthorized feature activation by an attacker. Our approach uses the Trusted Platform Module (TPM) 2.0 to protect the intellectual property and model feature sets while still enabling refurbishment by changing the concrete model assignment. The general idea is using the TPM as a security anchor to individually encrypt the different features and controlling access to these features using some of the new TPM 2.0 functionalities such as enhanced authorization. Using our approach, development and firmware update processes are eased drastically by enabling unified firmware provisioning for complete model series which is also shown by our proof-of-concept implementation.

2 General Idea

In our approach, a device is equipped with a TPM 2.0 that can be a dedicated hardware chip but also a software implementation to save costs. The TPM acts as security anchor by providing secure storage, secure execution, and additional features for controlling access (cf. Sect. 3.3).

The general idea of our approach is as follows. A *unified firmware image* for a product line consists of several separate read-only *feature filesystems*. Each feature filesystem contains a specific feature set and is encrypted using a unique cryptographic *feature key*. For each model of a product line, a different model number is stored inside the devices' TPM. This TPM-resident model number serves as a basis of the runtime configuration of the firmware image to be booted. By verifying a TPM-Policy, only the allowed features for a specific model of a product line are unlocked, i.e., decrypted. Note that accessing the unified firmware image is only possible on an original OEM device containing the preconfigured TPM. If required, additional read-write filesystems for local data *localdataFS* (e.g., for `/etc`, `/home`, `/srv`) and temporary files *tmpFS* (e.g., for `/var`) can be mounted. The protection of the *localdata* filesystem against manipulation is out of scope of this paper, but can be easily realized, for example, using the approach presented in [16].

Figure 1 shows an example to illustrate our approach in more detail. The unified firmware image is structured by the means of overlay read-only filesystems. A base file system *BaseFS* contains those parts that are shared between all models. It includes the operating system (OS) kernel and a multitude of basic libraries and services. Each model then comes with its feature filesystems *FeatureFS-X*. In the example, we have four of these filesystems, *FeatureFS-0* to *FeatureFS-3*. They contain the differences of this feature compared to the underlying filesystem. This may be additional files, removed files, or even altered files. For any given model, a stack can then be built that includes the base filesystem and a series of feature filesystems that are overlayed in order to provide the actual final boot system. The TPM decides, based on the model number,

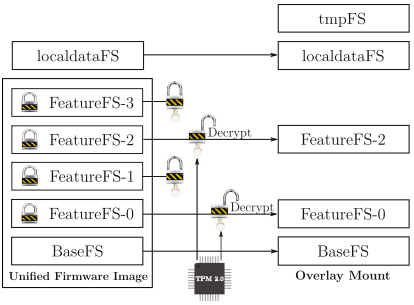

Fig. 1. General idea

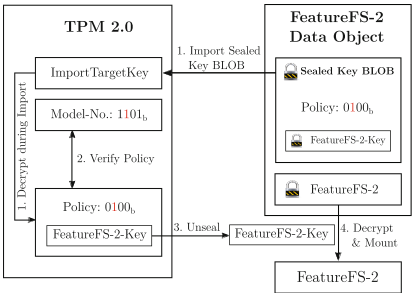

Fig. 2. Runtime feature unlocking

which features to unlock, i.e., unseal the respective *feature key* which was used to encrypt the feature filesystem to decrypt it.

The unlocking of individual features during runtime configuration is illustrated by the example shown in Fig. 2. The model number is used to denote a specific combination of encrypted feature filesystems for a model of the product line. In the example, the model number has a length of 4 bits to be able to encode the four different feature filesystems. The binary model number of 0101_b will enable the feature filesystems *FeatureFS-0* and *FeatureFS-2* to be loaded, since the respective bits were set within the model number's feature bitmask. The figure shows the unlocking of *FeatureFS-2*. As mentioned above, *FeatureFS-2* is encrypted using *FeatureFS-2-Key*. This key is encrypted using the TPM-resident *ImportTargetKey* and stored together with the *Policy:* 0100_b in an integrity-protected *Sealed Key BLOB*. The integrity of the BLOB can be verified using also the *ImportTargetKey*. The BLOB and the encrypted *FeatureFS-2* form the *FeatureFS-2 Data Object*. In step 1, the TPM imports the sealed key blob, decrypts the *FeatureFS-2-Key*, and checks the integrity of the key BLOB. The TPM verifies the policy in step 2, by checking that the third bit from the right of the policy 0100_b equals 1 (since it was also set to 1 in the model number 1101_b). If this is true, the decrypted *FeatureFS-2-Key* is unsealed and transfered to the host CPU (step 3), which uses it to decrypt and mount *FeatureFS-2* as part of the overlay mount (step 4).

In the example, the entire model number was used to encode the used features. It is also possible to define certain parts of the model number for encoding the used features. The remaining parts can be used for other purposes, e.g., to encode the color of the casing. In principle, the modeling of the model number can be arbitrary complex to realize certain policies (cf. Sect. 5.2).

3 Related Work

3.1 Secure Runtime Product Lines

The work in this paper extends the idea and concept of compile-time firmware product lines based upon unified software repositories to run-time firmware

product lines based upon unified firmware images. The most prominent example for compile-time firmware product lines are the Yocto [14], OpenEmbedded [7], and BitBake [22] projects. The BitBake project provides the build-chain and environment for a compile-time firmware product line. OpenEmbedded and Yocto provide the core and application level software recipes to build a multitude of firmware images for a multitude of devices. The Docker project [2] performs some form of packaging-time product lines by using overlay filesystems in their images similarly to our approach in order to stack feature filesystems. We use the same concept but for device boot of the basic operating system and additionally preventing activation and decryption of any unauthorized feature layer.

Another closely related case of product lines are so-called (feature oriented) software product lines, e.g., [8]. A lot of research has been conducted on software product lines [9,12,18,29] and even their security [10], practical implementations exist and given the inclusion in mainstream build systems, such as autotools [17], they are at the core of modern software engineering. The presented approach, just as the Yocto and OpenEmbedded projects, facilitate these capabilities and use them for realizing firmware product lines.

Other related technologies that are designed with a closely related focus are called feature activation. This term refers to the activation of features on devices (similar to the presented approach), but based on additional fees payed by the device owner. Though much research has been conducted on securing feature activation [24] and many more patents were filed, such as [15,20], these works have focused on the secure transfer of activation codes; i.e., on providing the evidence for an activated feature into the device. The actual securing of the feature-relevant programs on the device were to the best of the authors knowledge not yet solved. Also note that feature activation is not the focus of the presented approach and would require a rework of the model number deployment process (cf. Sect. 5.2).

Finally, it is known that many firmwares today are already partially self-reconfiguring upon startup due to their model by activating or deactivating certain software services. However, the security of these features remains yet to be solved, and a plausible approach is presented in this paper.

3.2 Overlay Filesystems

Overlay filesystems combine multiple filesystems into one single virtual filesystem and directories with equal paths are merged to one path. A typical application is the combination of read only devices (e.g., a CD) with writable devices, where all changes in the read only filesystem are stored on the writable device. This example use case could be realized under Windows with the Unified Write Filter [3]. For Linux the overlay filesystems AUFS [21] and OverlayFS [23] are available. AUFS is a reimplementation of UnionFS [6] the first available implementation of an overlay filesystem for Linux. UnionFS is also available for Free BSD and Net BSD. AUFS was the first device driver used in Docker [2] to layer Docker images and is very stable. OverlayFS is included in the mainline kernel and is

potentially faster than AUFS. Despite this fact, we used AUFS in our prototypical implementation to present our concept since the structuring of the data was more simple, and AUFS did match better the requirements for the integration of encrypted SquashFS [5] read only filesystems into the virtual overlay filesystem.

3.3 TPM 2.0

The second iteration of the Trusted Platform Module (TPM), namely the TPM 2.0 Library Specification [25] 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. Accompanying the TPM specifications, the TCG has developed 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 [26,28].

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 new features of TPM 2.0 in comparison to TPM 1.2 include:

- Cryptographic Agility: placeholders for cryptographic algorithms, e.g. RSA, ECC, AES, SHA1, and SHA256 (cf. TCG Algorithm Registry [27]).
- Support of Symmetric Algorithms: AES and HMACs.
- Enhanced Authorization: TPM 1.2 provided only limited authorization mechanisms: SHA1 hashes of passwords and binding a single set of PCR values for key use and sealing. TPM 2.0 included the concept of Enhanced Authorization, that 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 fulfillments.
- 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.
- Flexibility: the TPM can be realized as dedicated hardware chip but also in software, e.g., as a firmware TPM or software protected by a Trusted Execution Environment (TEE). In addition, TPM profiles can be defined to specify required functionalities of the TPM and TSS.
- Many more enhancement were made. A lot of those are outline in [13].

TPM 2.0 Features. In the following, those TPM 2.0 features and command relevant for the proposed solution are briefly introduced.

- **TPM Sealed Objects** The TPM 2.0 can seal data. 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 can be used. The commands used for working with sealed objects are TPM2_Create, TPM2_Import, and TPM2_Unseal.

- **TPM2_Create** This command is used to create all kinds of objects for the TPM. This includes sealed object as well as keys usable as import targets. During creation, a usage policy can be provided that restricts usage of the created object.
- **TPM2_Import** In addition to the local creation of objects inside the TPM, the TPM2_Import command further allows the external creation and importing of all kinds of objects, including keys and sealed objects. The creation of such an import blob is denoted as *create_Import*.
- **TPM2_Unseal** In order to retrieve the content of an encrypted, sealed object, the TPM2_Unseal command can be used. If the object was created or imported with a certain usage policy, this policy needs to be fulfilled for usage. This is done using so-called policy sessions.
- **TPM2_EvictControl** Since the TPM only has limited persistent internal memory, objects are usually stored externally, encrypted with a TPM-resident key. Any object can be made persistent inside the TPM on request by the owner. The TPM2_EvictControl command is used to store an object persistently in the TPM or to delete an object from persistent storage.
- **TPM NV Storage** 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. Classes of NV-Indices are Extendable NV-Indices, NV-Counters, and BitMasks. The latter can be used to individually set bits of an NV index's or to test bits within Enhanced Authorization policies.
- **TPM2_NV_DefineSpace** The TPM2_NV_DefineSpace command is used to define an NV index. Depending on the assigned index number, the NV index is either part of the user-owned (storage hierarchy) or of the platform-/OEM-owned (platform hierarchy) areas of the TPM. For the sake of this paper, only OEM-owned indices are used.
- **TPM2_NV_Read/TPM2_NV_Write** The TPM2_NV_Read/TPM2_NV_Write command is used to read/write data to a TPM NV index.
- **Enhanced Authorization** With 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 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 statements 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.
- **TPM2_StartAuthSession** In order to fulfill any authorization policy, the application needs to start a policy session using the TPM2_StartAuthSession

command. Then the actual policy statements are subsequently satisfied by invoking the corresponding TPM commands.

- **TPM2_PolicyNV** The PolicyNV element provides the possibility to include NV-indices in the evaluation of a policy. Amongst other operations, it can be used to test whether a certain bit is set or clear within the value of a specified NV-index: $Policy_{abc} := PolicyNV(NVindex, operation, value)$. For the operation of testing where the third bit $0x0004$ is set in the NV index NV_{abc} the following statement can be used: $Policy_{abc} := PolicyNV(NV_{abc}, BITSET, 0x0004)$. This policy will only succeed, if the equation $value\&0x0004 == 0x0004$ evaluates to true.
- **TPM2_PolicyNVWritten** This policy statement evaluates to true, if a given NV index has not been written to before.
- **sim_PolicyXYZ** In order to precalculate so-called policy digests – the representation of a policy used during creation – certain calculations need to be performed. To refer to these calculations, the notion of sim_PolicyXYZ is used, e.g. sim_PolicyNV.

4 Concept

Our concept for implementing *Secure Runtime Firmware Product Lines* consists of three parts: (1) configuration of the device during production, (2) creation of the firmware image, and (3) booting a model-specific firmware.

4.1 Device Production

During device production, two steps need to be taken. A key must be deployed to the device, that is used for importing the sealed key blobs, and the model type number must be stored inside the device's TPM. The following TPM commands are used to realize this:

```
// Set model number
TPM2_NV_DefineSpace(modelIdx, policyWritten, UserRead PolicyWrite)
// Create policy session
sess = TPM2_StartAuthSession(PolicySession)
TPM2_PolicyNVWritten(modelIdx, false, sess)
TPM2_NV_Write(modelIdx, modelNo, sess)
// Deploy (import) ImportTargetKey
ImportKeyBlob := TPM2_Import(ImportTargetKey)
tmpHandle := TPM2_Load(ImportKeyBlob)
TPM2_EvictControl(tmpHandle, ImportKeyHandle)
```

Model Number. The model number is stored inside the NV-storage of the TPM and can be read by anybody but cannot be altered. The number will subsequently be used to test whether a certain software feature shall be decrypted and activated at boot time for the given device. It is even possible to store the model number as part of the serial number, since the TPM-operations also allow the testing of only parts of a number stored in NV memory.

The purpose of the TPM2_PolicyNVWritten is to disallow subsequent writes by anybody to the model number NV index. Only as long as the NV index has not been written, can it be written. This also enables partial pre-production of products, where for example the board is assembled and fit into a chassis and the TPM is partially pre-provisioned but the model number is not yet set.

ImportTargetKey. The anchor for the decryption operations during boot is the ImportTargetKey. This key needs to be deployed to all devices of a product line. It is stored persistently inside the TPM for the lifetime of the device. This key is imported into the TPM and stored persistently using the TPM2_EvictControl command. When booting a firmware, the keys used for each of the features are imported underneath this key before being unsealed for actual usage during firmware decryption and activation.

4.2 Firmware Creation

Two additional steps are required for building a firmware image: the product line elements must be created by means of an overlay filesystem and the filesystems must be encrypted with a sealed TPM key bound to the corresponding model number bit. The process involving the TPM can be realized as follows:

```
firmware = createBaseImage()
for fs=overlay_1 to overlay_N do
  tmpfs = createFeatureLayer()
  fs.bitmask = createFeatureBitmask()
  key = TPM2_GenRandom()
  fs.data = encrypt(key, tmpfs)
  policyDigest = sim_PolicyNV(modelIdx, fs.bitmask, BITSET)
  fs.seal = create_Import(importTargetKey, featureKey, policyDigest)
  firmware.add(fs)
end for
release(firmware)
```

Filesystem Creation. The firmware for Secure Runtime Firmware Product Lines needs to be constructed in a specific way. For this example, we assume to have a base firmware image that is common to all models of the given product line. Note though that even different base images can be provisioned, this requires more space in the resulting firmware image. For the sake of explanation, we consider the construction of a firmware image for two models ModelA and ModelB with two mutually exclusive features FeatureA and FeatureB.

A base image is constructed as file image on the build PC. For a Linux system, this would include things such as libc, base-tools, init-system, etc. Next, a second (empty) image file is created and mounted as overlay to the base image. Then the software for FeatureA is installed into the overlay filesystem. This step can also include the rewriting or even the deletion of files. Some overlay filesystems represent deletion as entries to the 0-inode for example. When unmounting the filesystem, the differences from base image to ModelA are stored inside the

FeatureA filesystem. Now, a second file image can be created for FeatureB and the process of overlay mounting and installation can be repeated.

The approach can be further generalized by, e.g., mounting the images for base, FeatureA, and FeatureC versus base, FeatureB, and FeatureC if models differ for example in some intermediate layer, but not in the highest layer.

Filesystem Encryption. For each of the feature filesystem layers, a bitmask is created regarding which bits inside the model number represent the activation of the corresponding feature. For example, any model number with bit 0 set (i.e., uneven numbers) will include the filesystem for FeatureA. With this information, a corresponding TPM2_PolicyNV statement can be constructed that represents the test for this bit inside the model number NV index. It can also test for multiple bits or even the complete model number. This depends on the architecture for assignment of features to model numbers. The result of the policy statement creation is a policyDigest, i.e., a hash value that represents this policy.

Then a symmetric key is created that is used for encrypting the feature filesystem image. This key is then embedded within an import blob for the target TPM. This import blob is of type keyedHash, which means that it can be unsealed on the target TPM. It includes the key for the filesystem and also the policyDigest that restricts the unsealing of the key to those devices that have a model number corresponding to the policy's requirements. Finally, this import blob is encrypted using the ImportTargetKey as decryption key.

The bitmask, the encrypted key seal blob for TPM import, and the encrypted filesystem are then provided with the firmware image as a package *FeatureFS-X Data BLOB*. Note that the order of mounting the overlay images plays an important role, especially if file alterations or removals exist. This can be represented by naming of the files that contain the feature packages (if they are stored as files inside the firmware image) or by the order in which they are stored, if e.g., a partition table is used, where each package is contained inside its own partition.

4.3 Booting a Model-Specific Firmware

During firmware boot, the specific firmware for the given model of the product line is unlocked and mounted. This process uses the following algorithm:

```
current = mount(basefs)
modelNo = TPM2_NV_Read(modelIdx)
for fs=overlay_1 to overlay_N do
  if fs.bitmask == modelNo & fs.bitmask then
    seal = TPM2_Import(importKeyHandle, fs.seal)
    pSess = TPM2_StartAuthSession(PolicySession)
    TPM2_PolicyNV(modelIdx, fs.bitmask, BITSET, pSess)
    key = TPM2_Unseal(seal, pSess)
    overlay = decrypt(key, fs.data)
    current = overmount(current, overlay)
  end if
end for
```

```
current = overmount(current, localdatafs, localdata-directory)
current = overmount(current, tmpfs, runtime-directory)
chroot_and_boot(current)
```

Basic Preparations. During boot, the first step is to perform some basic operations. During this step, the basefs image is mounted and the model number is read from the TPM. This reading of the model number is not restricted in any way and can be performed by any running software. Only writing is restricted to the vendors production step.

Feature Loop. The loader will loop over all features that are provided within the unified firmware image and test for each of these, whether they are activated for the given model number. Those that are not active will be skipped. For all activated features, the loader will import the sealed feature key into the TPM under the ImportTargetKey. Then it will provide a policy session that proofs to the TPM that the policy for this sealed feature key is fulfilled by the model number stored inside the TPM. Then it will request the unsealing of the feature key from the TPM, based on the policy session that proofs the correctness of this attempt. With the unsealed feature key, the loader can decrypt the feature filesystem and mount it on top over the currently mounted system – either over the basefs or over the stack of basefs and previously mounted feature overlays.

Local Configuration and Runtime Data. Before switching into the final stack of overlay filesystems, the loader mounts two final filesystems. The runtime overlay is a temporary filesystem held only in RAM for the current boot cycle. This is necessary for the firmware in order to create sockets for local IPC connections, storing process identifiers, or to create temporary files for locking to schedule access to certain resources. The configuration overlay is an additional filesystem that contains local configuration data of the device. The reason for performing an overlay mount for this filesystem instead of a regular mount is that the feature filesystems can provide default configurations within their images. Thus, only changed configuration files are actually stored on local storage.

5 Discussion

5.1 Security

To upgrade his device to a higher-level model, an user/attacker can try the following attacks.

Attacking the TPM. The attacker can try to read out the feature keys of the TPM. The secrecy of the feature keys depends directly on the secrecy of the ImportTargetKey stored inside the TPM. An extraction of this key is unlikely if a hardware TPM certified by Common Criteria (usually using EAL4+) targeting the TCG's defined Protection Profile for TPMs is used. To further mitigate the impact of a successful key extraction from the TPM, the vendor could also rotate the import key with every production batch. This will, however, require the

vendor to provide import-blobs of the sealed feature keys for all of these rotated import keys. In order to compensate for the latter, the vendor can instead choose to deploy an additional intermediate key together with the firmware blobs. In this approach, every production batch would have its own import target key. Each firmware version would then include a specific intermediate import key that is prepared for importing under all import target keys. The sealed feature keys would then be encrypted for import under the intermediate keys. Given a realization with n import target keys, i.e., production batches, and m sealed feature keys, this would mean that each firmware blob includes one (unique) intermediate key that is packaged n times for the different import target keys and m sealed feature keys that are encrypted for the intermediate import key. This is a significant simplification compared to the $n \cdot m$ sealed feature key that would be required without an intermediate import key.

Attacking the Unsealed Feature Keys. To decrypt and mount a feature filesystem, the required feature key is decrypted by the TPM and transferred to the host CPU. An attacker could sniff on the transmission. This threat can be mitigated either by physically securing the bus between TPM and main CPU or by using the TPM protocols built-in encryption capabilities. For this, the boot-code of the device could include the public portion of the import key and use this to encrypt a salt value during session establishment. Attacks against the host CPU are not addressed by the presented approach. This includes attacks against the OS kernel or even cold boot attacks. To address the latter, mechanisms such as full memory encryption could be applied [19].

Attacking a Feature-Rich Model. An attacker could attack a feature-rich model and try to extract an unencrypted firmware image and inject it into a low-feature device. This attack would require access to the raw RAM during runtime via a software exploit. To cope with potentially unknown vulnerabilities, appropriate mechanisms for secure code update should be applied [16].

Manipulation of the Model Number. An attacker could try to change the model number to a number of a device with more features. The integrity of the model number depends on the inability of the user to write to the model number NV index. This needs to be ensured by disallowing TPM2_UndefineSpace and TPM2_UndefineSpaceSpecial with Platform-Authorization, which is supposed to be under vendor-control anyways. In order to mitigate attack potential even further, the vendor can set TPM2_NV_WriteLock on model number explicitly on each boot.

5.2 Extensions

Refurbishment of Devices. In many environments with device product lines, there exists the necessity to refurbish devices that are e.g. produced but never delivered. This can occur due to canceling of orders or because a certain hardware feature is defect that is required for a certain version of the device. In such scenarios, the vendor will refurbish the device to a new model by exchanging the

chassis or a model number sticker on the chassis and reconfigure the firmware on the device to the new device type.

In order to support this process with the presented scheme, the policy for the model number NV index on the device can be set to a policy that allows the vendor (and only the vendor) to perform write operations on this index. Such a case can be achieve using a TPM2_PolicySigned. This policy requires an external entity to sign a challenge from the TPM in order to perform a certain operation.

Model Number Attestation. Many devices nowadays are provided with the inclusion of additional services, such as cloud integration. This may, however, be a premium feature that is only activated for premium devices. The presented scheme of storing the model number inside the TPM supports such scenarios using the TPM2_NV_Certify command. Using this command, the vendor's cloud service can send a challenge to the device and the TPM will certify that the device has a given model number.

Model Numbering Schemes. The presented approach uses a very simplistic scheme for activation of feature filesystem for a given model number by querying whether certain bits in the model number are set. In addition, it is possible to extend these schemes further using a combination of AND and OR in the policy statements. For example, it is possible to require a (set of) bits to be zero using the TPM2_PolicyNV with the operation TPM_EO_BITCLEAR. These policies can then be extended using the TPM2_PolicyOR to enable a certain feature filesystem for other bit combinations as well.

6 Implementation

We implemented our concept on an Intel NUC D34010WYK equipped with a TPM 2.0 implementation running Ubuntu 16.04 with kernel version 4.4. An Apache and a Samba server were used as PLE (PLE) examples. For accessing the TPM, a TSS implementation of the TPM 2.0 System API was used together with accompanying bash tools for rapid prototyping [4].

In the device production phase, the storage root key (SRK) of the TPM is generated, and the model number is written to the NV-Storage. The firmware creation need to construct the overlay filesystems. For this purpose, we used the AUFS [21]. For every product line, an encrypted SquashFS filesystem [5] is created for overlay mounting. This filesystem is encrypted using dm-crypt [1] container files. Dm-crypt is a Linux module used for transparent disk encryption. The AES encryption keys of the encrypted SquashFS filesystem is encrypted with a private asymmetric key. The created TPM object is protected by a policy testing the flag corresponding to the PLEin the TPM's NV model number.

For booting a model-specific firmware, the mounting of these filesystems is integrated into the boot process by using initramfs which make necessary preparations before switching to the systemd init process. A shell script for the creation of the overly filesystem is added to the bottom stage of initramfs. Scripts in this stage are executed before procfs and sysfs are moved to the real rootfs and

execution is turned over to the init binary of the rootfs. The command '`mount -t aufs -o br=/r/tmps=rw:/r/02_apache:/r/ none /r/chr/`' mounts the overlay filesystem with a temporary top filesystem `/r/tmps`, the mounted Apache Squash filesystem `/r/02_apache`, and the Ubuntu base system root directory `/r` to the mount point `/r/chr`. This overlay is finally mounted over the root filesystem `/r/tmps` and the init process is started to boot the system. Since the top filesystem of the current overlay system is a temporary filesystem, all file changes of the running system will be temporary. In this example, only the Apache SquashFS is mounted because the flag for enabling the Samba container was not set. Only the key for the encrypted Apache SquashFS file system could be unsealed because the corresponding flag was set. The booted system then produces the same state as during the firmware creation after the installation of the apache packages. This also includes correct configurations, since no overrides are provided via a local configuration overlay.

A first performance analysis showed only a negligible delay in the boot process introduced by using or concept for runtime firmware product lines.

7 Conclusion and Future Work

In this paper, we propose an approach for secure runtime firmware product lines. Unified firmware images can be provisioned to a whole series of products while preventing unauthorized activation of features that belong to a different model instance. Using the features of TPM 2.0 Enhanced Authorization, Sealing, and Importing we show that this scheme can be implemented by using CotS hardware. We also present an implementation, to show the feasibility of this approach and its integration with a Linux-based system. Future work includes the extension of the presented approach to include unified images for hardware product lines, similar to the OpenEmbedded's Board Support Packages and the tight integration with the build process of unified firmware images as well as a thorough performance evaluation using different embedded platforms.

Acknowledgment. The work presented in this paper has been partly funded by the German Federal Ministry of Education and Research (BMBF) under the project "SURF".

References

1. dm-crypt: Linux kernel device-mapper crypto target. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt
2. Docker. http://www.docker.com/what-docker
3. Overlay for Unified Write Filter (UWF). https://msdn.microsoft.com/en-us/library/windows/hardware/mt571992(v=vs.85).aspx
4. Software Platform for TPM 2.0. https://www.sit.fraunhofer.de/en/tpm/
5. SQUASHFS. http://squashfs.sourceforge.net/
6. Unionfs: A Stackable Unification File System. http://unionfs.filesystems.org/
7. OpenEmbedded (2016). http://www.openembedded.org/wiki/Main_Page

8. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer, Heidelberg (2013)
9. Atkinson, C., Bayer, J., Muthig, D.: Component-based product line development: the kobra approach. In: Donohoe, P. (ed.) Software Product Lines: Experience and Research Directions, vol. 576, pp. 289–309. Springer, Heidelberg (2000)
10. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: SPL LIFT: statically analyzing software product lines in minutes instead of years. ACM SIGPLAN Not. **48**, 355–364 (2013)
11. Carnegie Mellon Software Engineering Institute Web Site: Software Product Lines (2016). http://www.sei.cmu.edu/productlines/
12. Clements, P., Northrop, L.: Software Product Lines. Addison-Wesley, Boston (2002)
13. Challener, D., Arthur, W.: A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security. Apress, New York (2015)
14. Flanagan, E.: The Yocto project. In: The Architecture of Open Source Applications, vol. 2, pp. 347–358 (2012)
15. Fountian, T.: Secure feature activation (Aug 25 2003), US Patent Ap. 10/526,252
16. Fuchs, A., Krauß, C., Repp, J.: Advanced remote firmware upgrades using TPM 2.0. In: Hoepman, J.-H., Katzenbeisser, S. (eds.) SEC 2016. IAICT, vol. 471, pp. 276–289. Springer, Cham (2016). doi:10.1007/978-3-319-33630-5_19
17. GNU: Autoconf (2016). http://www.gnu.org/software/autoconf/
18. Gomaa, H.: Designing software product lines with UML. In: SEW Tutorial Notes, pp. 160–216 (2005)
19. Henson, M., Taylor, S.: Memory encryption: a survey of existing techniques. ACM Comput. Surv. **46**(4), 1–53 (2014)
20. Ishii, A., Parthasarathy, N.: SIM-based automatic feature activation for mobile phones (Apr 23 2004), US Patent Ap. 10/830,755
21. Okajima, J.R.: Advanced multi layered unification Filesystem AUFS. http://aufs.sourceforge.net/
22. Lauer, M.: Building embedded linux distributions with bitbake and openembedded. In: Free and Open Source Software Developers' European Meeting (2005)
23. Brown, N.: Overlay filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt
24. Schramm, K., Wolf, M.: Secure feature activation. SAE Int. J. Passenger Cars Electron. Electr. Syst. **2**(2009–01-0262), 62–67 (2009)
25. Trusted Computing Group: Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.16 edn., October 2014
26. Trusted Computing Group: TSS Feature API Specification, Family 2.0, Level 00, Revision 00.12 edn., November 2014
27. Trusted Computing Group: Algorithm Registry, revision 01.22 edn. (2015)
28. Trusted Computing Group: TSS System Level API and TPM Command Transmission Interface Specification, Family 2.0, Revision 01.00 edn., January 2015
29. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Conference on Software Architecture, pp. 45–54. IEEE (2001)