Tux Armored:
Hardware Assisted Trust and Security in Linux

Dipl.-Ing. Michael Röder, Avnet Silica Poing
Dipl.-Inf. Martin Hecht, Avnet Silica Berlin

1 Introduction

The acronym IOT (Internet of Things) is probably one of the most overstressed buzz words that have been created in the past years. Many applications and use cases described as IOT innovations have been around for years, so it is safe to say that the innovation in IOT is mostly not in technology, but in the mere number of products implementing technologies such as cloud access or Smartphone connectivity. One of the positive outcomes of such immense public interest for connected devices is that lots of people also start thinking about potential side effects, some of the most important ones being security and especially data protection. In the past, for devices such as IP cameras or garage door openers, security was an afterthought. Now, that these products enter the market in big quantities and are sold even in discounter supermarkets, both public and government are alerted about potential misuse and the dangers imposed by cracking attempts to these devices. Federal agencies have started to look into criteria to be met for devices transmitting personal data over open communication channels and how to ensure the integrity of such devices.

SoCs have been equipped with cryptographic accelerators for years and some like NXP's Layerscape families, the i.MX6UL3 or i.MX8 offer amazing hardware security features by combining crypto accelerators (hardware security engines, HSM) with tamper detection features. However, these features are vendor- and part-specific, and it is hard to base a common security strategy on proprietary features.
At the same time, Trusted Platform Modules (TPM) are established more and more in embedded and industrial applications and support for TPM 2.0 in the Linux kernel has arrived for some devices. This prompts the question, to what extend TPMs can take over some of these functionalities.
This paper gives an introduction into both technologies and their advantages and disadvantages for certain use-cases.
We look into scenarios like encrypted, authenticated and measured boot over the various boot stages and the use of hardware security in the Linux Kernel and in applications such as OpenSSL or StrongSwan.
We show ways to combine hardware security technologies and software algorithms to create best-in-class solutions but also explore which hardware functionalities are currently supported in software and what is missing to create a complete, trusted solution.
Finally, we look into project-management implications of hardware assisted security, such as total cost of ownership, ease-of-use in production and into security certifications. Due to the time and page limitations, this paper can only give an introduction and completely leaves out implementation specifics and some details. Feel free to contact the authors for more details on the topics.

2 Hardware-accelerated Security

In this chapter we take a closer look at HSMs and TPMs as hardware implementations to provide security in embedded systems. We also discuss some generic advantages hardware security implementations provide over software.

2.1 Motivation

In 1995, former NSA Chief Scientist Robert Morris said*: "Systems built without requirements cannot fail; they merely offer sur-prises. Usually unpleasant!"* Some of the basic security requirements, when talking about SoC-based systems (as usually mentioned in threat analysis documents and security requirement sheets) are:

- Access Control: access and remote access to device has to be denied to unauthorized users
- Anti-Cloning: measures against overbuilding and counterfeiting of devices
- IP Protection: the manufacturer's intellectual property (e.g. software, FPGA netlists) is protected against theft
- Confidentiality: data is encrypted, especially in communication to outside world or when written to memories
- Resilience: device can detect attacks and initiate measures to protect data
- Data Integrity: the data generated or exchanged with the system is protected against modification
- Non-Repudiation: device can prove that data was generated by it and check that data arriving in it has the correct origin.

In the following chapters, we will give a short overview about how and why these requirements are addressed in recent SoC hardware security modules (HSM) and the limitations imposed by them. We will also show which functionality TPMs (TPMs) provide that can be added as a peripheral to existing systems to enhance security. We will discuss how TPMs can be used to complement or replace the integrated SoC functionality, along with the advantages and disadvantages of doing this.

2.2 HSMs (Hardware Security Modules)

At first, we take a look at the basic functionalities provided by HSM in SoC. These are as follows:

- **Trust:** measures taken and functionality provided to ensure that the system can be trusted and is untampered after boot and during operation. This includes secure and encrypted boot functionality, certified true random number generators, secure storage and use of individual keys and protection against access from the non-secure world from either software (malware) or hardware (JTAG, debugging pins). ARM Trustzone offers some basic support to isolate trusted from untrusted software parts and to restrict system access of untrusted software. Some SoCs offer far more advanced hardware features to assists software (e.g. hypervisors or secure operating systems) to separate software and restrict access to specific hardware resources using a rule based system. Trust mechanisms are crucial to provide **Access Control, Anti-Cloning, IP Protection and Non-Repudiation capabilities** to the product.
- **Crypto:** acceleration of crypto algorithms to offload the CPU and add additional security and key protection to crypto processes. This unit is usually closely linked to the units providing **Trust** and **Tamper Resistance**, but can

also be leveraged from user applications to provide **Confidentiality**, **Data Integrity** and **Non-Repudiation** at user level.

- **Tamper Resistance**: provides protection against attacks by moving the SoC out of its regular specifications. This includes units providing a secure RTC and active tamper pin monitoring along with temperature, clock and voltage monitors. The tamper unit is closely connected to all key storages in the crypto accelerators to ensure deletion of critical memories upon a tamper attempt. These units are required to provide **Resilience** and **Anti-Cloning** protection.

HSMs provide the best performance, power effectiveness and hardware costs and are ideally integrated into the SoC functionality. Therefore they can provide a comprehensive "security package" to the SoC user to leverage and are most easily used to achieve common security targets. For example, the integrated secure key memory can have a dedicated connection to the crypto unit to provide the encryption key to it without being snooped or the tamper detection unit automatically erases the secure key storage and other critical memory areas, if an attack is detected.
However, HSMs also have some disadvantages in the following areas.

- **Standardization and Reusability**: Most SoC vendors either develop their security modules as internal IP by themselves or buy third party IP which is then integrated into the SoC. There are no standardized ways how these modules are designed, integrated and used in the system scope. Therefore, a security concept and software written for one system can't be easily migrated to a different one. HSMs, drivers and usage concepts will most likely differ even among members of the same family of one vendor. This gets worse, if a common security concept has to be developed and maintained among several different platforms in a company.
- **Certification and Trust in Implementation Correctness:** this problem arises if security certifications such as Common Critera (EALn) or NIST are desired on the end product. Achieving such a certification usually requires providing a lot of material, sometimes including (semi-)formal verification reports or enabling source code / HDL review to the auditor. The end user is total dependent on the SoC manufacturer to assist in providing (and disclosing) this data, which most times will not happen. Even access to functional documentation is sometimes restricted to users or only available under NDA, which further decreases the trust level into these solutions. "Security through Obscurity" comes to mind. Unfortunately, this is not only a theoretical point, security problems in SoC hardware implementations that with an open and public implementation would probably have been detected within months, are exposed on a regular bases in such obscure implementations (last one known to the authors: https://community.nxp.com/docs/DOC-334996).
- **Ecosystem:** the complete ecosystem (software stacks, drivers, manufacturing utilities) is provided by the SoC vendor and therefore single-source and supported only by one company. Sometimes SoC vendors are hesitant to integrate complete support for their solutions into u-boot or Linux mainline to avoid exposing too much knowledge about the actual implementation so that users are still required to stick with proprietary versions.

2.3 TPMs (Trusted Platform Modules)

In opposite to HSMs which are internal to the SoC, Trusted Platform Modules (TPM) are external low-cost cryptographic modules. Trusted computing platforms may use a TPM to enhance privacy and security scenarios that software alone cannot achieve. A TPM offers the four primary capabilities **Authentication, Platform Integrity, Secure Communication,** and **IP Protection.** Depending on the version of the TPM, different cryptographic algorithms are implemented in the module. Additionally, TPMs include a small, secured non-volatile memory that can be used by user space applications to store confidential information. Other units of the TPM can be used to implement policies to manage access to this memory.

The hardware specification of TPMs is maintained by the Trusted Computing Group (TCG) [1] as a non-profit organization. The TCG also drove the specification to be accepted as international standard ISO/IEC 11889/15 which corresponds to TPM 2.0. All specifications as well as the according API are open to allow wide adoption and integration into any operating system and application software.

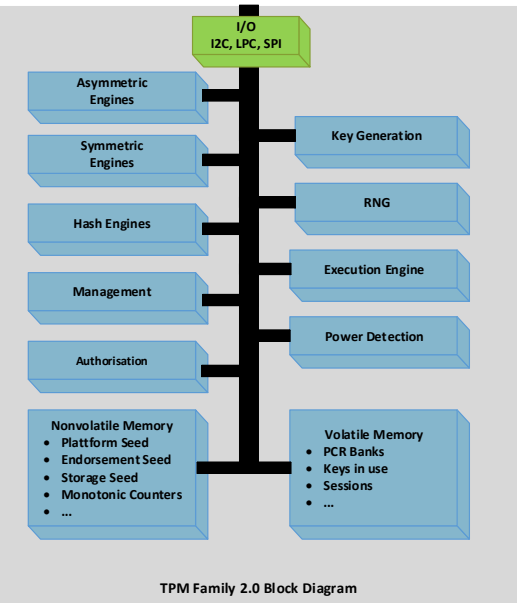
2.3.1 TPM Hardware and internal Firmware

As of today TPMs are mostly separated small hardware modules that are hardened against several forms of electrical, environmental and physical attacks to prevent that neither keys can be stolen nor the implemented cryptographic algorithms may be influenced to break the system security. Other implementations as Firmware TPM are possible.

In general, TPMs are passive components that neither measure nor monitor nor control anything directly on the system. They are physically connected by using standardized bus systems such as I2C, LPC or SPI to receive commands and responds. So they cannot influence the host system actively e.g. by stopping the execution of some kind of code on the host CPU or generating a system reset. The owner of the system even has the responsibility to manage the TPM by turning it on or off or to reset and initialize it. Unlike HSMs, TPMs are relatively slow and cannot be used as a cryptographic accelerators for encryption. The bus system is usually speed limited and depending on the particular TPM implementation some commands have execution times of several seconds. The most relevant use cases are key generation, key encryption to store keys externally, key signature and certification, and last but not least improved random number generation.

In this paper we will focus on TPM version 2.0 and skip version 1.2 for the simple reason that TPM 1.2 only implements SHA-1 as cryptographic hash algorithm. As of today there exist several serious attacks against SHA-1 [2] which leads to the conclusion that TPM 1.2 cannot be used to enhance the security of a system. Instead, in TPM 1.2 based security concepts, the TPM itself is now usually considered to be the weak point of that system. TPM 2.0 comprises all features of TPM 1.2 but with significant enhancements like an offering of several mandatory and optional algorithms instead of just one.

As shown in the right picture, a TPM comprises several hardware blocks. An important block is the **non-volatile memory**. In the production process the TPM vendor programs four individual and unique primary seeds. Three of these are permanent ones which only change when the TPM2 is cleared: endorsement (EPS), Platform (PPS) and Storage (SPS). Additionally, the TPM firmware implements a **Key Derivation Function (KDF).** A seed (which is simply a long random number) is hereby used as input to the KDF along with the key parameters and the algorithm to produce a key based on this seed. The KDF is deterministic, so if you input the same algorithm and the same parameters you will get the same key again. There's also a Null seed, which is used for ephemeral keys and changes every with every reboot, reset or power on. Seeds are never exposed by the TPM. The simple, unprotected physical connection between the TPM and CPU invites the idea to snoop on that connection to explore exported and imported keys. However, key import or export is always handled as **encrypted key blobs** (e.g. using AES) to ensure that keys generated in the TPM are protected. Using the *clear* command destroys all keys generated based on EPS, PPS and SPS along with these keys themselves.



TPM Family 2.0 Block Diagram

TPMs also contain several **monotonic, un-resettable counters** which can be used for instance to count the number of firmware updates or other events.

There also exists a special volatile memory block on each TPM. This block comprises memory locations to store keys and session information. For TPM 2.0 this block also contains two banks of 24 **Platform Configuration Registers (PCR)** that can be used for measurement, which works as follows:

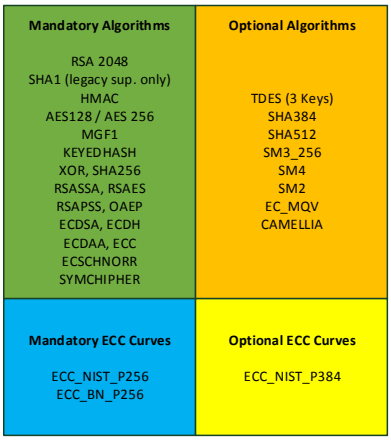
- The PCRs cannot be written directly.
- PCR 0 to 15 can only be extended by a value after an initial reset directly after power on.
- An individual reset of PCR 16 to 23 can be triggered by user.

The extend formula as calculated internally in the TPM is as shown here:

$$PCR[i]_{n+1} := hash\ (\ PCR[i]_n \,||\, [\text{extend value}]\ )$$

The index i specifies which PCR register will be extended and n is the current state of the PCR register. PCR registers are usually extended multiple times with data like sets of code, configuration data or policies to calculate a measure of this data. In other words, the measurement value of a PCR after some extensions is a measure of the code which was used as extend values. Due to the nature of a god hash function, the PCR values change significantly even for minor changes in the extend values. On the other hand, it is impossible to calculate PCR extend values to achieve a defined PCR value. Comparing PCR values in certain system states (e.g. after boot up) versus saved reference values is therefore a possibility to assess the trust state of a system.

Another important block of a TPM is the **Execution Engine** what is a small internal MCU which executes the protocol stack for host communication and controls the asymmetric and symmetric engines, key generation, key entropy checking, module self-test and other operations. An additional power detection block monitors external events like power on and is an essential part of the tamper detection.



Depending on the particular purpose of the system, the TCG publishes so called **Platform Profile specifications** to define a mandatory set of capabilities of the TPM as well as optional extensions for certain use cases. One example is the PC Client Platform TPM Profile (PTP) Specification for TPM family 2.0 (which can be used for Embedded Systems). The picture on the right shows mandatory and optional algorithms and curves for elliptic cryptography algorithms as defined in the PTP. There also exist other profiles, e.g. for Automotive and Automotive Thin Clients applications.

2.4 Why not simply use a software implementation

The following chapters will show, that for many use cases, speed and security can be highly improved when using hardware assisted cryptographic modules. However, in recent ARMv8, Intel and AMD CPUs specific instructions have been implemented that help to speed up algorithms such as AES significantly [3]. So for mere speed reasons, or if the key is exposed to parts of the software or operating system anyway, using such optimized implementations is a real option. In this paper's chapter about HSMs and TPMs on application level (chapter 4) we will also have a look at these software implementations and compare them with the ones in HSMs. However, the topic of the next chapter, establishing trust on an embedded system is a good example of a use case that highly profits from hardware implementations.

3 **Trusted Systems**

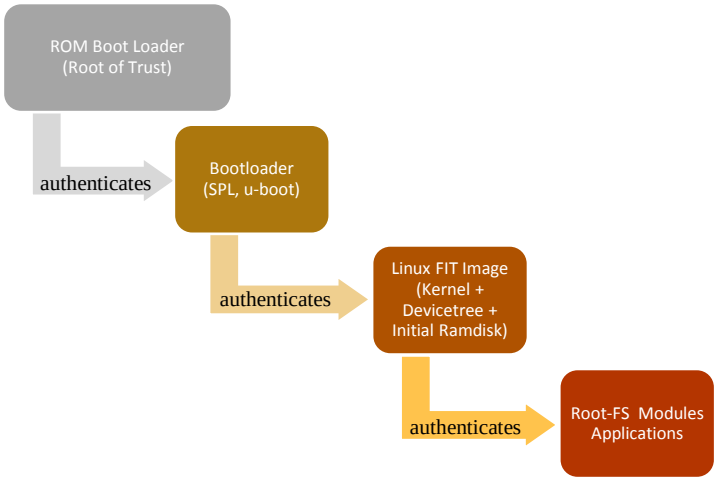
In this chapter, we take a closer look at possibilities of adding trust to embedded systems. This point is especially important for dynamically adaptable and network-connected systems (that might be attacked through this network), but also a sensible measure to ensure that system crashes or malfunctioning software is prevented from harming the system. Examples of dynamically adaptable systems are systems that support remote updates, user software installation or user customization.

When looking at trusted systems, the terms root of trust and chain of trust are important to understand so we look at these first.

3.1 Root of Trust, Chain of Trust

Similar to real live, trust in the Embedded Systems world has to be "earned". However, unlike humans who have the luxury of taking time to develop their own assessment whom to trust, in the embedded world trust has to be immediate in most cases. Therefore, a concept, called "chain of trust" is used, which is based on inheritance. It starts with an implicitly trusted component, the so called root of trust. This component then evaluates other components and decides if they can be trusted as well. If so, these components are added to the trust base, can be executed and can act themselves as new assessors of trust for other components. This way, a chain of

trust is constructed which (ideally) results in a completely assessed and trusted system. The picture below illustrates a chain of trust based on the boot process of an embedded system.



In this case, the ROM boot loader acts as the root of trust. Authentication of components (e.g. u-boot) is usually done based on hash values of the binary code of these components using either the complete component or selected (security relevant) parts of it. A root of trust is established by an unchangeable, implicitly trusted peace of code. Naturally, this code needs to be reviewed extensively for potential vulnerabilities and functional correctness and should therefore be kept as small as possible. In most SoC implementations that support hardware authentication, the root of trust is generated in the ROM bootloader (or BIOS code, if applicable). Since the complete trust of a system is inherited from the root of trust, it stands and falls with its correctness. Therefore special diligence needs to be exercised when selecting and evaluating the root of trust. In the next chapter, we will look at specific hardware implementation concepts to authenticate components and their advantages and disadvantages.

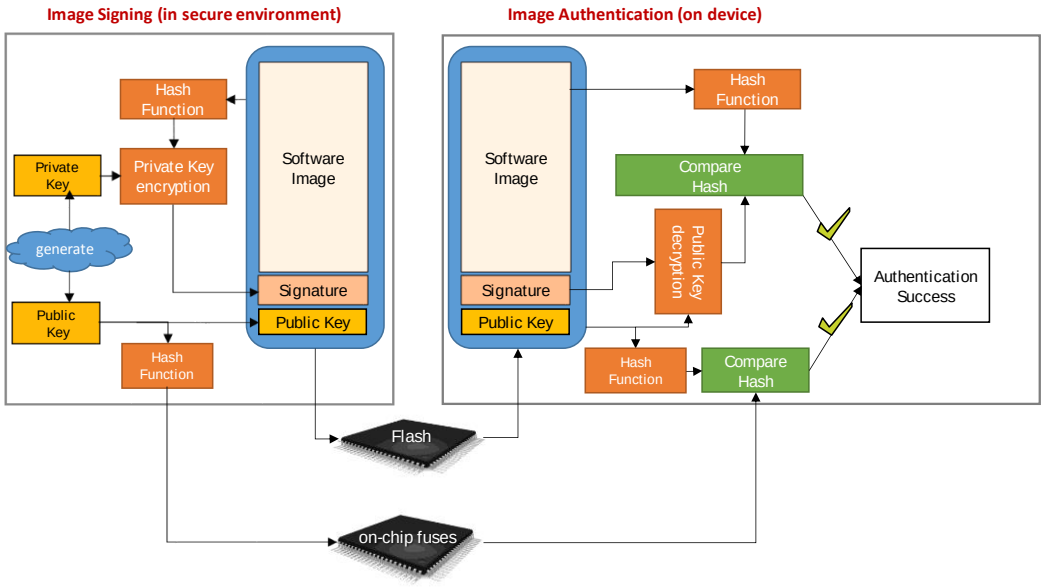
3.2 Authenticated and Encrypted Boot using HSMs

The basic task of authenticating software is very simple: generate a hash value and compare it to a reference value of this hash. If they match, the software is authenticated. However, this prompts some questions with conflicting answers:

- Where should the reference hash be stored?
  Since the reference hash is the foundation of the decision if an image is valid, it should be stored in a secure, unmodifiable location, such as in fuse arrays.
- What can I do if my software changes (updates) and the hash needs to be updated?
  To update the hash, it needs to be stored in a modifiable memory, such as flash.
- How can I dynamically store multiple hashes to validate multiple images and how can I assign a hash to an image?
  Fuse memory is usually limited and the assignment should be fixed for security reasons.

The solution to these conflicting answers is using private/public key cryptography. To sign an image for a target system, a combination of private and public key is generated during production on a development PC. This private key

is then used to encrypt the hash value and directly attached to the image along with the matching public key to verify this encrypted hash. Due to the nature of private/public key cryptography, it is very simple to verify the encrypted hash using the public key, but practically impossible to guess the private key to generate such a signature (e.g. with the updated hash value after modifying an image). To ensure that attackers can't just switch to a completely new pair of private/public key and authenticate their own images, a hash value of the public key is saved to the SoC's fuses and compared before using it. This methodology ensures that multiple images can be signed using the same private/public key with a fixed consumption of one-time programmable memory, as long as the signature of the hashes are attached to their respective images. The scheme below shows the process to sign and to authenticate an image. The orange blocks identify the cryptographic functions used on chip and host PC, while the green blocks show the actual verification checks performed on the SoC. Hashing is usually performed using SHA-256 while RSA is being used for private/public key cryptography.



To make this process of authenticating an image as attack-safe as possible, it is usually implemented in hardware, using Functional State Machines (FSMs) or small, completely isolated controllers. NXP's CAAM engine is an example of such hardware, which can perform authentication in a completely automated way. Highly simplified, after being told where the respective image to authenticate is located, it performs the authentications and then updates the chip "secure" state. If the images has been authenticated successfully, it keeps the state in "secure", otherwise it changes it to "unsecure". Once the state has been changed to "unsecure" it can't be changed back to "secure" without a hard reset. This way, a chain of trust is established. The authentication can be invoked from software (e.g. ROM bootloader or u-boot) by jumping into functions located in the SoC's ROM and some registers mirror the current trust state.

If encryption for the images is added to the boot process, an additional step is required which involves encrypting the image key with an integrated device key, therefore generating a cryptographic blob which can also be attached to this image.

This process allows users to select their individual key, but still keeps security at a high level by encrypting this key with unique, random keys not knows to anybody and specific to the device. In other words, if two devices boot images which are encrypted with the same key, the cryptographic blobs of these keys are different, while the encrypted image itself is the same.
This process highly simplifies remote updates of images, because it allows OEMs to use the same private key and encryption key on multiple images, but still minimize the attach surface by parallelized brute force attacks.
This high level of security is achieved by using hardware security units. In pure software, it would be impossible to realize, starting with the inability of processor cores to boot encrypted code, which would imply to keep some code unencrypted and unsigned. As a consequence, this code and the keys used for accessing and verifying flash program code would have to be stored on protected, one-time programmable memories, making it expensive and hard to update and adapt to exposed security leaks. Another advantage of using hardware security for authentication is the possibility to integrate these units with other security units on the SoC. For example, the symmetric units can also be used to export/import user specific keys or user data into cryptographic blobs. Upon detected attacks or images not being authenticated, internal key memories can be cleared and access to critical IP modules can be prevented.
If this method is implemented in a functional correct way, it is the most practical and secure way of establishing a root of trust. However, it also has some disadvantages:

- **License Restrictions**: Some open source licenses (e.g. GPLv3) impose the requirement on a system that users should be allowed to completely exchange the operating system on these devices. [4] for example states: *"When it comes to security measures governing a computer's boot process, GPLv3's terms lead to one simple requirement: Provide clear instructions and functionality for users to disable or fully modify any boot restrictions, so that they will be able to install and run a modified version of any GPLv3-covered software on the system."* Since HSM assisted trusted boot cannot be disabled in hardware once enabled, measures (e.g. using an intermediate boot loader) need to be implemented to ensure that this is possible. Contact the authors for more information on this topic.
- **Trust / Certification** issues: as mentioned in chapter 2.2
- **Little flexibility for bug fixing:** Since the implementation is completely in hardware with no software interaction, it offers little flexibility of later improvement or bug fixing if errors are detected or changes have to be made due to problems in later system updates.

3.3 Measured Boot with TPM

As mentioned in the last chapter, depending on the profile specification the PCR extend functionality can be used to measure dedicated parts of code, configuration data and policies. This functionality can for example be used to measure the components involved in the boot of system. The table below shows how PCRs are defined to be used on UEFI enabled X86 systems and a recommendation how to adapt this usage on ARM systems with GNU/Linux.

PCR index	PCR usage (UEFI + x86)	PCR usage (example on ARM)
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers	Boot ROM, if accessible First Stage Boot Loader
1	Host Platform Configuration	
2	UEFI driver and application Code	
3	UEFI driver and application Configuration and Data	
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts	u-boot (including SPL)
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table	u-boot Environment
6	Host Platform Manufacturer Specific	
7	Secure Boot Policy	Linux IMA
8-15	Defined for use by the Static OS	Linux IMA
16	Debug	

It is important to note, that all software code performing measuring operations (with or without using TPMs) has to be authenticated to be part in the chain of trust, otherwise the measurements performed by it can't be trusted. TPMs by themselves are not able to establish a root of trust, since they need to be triggered by some software code (unlike HSMs). So replacing HSM authentication with TPM measurement functionality can only start with the first code that allows the TPM to be accessed and can be authenticated by outside means (HSMs). Another important thing to know is that PCR extends of big chunks of code usually implies a big time penalty due to the limited operating speed of TPMs. Therefore, most of the times the preferred approach would be to authenticate a software algorithm (possibly utilizing trusted HSMs) to calculate a hash of the chunk in question and then extend the TPM PCR with the result of this calculation.

No public implementation to support measured boot across all boot stages exists to the knowledge of the authors. So in the remainder of this subchapter, we will outline the current state of TPM support in components of the boot chain.

Naturally, the ideal place to start TPM measurement would be the ROM bootloader implemented by SoC vendors. However, at the time of writing this, no SoC known to the author offers TPM support in the **ROM bootloader**. However, some SoC vendors offer support to modify and extend their first stage bootloader (**FSBL)** implementation, so some basic routines for TPM initialization and PCR-based measurement can be added there. If this is not possible, u-boot would be the first stage in which measurement can be started for components following in the chain of trust. This implies, that HSM-based methods have to be used before to authenticate u-boot itself and previous components in the chain of trust.

U-Boot currently only offers support for self test, provisioning, un-provisioning and PCR extension for TPM 1.2. Advanced features to support integrated measurement of payload data in u-boot and support for TPM 2.0 are missing and would have to be implemented by the user. For further questions about ongoing support on this, please contact the authors.

Starting with GNU/Linux Kernel 2.6.30 the Integrity Subsystem of the kernel has been introduced (with extensions in 3.3 and 3.7) that can be used to implement measured boot of the Kernel. Basically, it allows detection if files have been accidentally or maliciously modified, both locally or remotely. Files measurement can be appraised against a known "good" value stored as an extended attribute in the filesystem or via a server through a secure IPSEC connection ("remote attestation").

IMA uses the Extended Verification Module (EVM) to guarantee the integrity between the file and its extended attributes. IMA currently offers the following integrity functions:

- Collect measure a file before it is accessed.
- Store add the measurement to a kernel resident list and, if a hardware Trusted Platform Module (TPM) is present, extend the IMA PCR
- Attest if present, use the TPM to sign the IMA PCR value, to allow a remote validation of the measurement list.
- Appraise enforce local validation of a measurement against a "good" value, stored as hash or signature in 'security.ima' extended attribute of the file, protected by EVM
- Protect protect a file's security extended attributes (including appraisal hash) against off-line attack.

Hash values are extended into PCR10. So the final aggregate hash in PCR10 is the record of the state of the measured files and directories, for example after booting. IMA also offers some built-in policies that can be enabled on the boot command line. The kernel log also contains all information about what files have been appraised and which executables have been started. It provides the tool *evmctl* (contained in the *ima-evm-utils* package) which can be used for producing and verifying digital signatures and to store them into the xattr to be used by IMA. For further information on the configuration of IMA see the official documentation [5] or contact the authors.

3.4 Software Implementations for Root File System Security

Along with the very HSM and TPM centric techniques of hardware authentication and measurement mentioned above which have been design with hardware in mind or even rely completely on hardware, there are various methods in the GNU/Linux system to enhance security by use of software authentication and encryption. Some of these include support for hardware crypto acceleration (explicitly or implicitly through the Kernel Crypto API) and are therefore quickly mentioned in this chapter.
The Linux Kernel supports various methods to implement block level integrity protection such as DM-Verity [6] and DM-Integrity [7]. **DM-Verity** uses a cryptographic hash tree to authenticate block devices. The hashes are computed using kernel crypto services and therefore leverages HSM through the kernel's crypto API. DM-Verity can only be used to verify read-only partitions, updating or changes to these partitions therefore require a new integrity setup of the partition. It can therefore be used for system partitions which are supposed to remain unchanged.
DM-Integrity which was recently merged into Linux Mainline with Kernel 4.12, on the other hand supports the authentication of R/W partitions and supports journals. It also leverages the Kernel Crypto API and therefore profit from HSM acceleration.
DM-Crypt [8] provides support for encrypting block devices for GNU/Linux and works with both DM-Verity and DM-Integrity. **LUKS** (Linux Unified Key Setup) [9] provides some extensions to it, mostly to simplify key handling. DM-Crypt uses the Linux Kernel Cryptographic API and therefore is able to leverage HSM. The extension **tpm2_luks** [10] helps to store keys in TPM 2.0 modules and to seal access to the key to specific PCR values to ensure, that the key can only be accessed in safe measured states.

3.5 Conclusion: secure / authenticated / measured boot

In the past subchapters, we have looked at both authenticated boot using on-chip HSMs and at measured boot using TPMs to complement pure software based approaches. But what is the "ideal" method to implement trusted systems? The following table summarizes the advantages and disadvantages of both methods:

Aspect	**On-Chip HSM**	**TPM**	**pure Software**
boot speed (also see next chapter)	++	--	+ (ARMv7) to ++ (ARMv8)
can establish root of trust	+	-	-
ease of use	+	+	++
open source support	-	+	++
standardized / vendor independence	--	+	-
pre-certification	--	++	-
financial costs	+	--	++
integration into SoC security system	++	-	--
algorithm flexibility	-	+ (TPM 2.0)	++
updates / security fixes possible	--	+	++

For most of today's applications, the preference of system architects is to leverage TPMs as far as possible. The major problem with using TPMs for a complete authentication flow is the limited speed and the inability to establish a root of trust without a remote connection for attestation. This, however means that HSM-based methods have to be used anyway to complement TPMs. So a common approach is to use HSMs where they are absolutely required (root of trust) or show significant advantages (integration into SoC security system) and otherwise complement TPM's disadvantages using (TPM authenticated) software routines, which allows for higher flexibility and easier certification of the algorithm. Some recent security exposures show that this approach is the right one. NXP's high assurance boot had some security leaks, which are hard to mitigate in deployed systems. About at the same time, a problem in Infineon's SLB9670 TPM was exposed for RSA key generation. This problem could be fixed with a simple firmware update (because the TPM is running just software in a specially protected microprocessor) and took much less time to be detected due to the wide use of this TPM.

The table below shows an example flow, how an authenticated boot flow could be implemented, leveraging both HSMs and TPMs.

Step	Actions	Payload
ROM Bootloader (no changes possible)	- authenticate u-boot (including hash algorithms for TPM) using integrated HSM hardware support	u-boot (incl. TPM support)
u-boot	- Initialize TPM + measure u-boot binary in memory and extend PCR (alternatively: remote attestation) - Load FIT Image - Authenticate FIT Image using certified hash algorithm and extend PCR	FIT Image (Kernel + Devicetree + initial RAMdisk)
Linux Kernel	- Initialize HSMs + TPM support - load initial RAMdisk - enable IMA	Initial RAMdisk
Initial RAMdisk	- TPM: setup dm-integrity + dm-crypt with keys from TPM - mount encrypted, authenticated Root Partition	Root Partition Block Device
Application Level	- setup HSMs/SW engines using CryptoDev - setup access to key storages and crypto functions in TPM (protected by IMA / PCR sealing)	see next chapter

4 HSMs and TPMs on Application Level

In the last chapter we evaluated how hardware security modules are used to authenticate a system during boot and assure that it can be trusted. However, cryptography is also a common requirement on application level. Some common tasks include:

- Encrypting/Decrypting sensitive data
- Key Storage
- IPSeC tunnelling

We start with an overview, how HSMs and TPMs are used at a user space level in GNU/Linux. Then we will look into the specific support for these scenarios in more detail.

4.1 HSM on Application Level

In this chapter, we describe HSM support in GNU/Linux for kernel and userspace and show some benchmark results.

4.1.1 Hardware

The HSMs of recent SoCs) can be leveraged as cryptographic accelerators by user applications. We have analysed two specific implementations, NXP's CAAM and the HSM used in recent Marvell SoCs, SafeXCell IP197. These units support acceleration of all major cryptographic algorithms, including AES, DES/3DES, RC4, MD5, SHA-256 and some advanced message authentication and authenticated encryption algorithms. They include a secure, NIST certified random number generator and support the import and export of cryptographic blobs into DDR or flash memory and provide both, DMA support and memory mapped slave interfaces.
Internally, the units are accessed through memory mapped portals (Job Rings), which are basically FIFOs that can be loaded with cryptographic job descriptors. Highly simplified, a job descriptor is a structure describing the cryptographic task to be

performed (“encrypt using AES256”), the key to used (“key #2 from internal key storage”) and the payload to perform the task on.

4.1.2 GNU/Linux support

For both units, Linux mainline driver support is available. Enabling them on Kernel level is a matter of adding the unit to the device tree and enabling and loading the drivers. In case of SafeXCell IP197, a binary firmware has also to be provided to the driver to be loaded into the unit. Successful enablement of the HSM adds it to the kernel CryptoAPI so that the cryptographic services can be used from kernel space, e.g. to provide IPSEC support. This can be checked by analysing the output of /proc/crypto which lists all the algorithms and services available, along with their priority of use. The first line in the table below shows an example output for different SHA-256 implementations. On the left side of the table, the implementation provided by the SafeXCell 197 is shown, on the right hand the one provided by the kernel itself, realized in software using the ARM64-bit NEON crypto extensions. Note the different priorities.

HSM Implementation	**Software Implementations**
name : sha256 driver : safexcel-sha256 module : crypto_safexcel priority : 300 refcnt : 1 selftest : passed internal : no type : ahash async : yes blocksize : 64 digestsize : 32	name : sha256 driver : sha256-arm64-neon module : kernel priority : 150 refcnt : 1 selftest : passed internal : no type : shash blocksize : 64 digestsize : 32
name : cbc(aes) driver : safexcel-cbc-aes module : crypto_safexcel priority : 300 refcnt : 1 selftest : passed internal : no type : skcipher async : yes blocksize : 16 min keysize : 16 max keysize : 32 ivsize : 16 chunksize : 16 walksize : 16	name : cbc(aes) driver : cbc-aes-ce module : kernel priority : 300 refcnt : 1 selftest : passed internal : no type : skcipher async : yes blocksize : 16 min keysize : 16 max keysize : 32 ivsize : 16 chunksize : 16 walksize : 16

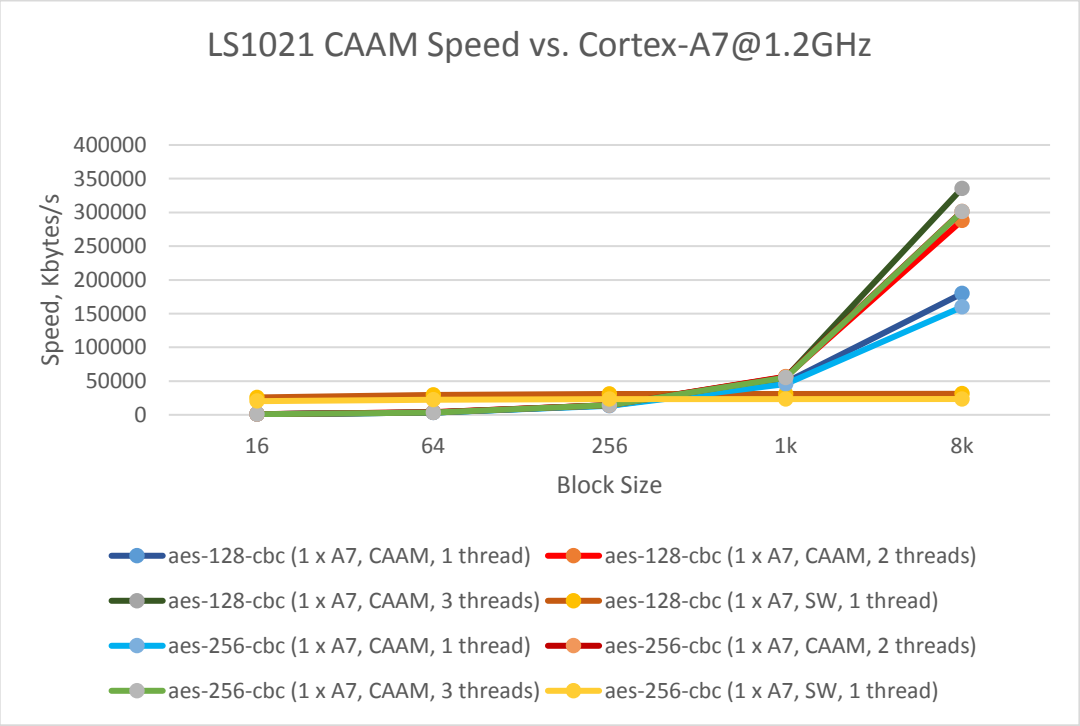
So in this example, if “sha256” as an algorithm is requested, the one provided by the SafeXCel HSM would be used because it has the higher priority. In the second example (line two of the table) for AES-CBC, two implementations have the same priority. The easiest way to select which one is used would be to unload the module providing the undesired one or to unselect it from the kernel build configuration. However, there are more sophisticated ways to granularly select specific implementations of a specific algorithm (contact the authors for details).

But how can the kernel cryptographic services be accessed from userspace? Several ways exist, e.g. AF_ALG [11] which provides services through sockets or

CryptoDev [12] which implements an OpenBSD compatible /dev/crypto device. Both methods have their advantages for specific applications, we will take a closer look at CryptoDev in this paper which is gaining more and more ground with SoC vendors and users.
Cryptodev is provided as an out-of-tree kernel module, support to build and include it is, for example provided through the Yocto Project, along with patches to work with recent Linux kernels. When loaded, it provides a new character device, called /dev/crypto which can be used to access the cryptographic services from userspace. Plenty of examples are provided that illustrate how to use CryptoDev in applications [13] and is also supported by well-known crypto libraries like GnuTLS [14] or OpenSSL [15].

4.1.3 Performance / Benchmarking

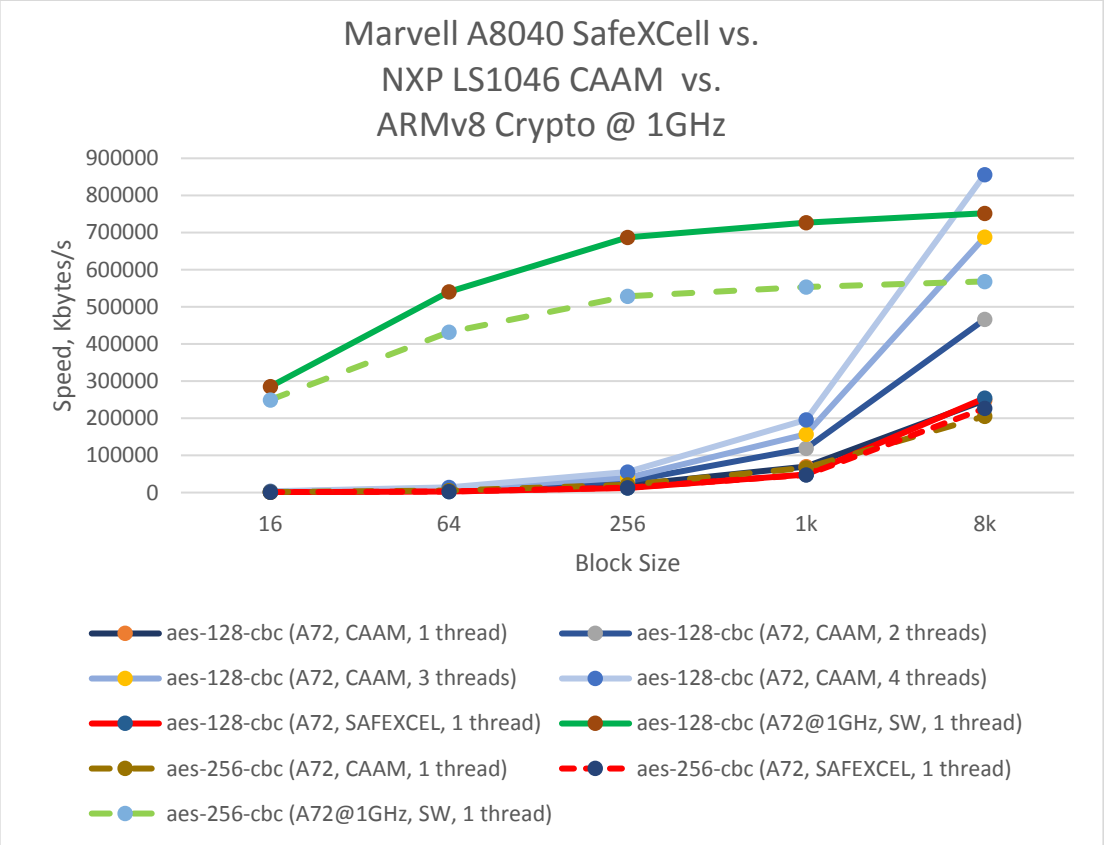
To get a first impression about the speed an HSM can provide, it should be benchmarked from userspace. OpenSSL is a great way to perform speed tests and comparisons between various engines and implementations using HSM, CPU Crypto-Acceleration functions or plain software algorithms. To do this, it should be compiled with HAVE_CRYPTODEV and USE_CRYPTODEV_DIGESTS defined, which is done automatically by Yocto if the CryptoDev recipe is included in IMAGE_INSTALL. Using the command openssl speed –engine cryptodev –elapsed –evp [cipher] starts the benchmark of a specific cipher algorithm implementation accelerated by a HSM supported by CryptoDev. The parameter –elapsed is important to get realistic results. It tells OpenSSL to measure the time it actually required to get the answer back from the engine (instead of the internal time used just to load the engine and process the results).
The diagram below shows the performance of NXP's CAAM engine on an ARMv7 Layerscape LS1021 device for AES128 and AES256, compared to running in software on ARM Cortex-A7 at 1.2 GHz.



Several observations are interesting to notice and generally true on most ARMv7 SoCs:

- HSM shows a significant speed advantage vs. implementation in software for bigger block size (up to 10 times)
- for smaller block sizes (below 512 bytes), the overhead of loading the engine with the job descriptors and the memory transfer latencies involved is so high, that the achieved throughput is less than in software
- Even for small block sizes, the use of a HSM might be preferred to save CPU cycles and because of its better energy efficiency
- The HSM only achieves optimal results if being used by multiple threads (OpenSSL option: -multi n). In other words, one thread alone is not able to exploit the engine to its maximum performance.
- CPU load (for providing data to the HSM, interface handling and housekeeping) is about 40% to 50% of the load compared to when actually executing the algorithm in Software (setup: cryptodev, openssl). So by using the HSMs CPUs can be relieved from crypto operation load and a better total system energy efficiency can be achieved.

The following diagram shows the same benchmark performed on two 64 bit SoCs (Marvell Armada 8040 with SafeXCell 197 HSM and NXP LS1046 SoC with CAAM HSM). The core frequencies have been limited to 1 GHz to achieve comparable results.



The following observations are interesting to notice and generally true on most ARMv8 SoCs:

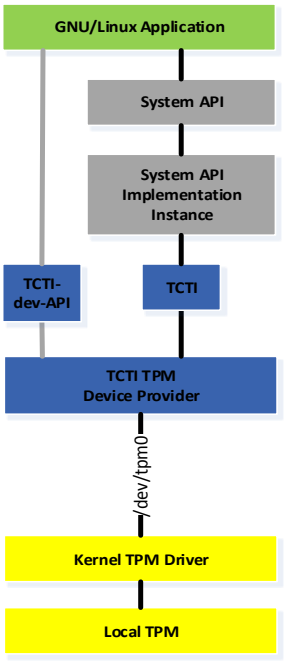
- ARMv8 crypto extensions show a significant speed advantage over the HSM implementation for all block sizes
- For larger block sizes, the HSM implementation is able to reach similar speeds to the single-core software implementation when multiple threads are being used to push crypto pushing data into the HSM do not increase the crypto speed
- HSM implementation shows little speed degradation compared to software implementations for increasing key lengths on AES and other algorithms, profiting from parallel hardware implementations
- CPU load (for providing data to the HSM, interface handling and housekeeping) is about 40..50% of the load compared to when actually executing the algorithm (setup: cryptodev, openssl). Therefore from a performance point of view, using HSM from userspace is not recommended. Example: on NXP LS1046, for 8k blocksizes, 2 CPUs have to be used to 70% each to achieve an AES throughput through the CAAM engine which is equivalent to running the algorithm on one single core (90% load). The throughput achieved in software can't even be reached on CAAM for smaller block sizes.
- The use of HSMs might still be desired because of their better energy efficiency or if performance is not relevant.
- The main advantage of HSMs is when being used by other hardware engine, such as the Ethernet offloading system (DPAA/DPAA2 for NXP CPUs).

As the diagrams and further experimental results (available through the authors) show, for most algorithms there is a break even point in block size, at which the HSM implementation starts to become more efficient (in terms of energy, speed, or total system load) than a software implementation. This break even point can to be determined from experimental results. From an optimization point of view, it is then possible (e.g. in cryptodev) to distribute tasks to either the software or hardware engine, depending on these even points. In a more advanced design (either implemented in cryptodev or directly in the kernel crypto framework) a concept can implemented to distribute crypto operations to both HSMs and software implementations (e.g. locked to crypto-designated cores) to implement a speed and energy optimized high-end crypto system. Please contact the authors for more information on this.

4.2 TPMs on Application Level

In this chapter we will describe the support for TPM 2.0 in GNU/Linux for Kernel, userspace and some application examples.

4.2.1 GNU/Linux Support



Driver support implementing TIS and TCTI for TPM 2.0 has been mainlined with Kernel 4.8 for several vendors. To avoid the limitation of one userspace application blocking the TPM for exclusive use, an access broker and resource management daemon was added with Kernel 4.12.
But on top of the driver, there is some more infrastructure required to actually use a TPM.
In addition to the module specification itself the TCG also defines the **TPM Software Stack (TSS)** that defines the Software API (SAPI) and the TPM Command Transmission Interface (TCTI). There exists also an additional Enhanced SAPI to simplify the userspace access on TPM functions out of several programming languages.
At the moment several TSS implementations exist, e.g. from Intel or IBM that can be used on GNU/Linux systems. The picture right shows the basic concept on a GNU/Linux system. A kernel driver is used to communicate with the TPM e.g. by using the TPM Interface Specification (TIS). This driver creates the */dev/tpm0* device to tunnel TCTI communication to the TPM. Optionally (depending on CONFIG_HW_RANDOM_TPM in the kernel configuration) another device for reading random numbers can be generated. One layer above resides the TCTI TPM Device Provider service as part of the TSS what will be used mostly by the SAPI layer. The SAPI is implemented as a userspace library to be linked dynamically or statically. Applications may also connect to the TPM device provider directly. There exist possible modifications and extensions of that layered stack to support more complex systems (virtualization, remote TPM access) which are explained in more detail in the TSS System Level API and TCTI Specification.
Before a TPM can be used on a specific system it needs to be provisioned. During provisioning, a TPM needs to be enabled and activated. In a next step the Endorsement primary key pair will be created using TPM2_CreatePrimary. Because the fact that the calculation of the key pair is based on the Endorsement Seed in conjunction with the KDF it can be recreated each time again as long as the TPM hasn't be cleared and doesn't need to be stored in the NVM. Based on the several asymmetric algorithms and added template entropy of TPM 2.0, there can be more than one primary endorsement key pair or storage key pair as well as other keys. On GNU/Linux this is a manually process what can be automated by a script of course. A useful feature is the possibility to setup policies on the TPM to control access to keys or NVRAM data depending on e.g. specific PCR values ("sealing").

4.2.2 TPM application usecases in GNU/Linux

The following mentions some interesting tools complementing the TSS on userspace which might be beneficial for applications:

- tpm2-tools: According to the TSS SAPI that project implements most of the API as
  command line tools to access TPM 2.0 functionalities.
  Available from https://github.com/01org/tpm2-tools

- tpm2-abrmd: an implementation of a TPM2 access broker and resource management daemon.
  Available from: https://github.com/01org/tpm2-abrmd

- eltt2: The Embedded Linux TPM toolbox, the swiss army knife to access TPM 2.0 functionality from the Linux command line.
  Available from https://github.com/Infineon/eltt2

The SAPI as the standardized interface to that functions or alternatively the TPM2-tools are available from the command line. For example, the tpm2_getrandom function read a random number from the TPM's Hardware Random Number Generator.
Other functions may be used to create and manage keys for using them with the various symmetric and asymmetric encryption algorithms of the TPM for small portions of data or key management and distribution of encrypted keys for symmetric cryptography. Users can export and import their keys while the key itself will be encrypted before leaving the TPM to keep the keys secret at all.
As mentioned before PCR 17 to 24 are under control of the user. This includes a reset of that registers. They can be used for doing some application specific measurement to monitor the healthiness of specific software products and configurations or to do software license management. Also the NV-RAM van be used by user applications. However, users need to take care of the specified maximum number of write cycles and data retention limitations.
Alternatively some of the TPM functions are implemented in the eltt2 command line tool which communicates directly over the TCTI.

4.3 Advanced Usecases

In this chapter we briefly look into some advanced use cases on application level combining HSMs and TPMs.

4.3.1 Encrypting / decrypting sensitive data

Encrypting and decrypting sensitive data is a very common requirement on application level. For small payloads, this task can be easily done using a TPM, which offers the following advantages:

- Key generation and protection without exposing them to the SoC world
- Key access rights can be restricted or granted based on the trust state of the system
- Encryption is performed with maximum security against snooping

However, in the following situations, combining the TPM with other methods might be preferred.

- huge payloads that would take too long on the TPM to encrypt / decrypt or high throughput demands
- the SoC's tamper detection system should be leveraged to protect and destroy keys and sensitive data upon attack

In these cases, there are still two possibilities:

- exclusive use of HSM
  This would be the preferred way if the SoCs tamper detection system shall be leveraged. Most HSMs support advanced methods to import and export cryptographic blobs into memory and advanced key generation methods. However, these methods are highly proprietary and require access to the detailed documentation of the HSM (e.g. the "Security Reference Manual").
- combining HSM and TPM
  This methods offers the best of both worlds by using the TPM as highly secure and certified key generator and storage and the HSM to actually encrypt the data using this key. This task can be either performed using proprietary software that directly accesses the HSM or using the methods described in chapter 4.1.2 e.g. through OpenSSL. There are ongoing actions to integrate a "tpm2" engine into OpenSSL [16] and some manual approaches [17] to achieve this goal.

4.3.2 IPSEC

If a kernel driver for an HSM is available and the required algorithms are supported, the HSM can be used to accelerate IPSEC traffic. TPMs can be integrated into this solution as key generators, storages and to perform low-performance cryptographic operations such as verifying signatures. Solutions such as StrongSwan support the use of TPM 2.0 though a plugin. [18] and [19] describes the possibilities which are offered by this, including remote attestation of IMA results.

4.3.3 Network Acceleration

Typically the most powerful HSMs can be found on networking processors, such as NXP's Layerscape or Marvell's Armada 8040 family to accelerate the encryption of network traffic. To be able to do this, these modules need to be highly embedded into the hardware network acceleration modules of these SoCs, such as queue, buffer and frame managers in NXP's DPAA or
DPAA2. If such an acceleration system is used in conjunction with optimized network stacks to leverage all components, Ethernet traffic can be encrypted in line-speed on such platforms. Please contact the authors for more information on this subject.

5 **Conclusion**

Hardware accelerated cryptography support is an extremely valuable addition to enhance GNU/Linux security. However, while some applications are well enabled (user space cryptography, IMA, image authentication), support for others is missing completely (TPM 2.0 and measurement support in u-boot, TPM integration as a key storage into dm-crypt). Due to the rising security demands, this will most likely improve in the future and help to replace proprietary HSM enabled solutions with TPMs. ARMv8 crypto accelerations show great speeds for most algorithms, so that the predominant use of HSMs (crypto speed) will most likely also be complemented and replaced by pure software implementations in the future. The maturity and adoption for productive use of such solutions also highly depends on community evaluation and implementation. So feel free to contact the authors to discuss collaboration on implementation or just some feedback about your use cases and experiences.

The authors would like to thank our partners at Infineon Technologies AG and Vector Informatik GmbH for their invaluable help and support. For reasons of privacy we are not mentioning you by names, but you know if you're meant.

References

[1] https://trustedcomputinggroup.org/
[2] https://shattered.io/
[3] https://github.com/torvalds/linux/blob/master/arch/arm64/crypto/aes-ce-cipher.c;
https://www.linaro.org/blog/core-dump/accelerated-aes-for-the-arm64-linux-kernel/
[4] https://www.fsf.org/campaigns/secure-boot-vs-restricted-boot/whitepaper.pdf
[5] https://sourceforge.net/p/linux-ima/wiki/Home
[6] https://www.kernel.org/doc/Documentation/device-mapper/verity.txt
[7] https://github.com/torvalds/linux/blob/master/Documentation/device-mapper/dm-integrity.txt
[8] https://github.com/torvalds/linux/blob/master/Documentation/device-mapper/dm-crypt.txt
[9] https://gitlab.com/cryptsetup/cryptsetup
[10] https://github.com/rqou/tpm2-luks
[11] http://www.chronox.de/libkcapi/html/ch01s02.html
[12] http://cryptodev-linux.org/
[13] https://github.com/cryptodev-linux/cryptodev-linux/blob/master/examples/
[14] http://www.gnutls.org/
[15] http://www.openssl.org/
[16] https://dguerriblog.wordpress.com/2016/03/03/tpm2-0-and-openssl-on-linux-2/
[17] https://mta.openssl.org/pipermail/openssl-dev/2016-December/008924.html
[18] https://wiki.strongswan.org/projects/strongswan/wiki/TPMPlugin
[19] https://www.strongswan.org/docs/ConnectSecurityWorld_2016.pdf