PART 2

Application Specific MPSoC Architectures

3

Automation for Industry 4.0 by using Secure LoRaWAN Edge Gateways

Marcello COPPOLA[1] and George KORNAROS[2]

[1]*ST Microelectronics, Grenoble, FR*
[2]*ECE Dept., Hellenic Mediterranean University, Iraklio, Crete, GR*

Industrial IoT (IIoT) and Industry 4.0 signify the advent of connecting industrial automation devices and equipment, or "things", with cloud-based systems to harvest, process and analyze information faster than ever and to increase business value through novel customer experience and services (real-time analytics or visualization of data) and improving their operational efficiency. On the other hand, the industry will need to reduce risks in security more than ever on. Therefore, they need to protect all connected devices and environments on all fronts. In this increasingly connected and software-dominated ecosystem scenario, IoT gateways are the key components. IoT gateways communicate to multitude of sensors and automation controllers called "edge devices" and provide the bridge between an on-premise communications network and cloud-based computing supremacy and visualization.

In an Industrial IoT ecosystem, security is the key aspect in which IoT gateway security is of prime importance since a secured gateway enables protection of the industrial production cycle. To guarantee safe and secure operation, services availability and improved user experience, it is mandatory to ensure sufficient

For a color version of all figures in this book, see www.iste.co.uk/andrade/multi2.zip.

Multi-Processor System-on-Chip 2 – Applications,
coordinated by Liliana ANDRADE and Frédéric ROUSSEAU. © ISTE Ltd 2020.

security measures, thus avoiding chances of potential risks such as malicious threats, spoofing, man-in-the-middle (MITM) attacks, data snooping or privacy breaching.

In conjunction with many technological, environmental and economic benefits, the rapidly moving connected world also represents an array of growing attacks such as side-channel attacks, fault attacks and physical tampering. Considering these risks, ensuring security, trustworthiness and robustness of IIoT becomes imperative, in which IIoT gateways play an important role. Providing complete IoT security not only requires that the communication from the gateway to the cloud is secure but also that the gateway can participate in the secure communication and management of connected edge node devices, which themselves must be secured. This chapter presents the key concepts required to implement a security architecture to enable complete IIoT security in a LoRaWAN environment through using in-depth analysis of methods.

3.1. Introduction

The digital revolution blurred the boundaries between the physical and digital worlds within Industry 4.0, which becomes the place where employees, machines and products interact via an Industrial IoT (IIoT) system. An IIoT system leverages on CPS that enables different physical sensors, actuators and controllers to be locally interconnected among them and globally connected to cloud computing servers, forming complex online systems a collection of independent interconnected computing elements. The novel Industry 4.0 manufacturing process implies a strong interaction between employees and products at all steps. Industry 4.0 plays a significant role in strategy to take the opportunities of digitalization of all stages of production and service systems. The Fourth Industrial Revolution is increasingly establishing by the combination of numerous physical and digital technologies such as artificial intelligence, cloud computing, adaptive robotics, augmented reality, additive manufacturing and Internet of Things (IoT). As an industrial IoT platform (IIoT) is composed of IIoT devices that enable to connect objects, assets, machinery, embed sensors and tags and establish an Internet-enabled connection to transmit data and remotely monitor and control the manufacturing production line. Generally, an IIoT device is composed of a microcontroller (e.g. an STM32) that is the brains of the IIoT device itself, where the application runs, a *communication module* (such as Wi-Fi, BLE and LORA) and one or more *sensors* or *actuators* of some kind. The microcontroller handles the application, data acquisition and communications with the Cloud Service. On top of these IIoT platforms, several services can be provided by IIoT solution providers. Services are what the user sees and interacts with. Since these services use Internet, it is important for IIoT solution providers to make sure that the IIoT device is trustworthy, because they wish to protect their services, network and brand.

As IIoT deployments have grown, so have threats to security. According to the SonicWall Cyber Threat Report (SonicWall 2019), through the first six months of 2019, SonicWall has registered 2.4 million encrypted attacks on IoT, almost eclipsing the 2018 full-year total in half the time. Security has to be considered from the very beginning of the design phase so as to mitigate vulnerabilities and threats. In order to be considered trustworthy, IIoT devices need to be legitimate, identifiable and manageable to prevent hackers, counterfeits, ODM overproduction and IP theft. One reason for the difficulty in securing connected devices is that many stakeholders are involved in the IIoT device life cycle, with each of them facing different threat vectors that implies different security requirements. Since OEMs, manufacturers, integrators and designers are involved in developing and implementing the IIoT devices, it becomes quite complex to legitimate and to identity all of them.

The generic architecture for IIoT solution includes IIoT devices that are connected to the private or a private area of the public cloud (e.g. AWS IoT Core) using some sort of network. Thus, IIoT devices can be linked to services that are executed in the Cloud related to Analytics, Billing, CRM, NoC, etc. (e.g. AWS services, Chargify, tableau, saleforces) and other devices. Generally, Cloud providers try to secure data and interactions, process the data transmitted by different devices and trigger actions accordingly. In addition, they allow IIoT applications to interact with IIoT devices or machines even when IIoT devices/machines are not connected.

As it stands today, the IIoT industry does not currently have a widely used and accepted standard by vendors on how IIoT devices should connect to each other. This makes it hard for developers to understand how this can be achieved across multiple manufacturers when developing IIoT applications. That traditional way to connect IoT devices to the clouds is by specific SDKs. For instance, the SDK for AWS IoT devices makes it possible to connect and authenticate IoT devices to AWS IoT Core, as well as exchange messages with the service, via MQTT, HTTP or WebSockets protocols. This SDK for AWS IoT devices supports C, JavaScript and Arduino and includes client libraries, a manual for developers and a porting guide for manufacturers. Although these solutions address security, they are not able to mitigate vulnerabilities and threats. As a matter of fact, in Industry 4.0, hence IIoT Security needs to combine IIoT device security with network security. The IIoT device security requires insuring a trusted and authentic software and firmware execution that implies a secure and verified boot, while the network security needs to insure data integrity, authentication via unique device identity and data communication protection.

The most important decision when designing an IIoT device is the choice of a robust root-of-trust management. This is a basic need and is today the weakest link in the IoT. Proving ownership of the root of trust is the baseline for cryptographic and security foundation to enable safe data communication and functional operation of IIoT networks. It is widely acknowledged in the security industry that strong security mechanisms have to be based on hardware because software can be

always circumvented by software. Any IIoT solution has to combine operational technology (OT) and information technology (IT) in order to implement **machine to machine connectivity, to better monitor the manufacturing process and control the physical machine remotely**. In IIoT the IT network plays an important role since it moves data from the sensor to the backhaul that takes it up to the Internet. Several low-power wide-area network (LPWAN) (Raza *et al*. 2017) technologies have been proposed as an alternative solution for implementing private cellular-like networks with infrastructure costs as low as possible. Coverage, scalability and energy efficiency requirements of Internet of Things (IoT) deployments have recently steered the attention of the research community, industry and several network operators to long-range wide-area network (LoRaWAN) (Augustin *et al*. 2016; Palattella *et al*. 2016; Raza *et al*. 2017). It integrates the long-range (LoRa) technology at the physical (PHY) layer, provides specifications for both media access control (MAC) and network layers and defines the four key components of the overall architecture, which are end nodes, gateways, cloud/network servers and remote applications (LoRa Alliance Technical Committee 2017).

OT is responsible for monitoring and managing industrial process assets and manufacturing equipment. OT is a technology that provides a way to interface the physical world. OT includes industrial control systems (ICS), supervisory control and data acquisition (SCADA) and distributed control systems (DCS). As defined by Gartner: "Hardware and software that detects or causes a change through the direct monitoring and/or control of physical devices, processes and events in the enterprise". Combining OT with IT-based technologies such as big data and the Internet of Things, we move to IIoT that open the door to novel services within the industrial processes, such as remote diagnostics and predictive maintenance. As a result, physical machines will produce a large amount of data, which if properly analyzed can improve the management of the manufacturing process. Since connectivity opens a door to the outside, it is important to tackle all the malware, identity management and access control security challenges that may rise for OT, since unmanaged vulnerabilities in an OT system can leave critical infrastructure at risk of sabotage. Thus, the ongoing digitization required security. There are also important differences between IT and OT security. The latter traditionally was more about protecting physical processes, safety, uptime/production/efficiency and protection of people, while IT security is more oriented on protecting all aspects of data and how information is stored, transmitted, processed and used in business processes.

3.2. Security in IIoT

Given the trends and tremendous growth of IoT environments, privacy and security have been of prime importance in both the industrial and commercial requirements of IoT devices. The security threats and privacy concerns in each layer of the IoT architecture have now become an exigent area of research (Abomhara and Køien

2015; Chen *et al*. 2018; Sisinni *et al*. 2018; Bhat and Dutta 2019; Hassija *et al*. 2019). The primary principles to guarantee for security in the entire IIoT system involve the following:

– **Confidentiality**: Confidentiality is the ability to hide information from people who are unauthorized to access it and thus needs protection from unauthorized access. Confidentiality is an important security feature in IoT. In most situations and scenarios, sensitive data, for instance, patient data, private trade data and/or military data as well as security credentials and secret keys, must be hidden from unauthorized accesses.

– **Integrity**: Integrity of information refers to protecting information from unauthorized, unanticipated or unintentional modification. Integrity is a mandatory security property in most cases in order to provide reliable services to IoT entities (i.e. devices, users). Different systems in IoT have diverse integrity needs.

– **Availability**: Availability is the access to information whenever needed by a user of a device (or the device itself). Therefore, the IoT resources must be available on a timely basis to meet needs or to avoid significant losses.

– **Authenticity**: The authenticity property allows identification of the sender entity and further, only authorized entities to perform certain operations in the device or network. Different authentication needs require different solutions. Some solutions must be strong control, for example, authentication of finance systems.

– **Non-repudiation**: IoT service must provide a trusted audit trail. The property of non-repudiation presents certain evidence in cases where the user or device cannot deny an action, for instance, payment action.

– **Privacy**: Privacy is an entity's right to determine the degree to which it will interact with its environment and to what extent the entity is willing to share personal information with others.

3.3. LoraWAN security in IIoT

To address wireless security concerns, LoRaWAN uses two layers of security, as can be seen in Figure 3.1: one for the network and one for the application layer. Network layer security ensures the authenticity of the device in the network. Application layer security ensures that the network operator does not have access to the end user's application data.

An end device (Node) is activated before it can communicate on the LoRaWAN network either by using *over-the-air activation* (OTAA) or *activation by personalization* (ABP) method. OTAA is the recommended activation method for higher-security applications (LoRa Alliance Technical Committee 2017).

The first method uses over-the-air *join requests* coupled with *join accept* messages. Each end device (Node) is deployed with a 64-bit DevEUI, a 64-bit AppEUI and a

128-bit AppKey. The DevEUI is a globally unique identifier for the device which has a 64-bit address comparable with the MAC address for a TCP/IP device. The AppKey is used to cryptographically sign the join request. All three values are then made available to the application server to which the device is supposed to connect. The AppKey is used when the node sends a join request message. The node sends the join request message, composed of its AppEUI and DevEUI. It additionally sends a DevNonce, which is a unique, randomly generated, 2-byte value used for preventing replay attacks.

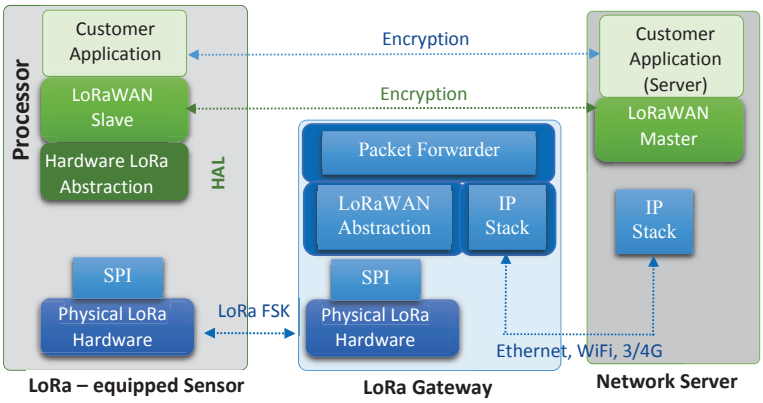

Figure 3.1. *Security in LoRaWAN*

These three values are signed with a 4-byte MIC (Message Integrity Code) using the device's AppKey. The server accepts join requests only from devices with known DevEUI and AppEUI values while validating the MIC using the AppKey. If the server accepts the join request, it responds to the device with a join accept message. The application and network servers calculate the node's two 128-bit keys: the application session key (AppSKey) and the network session key (NwkSKey). These are calculated based on the values sent in the join request message from the node.

Additionally, the application server generates its own nonce value: AppNonce. This is another unique, randomly generated value. The join accept reply includes the AppNonce, a NetID and an end device address (DevAddr) along with configuration data for RF delays (RxDelay) and channels to use (CFList). The device address (DevAddr) in the join accept reply is a 32-bit identifier, which is unique within the network. It is possible to use the device address to differentiate between end devices which have already joined the network. This allows the network and application servers to use correct encryption keys and to properly interpret the data. When receiving data back, the data is encrypted with the AppKey. The node then uses the

AppKey to decrypt the data and derives the AppSKey and the NwkSKey using the AppNonce value received in the join accept reply.

3.4. Threat model

The threat model we consider involves attacks seeking to perform malicious network or firmware attacks, by sniffing and replaying network traffic, by modifying the IoT node firmware or by injecting malicious code to the firmware, to the operating system or to the kernel driver. Because the presence of a malicious manufacturer is not a consideration for our attack model, intentionally implanted vulnerabilities or malwares are not assumed to be loaded.

3.4.1. *LoRaWAN attack model*

In the OTAA method, details are exchanged (using join request and join accept messages) and security contexts are generated. The generated security contexts make sure that both the end-device and the network are using the correct security keys for encryption and message signing. The join accept message however is vulnerable to an eavesdropping and replay attack combination. The first step in OTAA is that the end device transmits a join request message to join the network. Such a message contains an AppEUI, DevEUI and DevNonce. The AppEUI and DevEUI are unique addresses in the EUI-64 address space. The DevNonce is a random value which is used to prevent replay attacks. The join request message is processed by the network server. If the network server accepts the end device, a join accept message is generated and returned to the end-device. The join accept message contains details (such as AppNonce and NetID) that are used for generating a correct security context (NwkSKey, AppSKey, frame counters, etc.) so that the end device uses the same security keys as the network. Unfortunately, the join accept message does not include a reference to the join request message that it was generated for, and therefore OTAA is vulnerable for replaying older join accept messages addressed to the same end device. Additionally, the network server tracks not all but the most recently used N DevNonce values. These vulnerabilities in the LoRaWAN specification v1.0.2 have been successfully mitigated in v1.1, first by using an incrementing JoinNonce value to prevent replay attacks of Join-Accept messages, and second a v1.1 end-device does not reuse a DevNonce value (LoRa Alliance Technical Committee 2017). However, LoRaWAN specification v1.1 includes a compatibility mode for v1.0.2-only end devices. For these devices, this vulnerability is still present, even though the network is running on v1.1. Further, a malicious network server is of major concern, since by replaying OTA activation messages to the join server, this may in turn cause the overflow of a counter-nonce, and the compromise of the integrity and confidentiality of application data.

Root keys are used for protecting the integrity of join request messages and protecting the confidentiality of join accept messages triggered by join request

messages. These root keys stored in the end devices and in the join server are essential to the overall security of LoRaWAN. Hence, these keys must be protected against physical tampering. The specification leaves secure provisioning, storage and usage of root keys out of scope, but points to SE (Secure Elements) and HSM (Hardware Security Modules) as possible enablers for a solution. LoRaWAN does not feature OTA firmware updates. This makes issuing fixes for the discovered vulnerabilities to the deployed end devices impractical and also makes it more likely that different versions of the protocol will coexist in a network, which in turn elevates the importance of addressing security issues related to backward compatibility.

Finally, it has been demonstrated that the LoRa transmissions are prone to jamming attacks (Reynders *et al*. 2016). Specifically, simultaneous communications that use the same spreading factor and frequency can conflict with each other. Although it is hard to prevent jamming attacks, by detecting such abnormal behavior, network administrators can take appropriate action (e.g. by switching the operational frequency) to prevent the impact of jamming.

3.4.2. *IIoT node attack model*

An attacker can perform malicious firmware attacks by modifying the firmware of an IoT node or injecting malicious code to the firmware. Attackers with physical access to the device can even perform cold-boot attacks to recover (even imperfect) keys from on-device persistent memory (flash) or dynamically compromise the firmware running in DRAM memory. Attacks come in a large variety and may include side-channel attacks, where the physical realization of a cryptographic primitive can leak additional information, such as computation time, power consumption and radiation/noise/heat emission.

During firmware execution, an adversary can seek to exploit the dynamic memory regions (stack and heap) to circumvent secure boot and attestation. Exposure at execution time can also come from firmware's vulnerabilities such as code injection and code reuse attacks through buffer overflow (when new data exceed predefined fixed size) (Bishop *et al*. 2012) and privilege escalation, or API abuse. In a form of code reuse attack called return-oriented programming (Buchanan *et al*. 2008) or return-flow hijacking attack, the address of different code snippet is put on the stack (by leveraging a stack overwrite vulnerability) and then exploits the return address of a function to jump to this code snippet.

We do not consider cache/timing side-channel attacks (Lipp *et al*. 2016) and hardware attacks, for example, cold-boot (Halderman *et al*. 2009), bus snooping (EPN Solutions 2014) and rowhammer (van der Veen *et al*. 2016). Defense techniques against these attacks can be adopted in the future.

3.5. Trusted boot chain with STM32MP1

The primary principle to establish the foundation of trust in service and data of a device is based on the auditable and securely signed history of operations of a software running on the device.

3.5.1. *Trust base of node*

Modern commercial embedded devices commonly comprise ARM-based SoC hardware with security extension. These nodes encompass ARM TrustZone to play a role in keeping security-sensitive resources safe and to execute security-critical services. We assume that commercially available security features such as secure boot (ARM Ltd. 2009) and active monitoring are already activated as a base-line defense against attacks. Thus, a trusted execution environment (TEE) that is isolated from a rich execution environment (REE) targets to protect assets such as crypto keys and user credentials.

In such system, we consider attackers who can use any existing system vulnerability to hack into the normal-world or secure-world operating system, either by loading new unverified code binaries or by modifying code binaries that were previously mapped and already exist in memory. Thus, we consider attacks that transiently manipulate the legitimate code in user mode or map the malicious code in the data region of a pre-authorized application, but additionally attempt to leak secure critical data by malicious code or untrusted operating system.

3.5.2. *Trusted firmware in STM32MP1*

Trusted firmware-A (TF-A) is a reference implementation of secure-world software for Arm A-Profile architectures (Armv8-A and Armv7-A), including an exception level 3 (EL3) secure monitor (ST 2019; TrustedFirmware Association 2019). It provides a suitable starting point for productization of secure world boot and runtime firmware, in either the AArch32 or AArch64 execution states. Although it was first designed for Armv8-A platforms, it has been adapted to be used on Armv7-A platforms by STMicroelectronics.

To boot Linux in an STM32MP1 device, several steps are required to progressively initialize the platform peripherals and memories. In an STM32MP1 device, TF-A is a reduced version of the Linux kernel, which is configured with only the devices used during boot via the device tree. TF-A consists of the following binaries: BL1, BL2 and BL32, where the full boot sequence in AArch32 (ARM 32-bit processors) defines:

– boot loader stage 1 (BL1) application processor-trusted ROM;

– boot loader stage 2 (BL2) trusted boot firmware;

– boot loader stage 3-2 (BL32) runtime software;

– boot loader stage 3-3 (BL33) non-trusted firmware.

For STM32 MPU platforms, BL33 is the secondary stage boot loader (SSBL), which is U-Boot by default and is the first non-secure code loaded by TF-A. Figure 3.2 depicts the trusted boot chain process in STM32MP1 device.

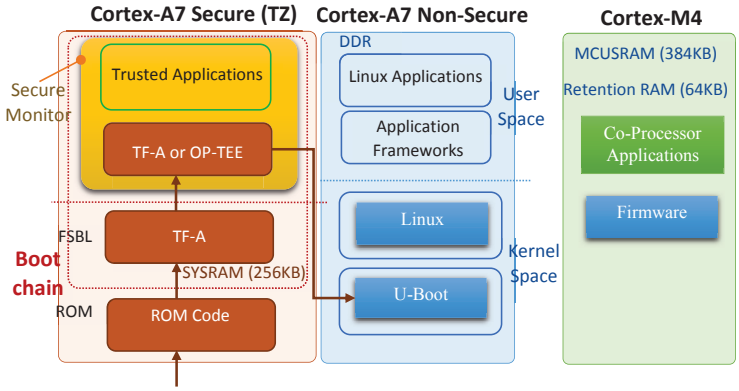

Figure 3.2. *Boot process in an STM32MP1 device*

By using asymmetric cryptography, the boot process is as follows: software image hash is encrypted with the private key and is deployed with its signature on device flash. The public key is burned inside STM32MP1 chip's OTP zone. After enabling secure boot feature, at every boot, the STM32MP1 device extracts image signature, decrypts it with the public key, calculates software image hash and compares the two hashes: software will be executed only if they are equal.

3.5.3. *Trusted execution environments and OP-TEE*

OP-TEE (Open Portable TEE, an open-source ARM TrustZone® solution) is a *trusted execution environment* (TEE) designed as a companion to a non-secure Linux kernel running on Arm Cortex®-A cores using the Arm TrustZone technology. TrustZone has been used as a cornerstone hardware technology for enabling TEE on ARM-based platforms which share the majority of mobile and embedded markets. OP-TEE implements TEE Internal Core API v1.1.x, which is the API that is exposed to trusted applications, and the TEE Client API v1.0, which is the API that describes how to communicate with a TEE. Those APIs are defined in the GlobalPlatform API specifications. OP-TEE is designed primarily to rely on the ARM TrustZone technology as the underlying hardware isolation mechanism, to enable a trusted execution environment, as shown in Figure 3.3. However, it has been structured to

be compatible with any isolation technology suitable for the TEE concept and goals, such as running as a virtual machine or on a dedicated CPU.

A *rich execution environment* (REE) is an execution environment that involves at least one device and all its components or an OS, excluding any trusted or secure component. In contrast, a trusted execution environment (TEE) provides a level of security to protect against attacks and secures data access. The TEE executes alongside the REE, but is shielded from it. A trusted application executes inside a TEE and exposes secure services to applications in the REE. To exploit the trusted functionalities provided by TEE, applications need to be split into client application (CA) and trusted application (TA), where the latter holds the security-sensitive functions (i.e. implementation of encryption algorithm).

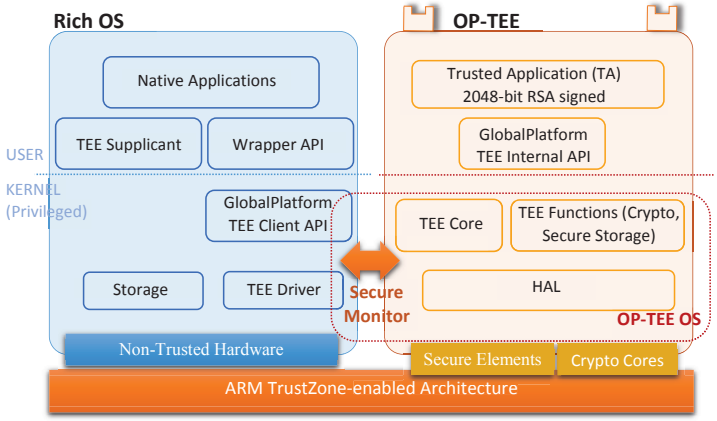

Figure 3.3. *Execution environments in OP-TEE enabled organization based on ARM TrustZone architecture*

ARM processors with TrustZone implement architectural security extensions in which each of the physical processor cores provides two virtual cores, or security domains, one being considered non-secure and called *non-secure world*, the other being considered Secure and called *secure world*, and a mechanism to context switch between the two, known as the *monitor mode*.

TrustZone is based on the principle of *least privilege*, which means that system modules such as drivers and applications do not have access to a resource unless necessary. The concept of secure (trusted) and non-secure (non-trusted) worlds extends beyond the processor to encompass memory, software, bus transactions, interrupts and peripherals within a SoC. By creating a security subsystem, assets can be protected from software attacks and common hardware attacks. TrustZone can

secure a software application, library or an entire OS to run in the secure area. Non-secure software is blocked from access to the secure side and resources that reside there.

When secure mode is active, the software running on the CPU has a different view on the whole system than software running in non-secure mode. Accordingly, system functions and, in particular, security functions and cryptographic credentials can be hidden from the normal world.

OP-TEE OS executes in the secure world. World switch is done by the core's secure monitor level/mode, referred next as the monitor. Switches of world execution context are handled based on SMC exceptions and interrupt notifications. Interrupt notifications are IRQ/FIQ exceptions which may also imply switching of world execution context: normal world to secure world, or secure world to normal world.

3.5.4. *OP-TEE scheduling considerations*

OP-TEE standard services are carried out through standard SMC. Execution of these services can be interrupted by foreign interrupts. To suspend and restore the service execution, OP-TEE OS assigns a trusted thread at standard SMCs entry. A trusted thread execution can lead OP-TEE OS to invoke a service in normal world: access a file, get the REE current time and so on. The trusted thread is suspended/resumed during remote service execution. When a trusted thread is interrupted by a foreign interrupt and when OP-TEE OS invokes a normal world service, the normal world gets the opportunity to reschedule the running applications. The trusted thread will resume only once the client application is scheduled back. Thus, a trusted thread execution follows the scheduling of the normal world caller context. OP-TEE OS does not implement any thread scheduling. Each trusted thread is expected to track a service that is invoked from the normal world and should return to it with an execution status.

3.5.5. *OP-TEE memory management*

OP-TEE currently requires more than 256 KB RAM for OP-TEE kernel memory. This is not a problem if OP-TEE uses TrustZone-protected DDR, but for security reasons, OP-TEE can be configured to use TrustZone-protected SRAM instead. TrustZone-protected SRAM is generally considered more secure than TrustZone-protected DRAM as there is usually more attack vectors on DRAM. The amount of available SRAM varies between platforms, from just a few KiB up to over 512 KiB. Platforms with just a few KiB of SRAM cannot be expected to be able to run a complete TEE solution in SRAM. But those with 128 to 256 KiB of SRAM can be expected to have an appropriate TEE solution in SRAM. The *pager* provides a solution to this by demand paging parts of OP-TEE using virtual memory.

To prevent physical attacks, *pager* is a demand paging system separated from the OP-TEE kernel, which is responsible for maintaining execution of the rest components of OP-TEE kernel and TAs. It sets the on-chip SRAM memory (OCM) as the working memory for CPU to execute the OP-TEE system, and uses the DRAM as a backing store. *Pager* runs on the OCM, and the other components of the OP-TEE kernel and TAs are encrypted and stored in the DRAM to guarantee the confidentiality and integrity properties for the backing store. *Pager* manages swapping between the OCM and DRAM: when code or data stored in DRAM is demanded, *pager* decrypts and performs integrity check on the corresponding page and loads it into the OCM; when an OCM page needs to be swapped to DRAM, *pager* encrypts it.

OP-TEE uses several L1 translation tables, one large spanning 4 GiB and two or more small tables spanning 32 MiB. The large translation table handles kernel mode mapping and matches all addresses not covered by the small translation tables. The small translation tables are assigned per thread and covers the mapping of the virtual memory space for one TA context.

Shared memory is a block of memory that is shared between the non-secure world and the secure world. It is used to transfer data between both worlds. The shared memory is allocated and managed by the non-secure world, i.e. the Linux OP-TEE driver. The secure world only considers the individual shared buffers, not their pool.

The Linux kernel driver for OP-TEE is responsible for allocating chunks of shared memory. OP-TEE Linux kernel driver relies on Linux kernel generic allocation support (CONFIG_GENERIC_ALLOCATION) to allocation/release of shared memory physical chunks. OP-TEE Linux kernel driver relies on Linux kernel dma-buf support (CONFIG_DMA_SHARED_BUFFER) to track shared memory buffers references.

3.5.6. *OP-TEE client API*

The TEE Client API describes and defines how a client running in a rich operating environment (REE) should communicate with the TEE. To identify a trusted application (TA) to be used, the client provides an UUID. All TA's exposes one or several functions, which correspond to a so-called commandID, which is also sent by the client.

The TEE context is used for creating a logical connection between the client and the TEE. The context must be initialized before the TEE session can be created. When the client has completed a job running in secure world, it should finalize the context and thereby also release resources.

Sessions are used to create logical connections between a client and a specific trusted application. When the session has been established, the client has opened up

the communication channel towards the specified trusted application identified by the UUID. At this stage, the client and the trusted application can start to exchange data.

3.5.7. *TEE internal core API*

The internal core API is the API that is exposed to the trusted applications running in the secure world. The TEE internal API consists of four major parts:

1. trusted storage API for data and keys;
2. cryptographic operations API;
3. time API;
4. arithmetical API.

All trusted applications (TA's) are signed with the pre-generated 2048-bit RSA development key (private key). This key is located in the keys folder (in the root of `optee_os.git`) and is named `default_ta.pem`. This key must be replaced with your own key and you should never ever check-in this private key in the source code tree when in use in a real product. The recommended way to store private keys is to use some kind of HSM (Hardware Security Module), but an alternative would be temporary put the private key on a computer considered as secure when you are about to sign TA's intended to be used in real products.

3.5.8. *Root and chain of trust*

To be able to assure that devices are running, the (untampered) binaries were originally intended to run some kind of trust anchor on the devices needs to be established. The most common way of doing that is to put the root public key in some read-only memory on the device. Quite often, SoC/OEM stores public key(s) directly or the hash(es) of the public key(s) in one-time-programmable memory (OTP). When the boot ROM (which indeed needs to be ROM) is about to load the first-stage bootloader, it typically reads the public key from the software binary itself, hash the key and compare it to the key in OTP. If they are matching, then the boot ROM can be sure that the first-stage bootloader was indeed signed with the corresponding private key.

3.5.9. *Hardware unique key*

Most devices have some kind of hardware unique key (HUK) that is mainly used to derive other keys. The HUK could, for example, be used when deriving keys used in secure storage and so on. The important thing with the HUK is that it needs to be well protected, and in the best case, the HUK should never ever be readable directly from

software, not even from the secure side. There are different solutions to this, crypto accelerator might have support for it, or it could involve another secure co-processor.

In OP-TEE, the HUK is just stubbed and you will see that in the function called `tee_otp_get_hw_unique_key()` in `core/include/kernel/tee_common_otp.h`. In a real secure product, you must replace this with something else. If your device lacks the hardware support for an HUK, then you must at least change this to something else than just zeroes. But remember it is not a good secure practice to store a key in software, especially not the key that is the root for everything else, so this is not something we recommend that you should do.

3.5.10. *Secure clock*

The Time API in GlobalPlatform Internal Core API specification defines three sources of time: system time, TA persistent time and REE time. The REE time is by nature considered as an unsecure source of time, but the other two should in a fully trustable hardware make use of trustable source of time, i.e., a secure clock. Note that from GlobalPlatform point of view it is not required to make use of a secure clock, i.e. it is OK to use time from REE, but the level of trust should be reflected by the *gpd.tee.systemTime.protectionLevel* property and the *gpd.tee.TAPersistentTime.protectionLevel* property (100=REE controlled clock, 1000=TEE controlled clock). Therefore, the functions that we need to pay attention to are `tee_time_get_sys_time()` and `tee_time_get_ta_time()`. If the hardware has a secure clock, then this implementation may change there to instead use the secure clock (and then update the property accordingly, i.e. `tee_time_get_sys_time_protection_level()` and the variable `ta_time_prot_lvl` in `tee_svc.c`).

3.5.11. *Cryptographic operations*

By default, OP-TEE uses a software crypto library (currently mbed TLS and LibTomCrypt), while there is the ability to enable crypto extensions that were introduced with ARMv8-A (if the device is capable of that). By default, OP-TEE is configured with a software PRNG. The entropy is added to software PRNG at various places, but unfortunately, it is still quite easy to predict the data added as entropy. As a result, unless the RNG is based on hardware, the generated random will be quite weak.

In OP-TEE, TAs can use services accessible through GlobalPlatform Internal Core API implemented in *libutee*. TAs are statically linked against *libutee*, which wraps the API functions around assembler macros to OP-TEE OS system calls. The library provides interfaces to secure storage, time, arithmetic and cryptographic operations.

The secure storage API encrypts data objects by the use of a secure storage service. The encryption process involves three keys: secure storage key (SSK), trusted application storage key (TSK) and file encryption key. The SSK is generated from the hardware unique key and is used to derive TSKs. Each TA has a TSK that is generated from the SSK and the TA's UUID. Both SSK and TSK are generated using the HMAC SHA256 algorithm. Finally, for every created file, an FEK is generated from the pseudo-random number generator. The encrypted data objects are then transferred to the tee-supplicant by a series of remote procedure calls and stored in a special file. OP-TEE further provides TAs with libraries for TLS and SSL protocols (*libmbedtls*) (ARM Ltd. 2019), arithmetic (*libmpa*) and a subset of ISO C functions (*libutils*). Once the REE application has no further service requests, the session is terminated and the context is destroyed.

3.6. LoRaWAN gateway with STM32MP1

The LoRaWAN gateway is realized through integrating the RAK831 RF front-end LoRaWAN gateway concentrator module (Shenzhen RAKwireless Technology Co., Ltd 2020) with the host system STM32MP1-DK2 connected via SPI interface. The LoRaWAN gateway, shown in Figure 3.4, connects to the *The Things Network* (TTN) where it publishes temperature and humidity, which are sent by a LoRa end-node.

Figure 3.4. *LoraWAN gateway using an RAK831 RF with a GPS (top two shields), the STM32MP1 and a 3G modem (bottom shield) when a wired connection is not available*

The gateway packet forwarder executes inside the STM32MP1 platform, which supports the full trusted boot chain and OP-TEE. The packet forwarder has been

extended by using OP-TEE infrastructure for secure storage with the *gwsecstore* library, as depicted in Figure 3.5. We developed this library consistent with the specifications of GlobalPlatform's TEE Internal Core API, to enable the packet forwarder to secure the gateway identifier. The gateway packet forwarder requires a *gateway ID*, as configured by the TTN network to initialize and connect to it. This *gateway ID* is now provided by the trusted application part of the packet forwarder. The API calls of the *gwsecstore* library follow next.

```
- TEEC_Result read_secure_object(char *id, char *data, size_t data_len);
- TEEC_Result write_secure_object(char *id, char *data, size_t data_len);
- TEEC_Result delete_secure_object(char *id);
```

Initially, in the packet forwarder setup phase, the *gateway ID*, which is retrieved from the TTN, is stored securely, while the created object ID is stored to the file system. Thus, to connect to the TTN, the packet forwarder retrieves the *gateway ID* through the *gwsecstore* library by providing the known object ID and stored object data length.

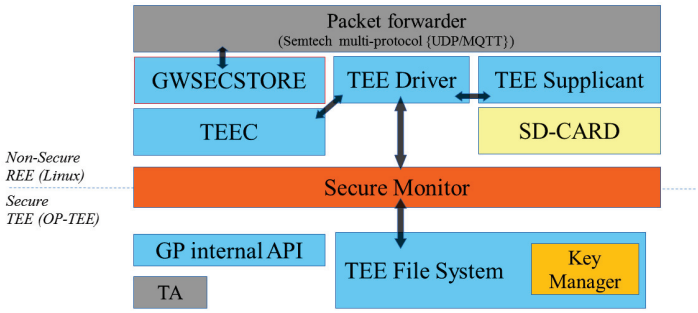

Figure 3.5. *Execution of gateway packet forwarder in OP-TEE enabled organization based on the STM32MP1 platform*

To implement secure storage in OP-TEE, it is implemented according to what has been defined in GlobalPlatform's trusted storage specification, which must provide a minimum level of protection against rollback attacks. Currently, the REE filesystem is used while in the near future we will make use of the replay protected memory block (RPMB)-based secure storage.

3.7. Discussion and future scope

We anticipate several counter–measures against different attacks. Through exploiting trusted boot chain and OP-TEE, we address security operation of the

LoRaWAN gateway according to the specified threat model. However, consider that ARM's TrustZone technology is designed mainly to resist software attacks and is difficult to resist physical attacks (Carru 2017). For systems that use sensitive data for a short amount of time (e.g. cryptographic keys), zeroing out the data immediately after use (Halderman *et al*. 2009) would significantly reduce the risk from Rowhammer-like attacks. While this countermeasure is effective for protecting short lived data, it cannot be used for data that reside in memory for a long duration.

In regard to rapid deployment of LoRaWAN since 2018, we work towards developing features such as firmware updating over the air. However, lack of OTA firmware updates is challenging for full migration to LoRaWAN v1.1 even if all the deployed end devices are capable of running version 1.1. Potential security problems caused by backward compatibility of LoRaWAN are twofold: a fallback may enable the exploitation of vulnerabilities associated with the older version, or it may break a feature introduced in the new version.

Even though LoRaWAN security design adheres to state-of-the-art principles, use of standards, NIST-approved algorithms and end-to-end security, secure devices of low complexity and cost must also be future-proof. The full firmware stack executing in these devices should be authentic and trustworthy during all its lifetime. This guarantee, combined with mutual device authentication and servers' security countermeasures, can ensure that network traffic is coming from a legitimate device, is not comprehensible to eavesdroppers and has not been captured and replayed by rogue actors.

3.8. Acknowledgments

The research leading to these results received funding from the European Union (EU) Horizon 2020 project AVANGARD (advanced manufacturing solutions tightly aligned with business needs) under grant agreement no. 869986.

3.9. References

Abomhara, M. and Køien, G.M. (2015). Cyber security and the internet of things: Vulnerabilities, threats, intruders and attacks. *Journal of Cyber Security*, 4, 65–88.

ARM Ltd. (2009). ARM Security Technology Building a Secure System using TrustZone® Technology [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

ARM Ltd. (2019). Mbed TLS [Online]. Available: https://tls.mbed.org/.

Augustin, A., Yi, J., Clausen, T.H., and Townsley, W. (2016). A study of LoRa: Long range & low power networks for the Internet of things. *Sensors*, 16, 1466.

Bhat, P. and Dutta, K. (2019). A survey on various threats and current state of security in android platform. *ACM Computing Surveys*, 52, 1–35.

Bishop, M., Engle, S., Howard, D., and Whalen, S. (2012). A taxonomy of buffer overflow characteristics. *IEEE Transactions on Dependable and Secure Computing - TDSC*, 9(3), 305–317.

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to RISC. *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 27–38.

Carru, P. (2017). Attack TrustZone with Rowhammer, GreHack [Online]. Available: https://grehack.fr/data/2017/slides/GreHack17_Attack_TrustZone_with_Rowhammer.pdf.

Chen, K., Zhang, S., Li, Z., Zhang, Y., Deng, Q., Ray, S., and Jin, Y. (2018). Internet-of-things security and vulnerabilities: Taxonomy, challenges, and practice. *Journal of Hardware and Systems Security*, 2(2), 97–110.

EPN Solutions. (2014). Analysis tools for DDR1, DDR2, DDR3, embedded DDR and Fully Buffered DIMM modules [Online]. Available: http://www.epnsolutions.net/ddr.html.

Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., and Felten, E. (2009). Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5), 91–98.

Hassija, V., Chamola, V., Saxena, V., Jain, D., Goyal, P., and Sikdar, B. (2019). A survey on IoT security: Application areas, security threats, and solution architectures. *IEEE Access*, 7, 82721–82743.

Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. (2016). ARMageddon: Cache attacks on mobile devices. *Proceedings of the 25th USENIX Conference on Security Symposium*, pp. 549–564.

LoRa Alliance Technical Committee. (2017), LoRaWAN ™ 1.1 Specification [Online]. Available: https://lora-alliance.org/resource-hub/lorawanr-specification-v11.

Palattella, M.R., Dohler, M., Grieco, A., Rizzo, G., Torsner, J., Engel, T., and Ladid, L. (2016). Internet of things in the 5G Era: Enablers, architecture, and business models. *IEEE Journal on Selected Areas in Communications*, 34(3), 510–527.

Raza, U., Kulkarni, P., and Sooriyabandara, M. (2017). Low power wide area networks: An overview. *IEEE Communications Surveys Tutorials*, 19(2), 855–873.

Reynders, B., Meert, W., and Pollin, S. (2016). Range and coexistence analysis of long range unlicensed communication. *Proceedings of the 23rd International Conference on Telecommunications (ICT)*, pp. 1–6.

Shenzhen RAKwireless Technology Co., Ltd. (2020). RAK831 LoRaWAN® Gateway Concentrator [Online]. Available: https://doc.rakwireless.com/datasheet/rakproducts/-.

Sisinni, E., Saifullah, A., Han, S., Jennehag, U., and Gidlund, M. (2018). Industrial Internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, 14(11), 4724–4734.

SonicWall (2019). SonicWall Cyber Threat Report [Online]. Available: https://www.sonicwall.com/resources/white-papers/2019-sonicwall-cyber-threat-report/.

ST (2019). TF-A Overview [Online]. Available: https://wiki.st.com/stm32mpu/wiki/TF-A_overview.

TrustedFirmware Association (2019). Trusted Firmware-A Documentation [Online]. Available: https://trustedfirmware-a.readthedocs.io/en/latest/.

van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. (2016). Drammer: Deterministic rowhammer attacks on mobile platforms. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1675–1689.