Embedded System Architecture

Software security techniques
Software security techniques

What is security?

In very abstract terms, the term **security** can be used to cover a number of very different underlying features of a design. However, it is essentially a property of the system which ensures that resources of value cannot be copied, damaged, or made unavailable to genuine users.

Every system design will require a different set of security properties, depending on the type and value of the **assets** it is trying to **defend** against malicious **attack**.
Software security techniques

**Asset**
A resource of value which is worth protecting. An asset may be a tangible object, such as a user password, or may be an intangible asset, such as network availability.

**Attack**
The act of intentionally trying to acquire, damage or disrupt an asset which you do not have permission to access.
An attack may include the use of malicious software, hardware monitoring and hardware tampering.
Software security techniques

**Confidentiality**

An asset that is confidential cannot be copied or stolen by a defined set of attacks.

This property is essential for assets such as passwords and cryptographic keys.
Software security techniques

**Integrity**
An asset that has its integrity assured is defended against modification by a defined set of attacks.
This property is essential for some of the on-device root secrets which the rest of the system security is based on, and for the security software once it is running.
**Authenticity**

In some circumstances a design cannot provide integrity, so instead provides the property of authenticity. In this case the value of the asset can be changed by an attacker, but the defender will be able to detect the change before the asset is used and hence before the attack can cause a security fault.

This property is essential for security software. If an attacker can tamper with the program code before it is loaded into a safe execution location, without being detected, then the security provided by the software can be bypassed.
Software security techniques

How are devices attacked?

The most important factor in the design is the mechanisms that are used to perform the attacks. Different mechanisms of performing attacks are known as attack vectors, and these break down into three classes – hack attacks, shack attacks and lab attacks.
Software security techniques

**Hack attack**

A hack attack is one where the hacker is only capable of executing a software attack. Examples of hack attacks include viruses and malware which are downloaded to the device via a physical or a wireless connection.

In many cases of a successful hack attack the device user inadvertently approves the installation of the software that then executes the attack. This is either because the malware pretends to be a piece of the software that the user does want to install, or because the user does not understand the warning messages displayed by the operating environment.
Software security techniques

**Shack attack**

A shack attack is a low-budget hardware attack, using equipment that could be bought on the high street from a store such as Radio Shack. In these scenarios the attackers have physical access to the device, but not enough equipment or expertise to attack within the integrated circuit packages.

The attackers can attempt to connect to the device using JTAG debug, boundary scan I/O, and built-in self test facilities. They can passively monitor the system using logic probes and network analyzers to snoop bus lines, pins and system signals. The attackers may also be able to perform simple active hardware attacks, such as forcing pins and bus lines to be at a high or low voltage, reprogramming memory devices, and replacing hardware components with malicious alternatives.
Software security techniques

**Lab attack**

The lab attack vector is the most comprehensive and invasive. If the attacker has access to laboratory equipment, such as electron microscopes, they can perform unlimited reverse engineering of the device. It must be assumed that the attacker can reverse engineer transistor-level detail for any sensitive part of the design - including logic and memories.

Attackers can reverse engineer a design, attach microscopic logic probes to silicon metal layers, and glitch a running circuit using lasers or other techniques. Attackers can also monitor analog signals, such as device power usage and electromagnetic emissions, to perform attacks such as cryptographic key analysis.
Software security techniques

**Lab attack**
Lab attacks are outside of the scope of the protection provided by TrustZone technology, although a SoC using TrustZone can be used in conjunction with an ARM SecurCore© smartcard if protection against physical attacks is needed for some assets.
Software security techniques

Security of the:
- OS / Firmware.
- Input Data.
- Application.
- Output Data.

Hardware security of application/OS/Firmware:
- **Security Bit** – access lock to the interfaces for code reading and modifying. Re access to the device is possible only after erasing program memory;
- **Verify Device ID** – verification device ID stored in a file with the contents of "signature byte".
Software security techniques

**Data security:**

- Data Encryption Standard (DES) CPU instruction
- Advanced Encryption Standard (AES) crypto module
- DES Instruction
  - Encryption and decryption
  - DES supported
  - Encryption/decryption in 16 CPU clock cycles per 8-byte block
- AES crypto module
  - Encryption and decryption
  - Supports 128-bit keys
  - Supports XOR data load mode to the state memory
  - Encryption/decryption in 375 clock cycles per 16-byte block
Software security techniques

- **Hardware keys**
- **USB keys** (HASP - Hardware Advanced Software Protection – dongles).

**Cryptographic algorithms:**
- RSA 512/1024/RSA 2048 bit (Rivest, Shamir, Adleman)
- ECDSA 192/256 bit - Elliptic Curve Digital Signature Algorithm
- DES/3DES
- AES 128/192/256 bit
- SHA-1 / SHA-256 - Secure Hash Algorithm;
Software security techniques

- Cryptographic API:
  - Microsoft Crypto API (CAPI), Cryptography API: Next Generation (CNG)
  - Microsoft Smart Card Mini Driver
  - PKCS#11
  - PC/SC
Software security techniques

Data integrity refers to maintaining and assuring the accuracy and consistency of data over its entire life-cycle, and is a critical aspect to the design, implementation and usage of any system which stores, processes, or retrieves data. The term data integrity is broad in scope and may have widely different meanings depending on the specific context – even under the same general umbrella of computing.

Data integrity can be divided into two categories:

- Physical integrity;
- Logical integrity.
Software security techniques

**Physical integrity** - deals with challenges associated with correctly storing and fetching the data itself. Challenges with physical integrity may include:

- electromechanical faults,
- design flaws,
- material fatigue,
- corrosion,
- power outages,
- natural disasters,
- acts of war and terrorism,
- other special environmental hazards:
  - ionizing radiation,
  - extreme temperatures,
  - pressures and g-forces.
Software security techniques

Ensuring physical integrity includes methods such as:

- redundant hardware,
- an uninterruptible power supply,
- certain types of RAID arrays,
- radiation hardened chips,
- Error Checking and Correction, Error Correction Code, ECC memory,
- use of a clustered file system,
- using file systems that employ block level checksums such as Zettabyte File System ZFS,
- storage arrays that compute parity calculations such as Exclusive OR
- use a Cryptographic hash function and even having a watchdog timer on critical subsystems.
Software security techniques

- Error detecting algorithms known as error-correcting codes.
- Damm algorithm or Luhn algorithm.
- Transcription errors - hash functions.

In production systems these techniques are used in combination to ensure various degrees of data integrity.
Software security techniques

**Logical integrity** - concerned with the correctness or rationality of a piece of data, given a particular context.

This includes following topics:

- referential integrity;
- entity integrity in a relational database;
- correctly ignoring impossible sensor data in robotic systems.

Challenges include:

- software bugs,
- design flaws,
- human error.
Software security techniques

Well-controlled, and well-defined data-integrity system increases:
- stability;
- performance;
- reusability;
- maintainability.
Software security techniques

*Intel® Trusted Execution Technology Intel® (TXT):*

- is the hardware basis for mechanisms that validate platform trustworthiness during boot and launch, which enables reliable evaluation of the computing platform and its protection level;
- is compact and difficult to defeat or subvert;
- allows for flexibility and extensibility to verify the integrity of platform components during:
  - boot,
  - launch,
  - including BIOS,
  - operating system loader,
  - hypervisor.
Software security techniques

*Intel® Trusted Execution Technology Intel® (TXT)*:

- reduces the overall attack surface for both individual systems and compute pools;
- provides a signature that represents the state of an intact system’s launch environment;
- the corresponding signature at the time of future launches can then be compared against that known-good state to verify a trusted software launch, to execute system software, and to ensure that cloud *Infrastructure as a Service (IaaS)* has not been tampered with;
- security policies based on a trusted platform or pool status can then be set to restrict (or allow) the deployment or redeployment of *virtual machines (VMs)* and data to trusted platforms with known security profiles.
Software security techniques

Intel® Trusted Execution Technology Intel® (TXT):
- builds trust into a known software environment and thus ensures that the software being executed hasn’t been compromised.

Intel® Trusted Execution Technology Intel® (TXT):
- must be enabled at multiple levels, including hardware, BIOS, OS, and hypervisor. Attestation and cloud-management software work with those components to enable management and reporting for the trusted system environment.

Figure 1. Simplified Intel® TXT component diagram.
Software security techniques

*Intel® Trusted Execution Technology Intel® (TXT)*

**Hardware-Layer Requirements:**

- Intel® TXT and Intel® VT-x support
- Intel® VT-d, Intel® VT-x, Intel® TXT, and TPM support
- Authentication Code Modules (ACM) created and signed by Intel
- BIOS
- TPM
- MLE
- Measured Launched Environment (MLE) for 3rd party software OS, VM, etc.
- Trusted Platform Module from 3rd party (TCG* Compliant)

*Figure 2. Component overview of a Measured Launch Environment (MLE).*
Software security techniques

**Intel® Trusted Execution Technology Intel® (TXT)**

**Hardware-Layer Requirements:**

- **Processor** – Intel® Xeon® processor, with support for Intel® TXT and Intel VT-x (VMX and SMX). These features were introduced with the Intel Xeon processor 5600 series.

- **Chipset** – **Trusted Platform Module (TPM)** must be integrated with the chipset. The chipset and the TPM work together to ensure that the measurements and security properties of the system are not spoofed by untrusted components. TPMs are devices manufactured by various third-party silicon providers that attach to the chipset via the Low Pin Count (LPC) bus or Serial Peripheral Interface (SPI), and they provide a number of security functions. TPM Details can be found in the main **TPM specification**.
Software security techniques

**Intel® Trusted Execution Technology Intel® (TXT)**

**Hardware-Layer Requirements:**

- **Trusted Platform Module (TPM) manufacturers:**
  - Atmel
  - Broadcom
  - Infineon
  - Infineon
  - Sinosun
  - Sinosun
  - STMicroelectronics
  - Winbond
Software security techniques

*Intel® Trusted Execution Technology* Intel® (TXT)

**Hardware-Layer Requirements:**

- **BIOS** – *Intel® TXT* and the TPM must enabled within BIOS. *Authenticated Code Modules (ACMs)* created and signed by Intel must be present inside the BIOS. The ACM contains platform-specific code which is authenticated to the chipset and executed in an isolated environment within the processor and the trusted environment (*authenticated code mode*), enabling the ACM to perform secure tasks.
Software security techniques

*Intel® Trusted Execution Technology Intel® (TXT)*

Software-Layer Requirements:

- **Operating System and Hypervisor** – An Measured Launch Environment (MLE) provides a software-verification process to attest that all of the critical components of the pre-OS launch environment have been verified against a known good source, ensuring a secure chain of custody from the moment a system is powered on until the system kernel or hypervisor takes control.

An Intel TXT – aware hypervisor provides isolation for host OSs and applications.
Software security techniques

**Intel® Trusted Execution Technology Intel® (TXT)**

**Measured Launch Environment (MLE):**

- The MLE includes the following components:
  - **Authenticated Code Module ACM**, which performs the measured launch, starting the dynamic chain of trust,
  - **Server platform**, including BIOS code, BIOS configuration, **System Management Mode SMM** code, option ROM code and configuration, system state, master boot record, and boot configuration,
  - **Initial system software code** (referred to as MLE code) that sets up the platform to protect the OS/hypervisor kernel code.

A successful measured launch requires that the ACM is valid, the server platform (as measured by the BIOS) has passed the launch control policy, and the MLE code measurement has passed the launch control policy.
Software security techniques

Intel® Trusted Execution Technology Intel® (TXT) Measured Launch Phases:

![Diagram showing measured launch timeline]

Figure 3. Measured launch timeline.
Software security techniques

*Intel® Trusted Execution Technology Intel® (TXT)*

**Measured Launch Phases:**

- **Pre-boot phase** is performed by the system firmware (BIOS/UEFI).

![Diagram of measured launch timeline](image)

*Figure 3. Measured launch timeline.*
Software security techniques

Intel® Trusted Execution Technology Intel® (TXT)
Measured Launch Phases:

- **Initial Program Loader (IPL)** represents the normal boot process up until the time, that the process would normally load and execute the kernel. The first module executed should now be the **Trusted Boot (tBoot)** module:

![Diagram showing measured launch timeline]
Software security techniques

**Intel® Trusted Execution Technology Intel® (TXT)**

**Measured Launch Phases:**

- **TBOOT Pre-Launch** is the part of tBoot that determines whether a measured launch is possible and sets up the platform to perform the measured launch.

- **TBOOT Launch** is the part of tBoot that starts the measured launch process by executing the `GETSEC [SENTER]` instruction. This execution starts the dynamic chain of trust measurements, extending the root of trust measurement into PCR 17 and measures tBoot Post Code into PCR 18.

![Figure 3. Measured launch timeline.](image-url)
Software security techniques

**Intel® Trusted Execution Technology Intel® (TXT) Measured Launch Phases:**

- **TBOOT Post Launch** is the code that executes as a result of the measured launch. Its purpose is to securely bring the platform to a protected, usable state. This is the first system code to be measured, and it starts the chain of measurements.

- **OS/VMM Post Launch** includes the kernel and any other modules that need to be loaded. The kernel is responsible for measuring other modules before they are executed if they have not already been measured by the tBoot code.

- **Regular Operation** commences after successful launch, when the OS/hypervisor performs its primary functionality (i.e., the same functionality as would occur in an environment without Intel TXT). However, there are some additional capabilities available to the OS/hypervisor, which must protect Intel TXT resources.
Software security techniques

**Intel® Trusted Execution Technology Intel® (TXT)**

**Measured Launch Phases:**

- **MLE Shutdown** occurs before turning off or resetting the platform; there are certain steps the OS/hypervisor is required to take to exit the secure environment. While this phase could be followed by another measured launch, it is typically followed by a platform reset or power cycle.
Software security techniques

Intel® Trusted Execution Technology Intel® (TXT)

Table 1. Contents stored in various PCRs.

<table>
<thead>
<tr>
<th>PCR Numbers</th>
<th>Description of Contents</th>
</tr>
</thead>
</table>
| 0-7          | Information associated with the BIOS  
Example: the value of PCR 6 would change as a result of an S3 shutdown/resume |
| 8-14         | Information associated with OSs |
| 17-21, 23    | Information associated with a Dynamic Operating System  
Example: PCRs 19-21 define VMware-specific information |
| 22           | Geo-tagging index, enabling any Dynamic Operating System to define the geographic location of a platform; an extension/dynamic PCR that can be written to by the Dynamic OS |
Software security techniques

*Intel® Trusted Execution Technology Intel® (TXT)*
Software security techniques

**Management of Trusted Systems with Use Case** – Software cloud-management solutions are available for various use cases, including policy and compliance enforcement, automation, auditing, reporting, workload migration, etc. Trusted compute pools can be combined with these solutions to enhance security.
Software security techniques

**ARM Security Technology**

*Building a Secure System using TrustZone® Technology*
Software security techniques

**ARM Security Technology**

System designs for embedded devices are complicated, including multiple independent processor cores, secondary bus masters such as DMA engines, large numbers of memory, peripheral bus slaves. In addition to these functional components there is typically a parallel system infrastructure that provides invasive and non-invasive debug capabilities and Built-In-Self-Test (BIST) facilities.
Software security techniques

ARM Security Technology

Figure 2-1: A simplified schematic of a typical cellular handset SoC design
Software security techniques

ARM Security Technology

Each of these subsystems in the platform has to be designed and integrated in such a way that it works with the security solution, rather than developing each sub-system independently to the security requirements. If the threat model for a device indicates that it needs to protect against shack attacks, there is no point securing only the functional part of the system. An attacker with unrestricted access to a debug port can bypass many of the functional protections that may exist.
Software security techniques

**ARM Security Technology**

External hardware module of security

The classic security solution for embedded applications is the inclusion of a dedicated hardware security module, or trusted element, that is outside of the main SoC. For example:

- a SIM card in a mobile handset
- a conditional access smartcard in a set-top box.
Software security techniques

ARM Security Technology

External hardware security module – Advantages

- the assets are protected inside a physical device designed for robust security;
- the use of a completely separate design and manufacturing flow allows to obtain high levels of tamper resistance and physical security;
- the smartcard manufacturing and personalization processes have frequently been formally certified through approved evaluation schemes.
Software security techniques

**ARM Security Technology**

External hardware security module – Disadvantages

- the use of smartcards adds significant effort to the design process and compromise the area, power efficiency and performance of a device;
- the manufacturing methods that give the physical security also force a lower processor performance, in the region of 5 – 20 MHz, and small quantities of RAM within the device;
Software security techniques

**ARM Security Technology**

TrustZone hardware security:
- is based on the concept of a trusted platform;
- a hardware architecture that extends the security infrastructure throughout the system design;
- the TrustZone architecture enables any part of the system to be made secure, enabling an end-to-end security solution that includes functional units and the debug infrastructure.
Software security techniques

**ARM Security Technology**

**TrustZone Hardware Architecture:**
- aims to provide a security framework that enables a device to counter many of the specific threats that it will experience;
- provides the infrastructure foundations that allow a SoC designer to choose from a range of components that can fulfill specific functions within the security environment.
Software security techniques

**ARM Security Technology**

TrustZone Hardware Architecture:
- the security of the system is achieved by partitioning all of the SoC’s hardware and software resources so that they exist in one of two worlds - the Secure world for the security subsystem, and the Normal world for everything else;
- hardware logic present in the TrustZone - enabled AMBA3 (Advanced Microcontroller Bus Architecture) AXI™ (Advanced eXtensible Interfacebus) fabric ensures that no Secure world resources can be accessed by the Normal world components, enabling a strong security perimeter to be built between the two;
Software security techniques

ARM Security Technology

TrustZone Hardware Architecture:

- The second aspect of the TrustZone hardware architecture is the extensions that have been implemented in some of the ARM processor cores.
- These additions enable a single physical processor core to safely and efficiently execute code from both the Normal world and the Secure world in a time-sliced fashion;
- This removes the need for a dedicated security processor core, which saves silicon area and power, and allows high performance security software to run alongside the Normal world operating environment.
Software security techniques

ARM Security Technology

TrustZone Hardware Architecture:

- The final aspect of the TrustZone hardware architecture is a security-aware debug infrastructure which can enable control over access to Secure world debug, without impairing debug visibility of the Normal world;
Software security techniques

ARM Security Technology

The AMBA3 AXI system bus:

- An extra control signal is defined for each of the read and write channels on the main system bus. These bits are known as the Non-Secure, or NS bits, and are defined in the public AMBA3 Advanced eXtensible Interface (AXI) bus protocol specification.
- AWPROT[1]: Write transaction – low is Secure and high is Non-secure.
- ARPROT[1]: Read transaction – low is Secure and high is Non-secure.
Software security techniques

ARM Security Technology

The AMBA3 AXI system bus:

- All bus masters set these signals when they make a new transaction, and the bus or slave decode logic must interpret them to ensure that the required security separation is not violated.

- All Non-secure masters must have their NS bits set high in the hardware, which makes it impossible for them to access Secure slaves. The address decode for the access will not match any Secure slave and the transaction will fail.

- If a Non-secure master attempts to access a Secure slave it is implementation defined whether the operation fails silently or generates an error. Two errors may occur SLVERR (slave error) or DECERR (decode error).
Software security techniques

**ARM Security Technology**

The AMBA3 APB peripheral bus:
- One of the most useful features of the TrustZone architecture is the ability to secure peripherals, such as interrupt controllers, timers, and user I/O devices. This enables the security environment to be extended so that it can solve some of the wider security issues which need more than just a secure data processing environment.
- The AMBA3 specification includes peripheral bus known as the **Advanced Peripheral Bus** (APB), which is attached to the system bus using an AXI-to-APB bridge.
- The AXI-to-APB bridge hardware is responsible for managing the security of the APB peripherals; the bridge must reject transactions of inappropriate security setting and must not forward these requests to the peripherals.
Software security techniques

**ARM Security Technology**

Memory aliasing:

- The addition of the NS bit to the bus transactions, and to any cache tags in the system, can be viewed as providing a 33\textsuperscript{rd} address bit. There is a 32-bit physical address space for Secure transactions and a 32-bit physical address space for Non-secure transactions.

- As with any address space, including those without TrustZone technology, care must be taken to ensure that the 33-bit address space is used in such a way that data remains coherent in all of the locations that it is stored, otherwise data corruption may result.
Software security techniques

ARM Security Technology

Memory aliasing:
- Consider the case where a Secure world master wants to access a Non-secure slave that is cached. A design may implement either of the following choices:
  - The master makes a Non-secure access to the slave.
  - The master makes a Secure access to the slave and the Non-secure slave accepts the Secure transaction. The slave treats these accesses as Non-secure.
Software security techniques

**ARM Security Technology**

Memory aliasing:
- In the second design the hardware must support address space aliasing. In this aliased memory system the same memory location appears as two distinct locations in the address map, one Secure and one Non-secure. As a result, it is possible to have multiple values representing the same data present in the cache simultaneously. For modifiable data this aliasing causes coherency problems; if one copy of the data is modified while the other exists in the cache you will have versions of the data but both will be different. System designers must be aware of potential data coherency problems, and must take steps to avoid them.
Software security techniques

**ARM Security Technology**

Processor architecture:
- ARM processors that implement the architectural Security Extensions. Currently these are the:
  - ARM1176JZ(F)-S™ processor;
  - Cortex™ - A8 processor;
  - Cortex™ - A9 processor;
  - Cortex™ - A9 MPCore™ processor.
- Each of the physical processor cores in these designs provides two virtual cores, one considered Non-secure and the other Secure, and a mechanism to robustly context switch between them, known as monitor mode.
Software security techniques

**ARM Security Technology**

Processor architecture:
- The value of the NS bit sent on the main system bus is indirectly derived from the identity of the virtual core that performed the instruction or data access.

![Diagram showing modes in an ARM core implementing the Security Extensions](image)

*Figure 3-1: Modes in an ARM core implementing the Security Extensions*
Secure operating system:
A dedicated operating system in the Secure world is a complex, but powerful, design. It can simulate concurrent execution of multiple independent Secure world applications, run-time download of new security applications, and Secure world tasks that are completely independent of the Normal world environment.
Secure operating system:
The most extreme version of these designs closely resembles the software stacks that would be seen in a SoC with two separate physical processors in an Asymmetric Multi-Processing (AMP) configuration. The software running on each virtual processor is a standalone operating system, and each world uses hardware interrupts to preempt the currently running world and acquire processor time.
Software security techniques

**ARM Security Technology**

Secure operating system:
A tightly integrated design, which uses a communications protocol that associates Secure world tasks with the Normal world thread that requested them, can provide many of the benefits of a Symmetric Multi-Processing (SMP) design. In these designs a Secure world application could, for example, inherit the priority of the Normal world task that it is assisting. This would enable some form of soft real-time response for media applications.
Software security techniques

**ARM Security Technology**

Secure operating system:

![Diagram of ARM Security Technology](image)

*Figure 5-1: A possible architecture with an independent Secure world OS*
Software security techniques

**ARM Security Technology**

Secure software and multiprocessor systems

![Diagram of secure software and multiprocessor systems]

**Figure 5-3** An example multiprocessor software architecture
Software security techniques

ARM Security Technology

The TrustZone API:

Figure 5-4: Two example systems that might make use of TrustZone API
Software security techniques

**ARM Security Technology**

**Boot sequence:**

1. **Device Power On**
2. **ROM SoC Bootloader (First Level Boot)**
3. **Flash Device Bootloader (Second Level Boot)**
4. **Secure World OS Boot**
5. **Normal World Bootloader**
6. **Normal World OS Boot**
7. **System Running**

*Any Secure world bootloaders may need to implement secure boot protocols to defend against some attacks.*

*Figure 5-2: A typical boot sequence of a TrustZone-enabled processor*
Texas Instruments Security techniques

**MSP430™ ultra-low power MCUs**

- **Device identity (unique ID)**
- **Debug security (JTAG lock w/ password)**
- **Cryptographic acceleration (AES 128/192/256)**
- **Software IP protection (Debug lock-out, IP encapsulation)**
Texas Instruments Security techniques

**C2000™ real-time control MCUs**

- **Device identity (unique ID)**
- **Debug security (JTAG lock w/ password)**
- **Software IP protection (IP Protected zones)**
Texas Instruments Security techniques

**SimpleLink™ MSP432™ MCUs**

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device identity (unique ID)</td>
</tr>
<tr>
<td>Debug security (JTAG lock w/ password, factory reset configuration)</td>
</tr>
<tr>
<td>Cryptographic acceleration (AES 128/192/256)</td>
</tr>
<tr>
<td>Software IP protection (IP Protected zones)</td>
</tr>
<tr>
<td>Secure firmware &amp; software update (AES-encrypted firmware update/password authentication)</td>
</tr>
</tbody>
</table>
Texas Instruments Security techniques

**SimpleLink™ Wi-Fi® Wireless MCUs**

- Secure boot (authenticated boot, standard secure, root-of-trust public key)
- Device identity (unique private key per device)
- Debug security (JTAG lock)
- Cryptographic acceleration (AES 128/192/256, DES/3DES, SHA 1/2)
- Networking security (WPA2, TLS)
- Secure storage (secure storage)
- Software IP protection (cloning protection)
- Initial secure programming (encrypted firmware flow for manufacturing)
- Secure firmware & software update (file system security, software tamper protection)
Texas Instruments Security techniques

**Sitara™ Processors**

- **Secure boot** (authenticated boot, standard secure, root-of-trust public key)
- **Device identity** (unique ID)
- **Debug security** (JTAG lock)
- **Cryptographic acceleration** (AES 128/92/256, DES/3DES, RSA, ECC, SHA 1/2, TRNG)
- **External memory protection** (encrypted execute-in-place with QuadSPI)
- **Trusted execution environment** (TEE)
- **Secure storage** (TEE)
- **Software IP protection** (Debug lock-out)
- **Initial secure programming** (secure boot, secure flashing)
- **Physical security** (anti-tamper module)
Texas Instruments Security techniques

Automotive Processors

- Secure boot (root-of-trust public key)
- Device identity (unique ID)
- Debug security (JTAG lock w/ password)
- Cryptographic acceleration (AES 128/92/256, DES/3DES, SHA 1/2, TRNG)
- External memory protection (firewalls)
- Secure storage (secure ROM APIs, anti-cloning)
- Software IP protection (Debug lock-out)
Software security techniques - References

Software security techniques - References
