Technical White Paper

Enabling Cybersecurity for High Performance Real-Time Control Systems

Ibukun Olumuyiwa and David Foley

ABSTRACT

As modern automotive and industrial products continue to increase in complexity, performance and connectivity, the need for strong embedded cybersecurity solutions has also increased. Given a growing landscape of threat actors, and evolving regulatory requirements across the world, both chip manufacturers and OEMs must adapt and implement stronger product security without compromising performance. To effectively defend against increasingly sophisticated attacks on embedded hardware and software, a multi-layered approach is required. Elements such as a secure root of trust, secure storage, cryptographic acceleration, trusted execution environments, secure key and code provisioning, and run-time context isolation are essential components of cybersecurity in a modern high-performance real-time microcontroller. The AM26x and F29x microcontroller families from Texas Instruments are designed from the ground up to achieve these security goals, while delivering industry-leading performance for real-time applications without compromise.

Table of Contents

Trademarks

C2000™ and Sitara™ are trademarks of Texas Instruments.
Arm® and TrustZone® are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
All trademarks are the property of their respective owners.

1 Introduction

Modern automotive and industrial products, from cars and trains to servo drives and server power supply units, have grown in complexity, requiring real-time control solutions with higher performance. At the same time, these products have become highly connected, requiring strong cybersecurity solutions to maintain the confidentiality, integrity, authenticity and availability of hardware and software assets and the systems built around them. Additionally, the security of the application during runtime has come into greater focus, with larger and more complex software stacks leading to increased attack surfaces for potential threat actors.

To effectively defend against increasingly sophisticated modern attacks on embedded hardware and software, a comprehensive multi-layered approach is required, effectively establishing a root of trust, providing secure storage for critical assets such as cryptographic keys, creating trusted execution environments for performing security-sensitive operations, secure key and code provisioning, and run-time context isolation and memory protection to mitigate the potential reach of malware in the system. This white paper examines these subjects, and how secure microcontroller architectures can maximize these cybersecurity objectives without compromising performance.

2 The Need for a Comprehensive Security Approach

Cybersecurity in embedded microcontrollers is directly connected to the safety and functionality of the systems they are deployed in. These systems are often used for critical applications, such as automotive systems, medical devices and industrial control systems. These devices can be vulnerable to various types of attacks, such as data and intellectual property theft, denial-of-service, malware injection, remote control, and physical tampering. Even devices not directly connected to the internet can be exploited by these types of attacks, as evidenced by high-profile exploits such as Stuxnet and Rowhammer. In the automotive market, vehicle thefts accomplished using man-in-the-middle attacks and CAN injection exploits have risen in prominence and frequency, highlighting the need for stronger run-time security protections.

The increased complexity of embedded software in these systems also provides more opportunities for attackers to gain a foothold. For instance, while secure debug (“JTAG Lock”) can provide a reasonably strong protection for the boundary of the chip, communication interfaces such as CAN, SPI and I2C can provide potential entry points for malware or malicious input. Additionally, side-channel analysis and fault injection attacks can be used to bypass existing security protections, or even extract secrets and encryption keys. A cryptographically authentic root of trust, known as secure boot, is necessary to defend against malware that can be potentially introduced into the system through unauthorized firmware updates. A wide variety of embedded devices today feature some form of secure boot to ensure firmware integrity before execution, whether code is stored in external or internal Flash memory.

However, even in embedded flash microcontrollers, secure boot alone is not sufficient. Effective context isolation and memory protection serves to protect critical system code from malware introduced through vulnerable external interfaces, such as communication ports.

Additionally, software implementations of cryptographic algorithms have been the subject of many side-channel attacks, such as timing attacks, resulting in exposed secrets. Software cryptographic algorithms may also fail to meet system performance requirements due to their computational complexity. Hardware cryptographic accelerators can significantly improve the performance of these algorithms, and provide built-in protections against common attacks.

3 Cryptographic Functions

Cryptography is at the heart of embedded security, and is the fundamental means by which cybersecurity properties such as confidentiality, integrity and authenticity can be protected. Confidentiality requires encryption using strong algorithms that cannot be deciphered or broken by unauthorized parties. The sender converts the data to be transmitted into undecipherable ciphertext using a cryptographic algorithm and an encryption key, and the receiver must subsequently use the corresponding decryption algorithm and a decryption key to convert the ciphertext back into the original data. Integrity and authenticity are related cybersecurity properties that focus on establishing trust. Integrity ensures that data has not been altered or corrupted during transmission or storage, while authenticity ensures that the data comes from a legitimate and trusted source. Hashing functions and digital signatures are needed to establish the integrity and authenticity of code or data. To achieve cybersecurity goals and effectively protect the confidentiality, integrity and availability of secured assets, one or more of these cryptographic functions can be used to build an embedded cybersecurity solution.

3.1 Encryption and Decryption

There are two types of encryption ciphers: symmetric and asymmetric. Symmetric encryption relies on a single shared key for both encryption and decryption, and preserves confidentiality so long as the shared key remains secret and is unaltered. However, if the shared key is exposed to a third party, the protected information can be decrypted into plaintext, and secrecy is lost. Maintaining the secrecy of the shared key used in symmetric cryptography is therefore critical to maintaining the confidentiality of private information. The Advanced Encryption Standard (AES) is one of the most widely used symmetric algorithms in the world, with key lengths ranging from 128 to 256 bits. AES has been adopted by the US National Institute of Standards and Technology (NIST).

On the other hand, asymmetric encryption uses a complementary pair of keys, one public and one private. Data encrypted by a public key can only be decrypted by its associated private key. Because asymmetric cryptography is much slower than symmetric, these functions are typically limited to encrypting symmetric keys. Once decrypted by the receiver, the symmetric key can then be used to decrypt the rest of the message. The Rivest-Shamir-Adleman suite of algorithms (RSA) and Elliptic-curve Cryptography (ECC) are the two most common families of asymmetric ciphers.

3.2 Hashing, Digital Signing, and Authentication

In addition to encryption and decryption, a secure system must be able to confirm the integrity of stored code and data assets on the device. Hash algorithms support this objective, reducing a blob of code or data of arbitrary length to a unique fixed-length digest. Without using a key, a cryptographic hash function always generates the same output digest for the same input, and has several important attributes that make it secure:

1. It is highly infeasible to deduce the original input string that was used to compute a given hash value.
2. Given an original message, it is highly infeasible to modify the message in such a way that it generates the same hash value as the original.
3. A cryptographic hash function is resistant to collisions—meaning it is highly improbable for two different inputs to generate the same output.
4. Additionally, any change to the input, even a single bit, results in a drastic change to the output. This is known as the avalanche effect.

By computing a hash digest on an authentication certificate or stored code to be booted and comparing it to a known reference, the system can confirm that the code or data has not been modified since its creation. The Secure Hash Algorithm (SHA) and Message Digest 5 (MD5) are examples of commonly used hash functions. Use of the SHA-1 and MD5 algorithms is discouraged, as successful collision attacks have been demonstrated against these functions. The SHA-2 and SHA-3 algorithms can be used instead to provide strong hash functions. The length of the hash digest is related to the cryptographic strength of the function. All things being equal, longer digests are more secure, at the expense of more computation time.

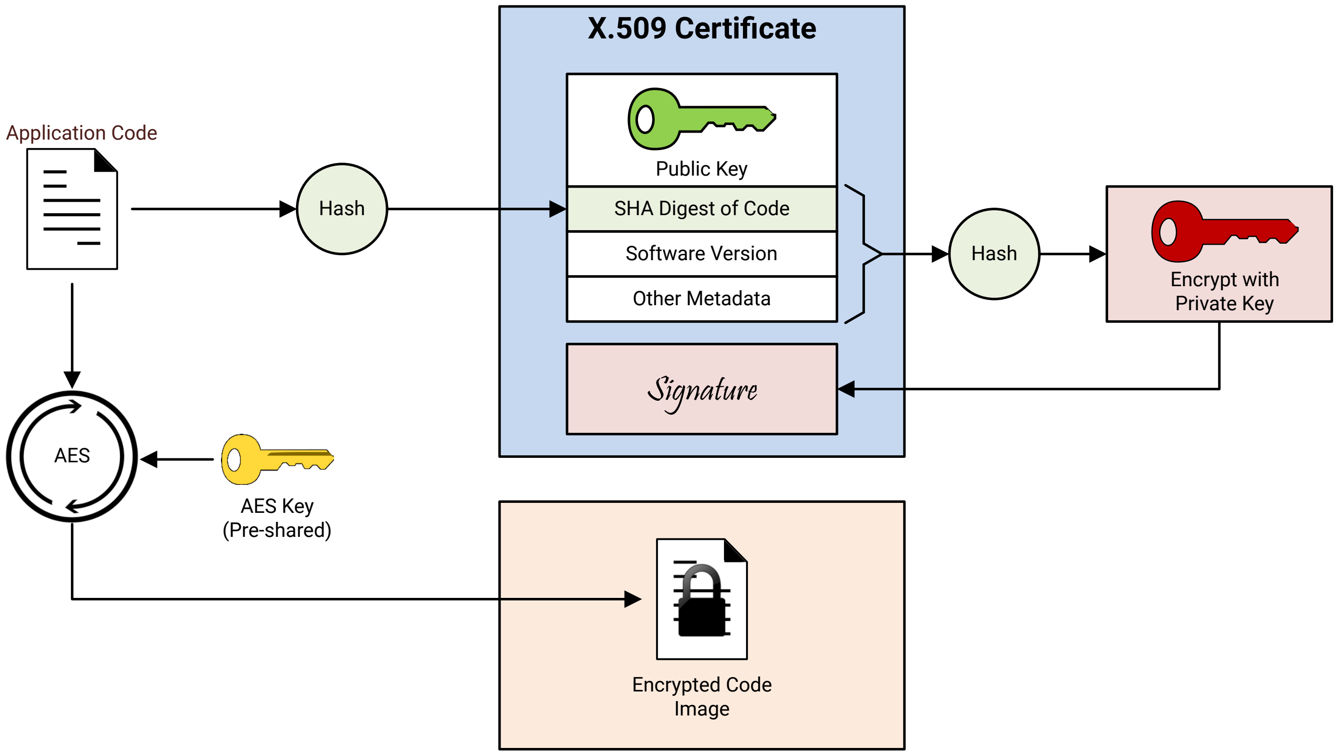

Figure 3-1. Code Encryption and Digital Signing Example

An asymmetric algorithm can be used to create a digital signature from a message, using a private key. A digital signature can be used to confirm the integrity and authenticity of a signed message. However, asymmetric algorithms are much slower than hash functions. Therefore, hashes work well when combined with asymmetric algorithms, by solving the problem of computation time across a large blob of data. Instead of signing the entire blob, a secure hash of the blob can first be computed. The resulting output hash digest is then signed using the sender's private key. The receiver reverses this process using the sender's public key, computes the hash of the received data blob, and authenticates it against the original hash. This process establishes both the integrity and authenticity of the data, while saving on computation time, and is commonly referred to as digital signing. These items are typically stored together with other important metadata in a digital certificate, using an industry-standard format such as X.509. This process is essential for factory provisioning and firmware updates, which are discussed in subsequent sections.

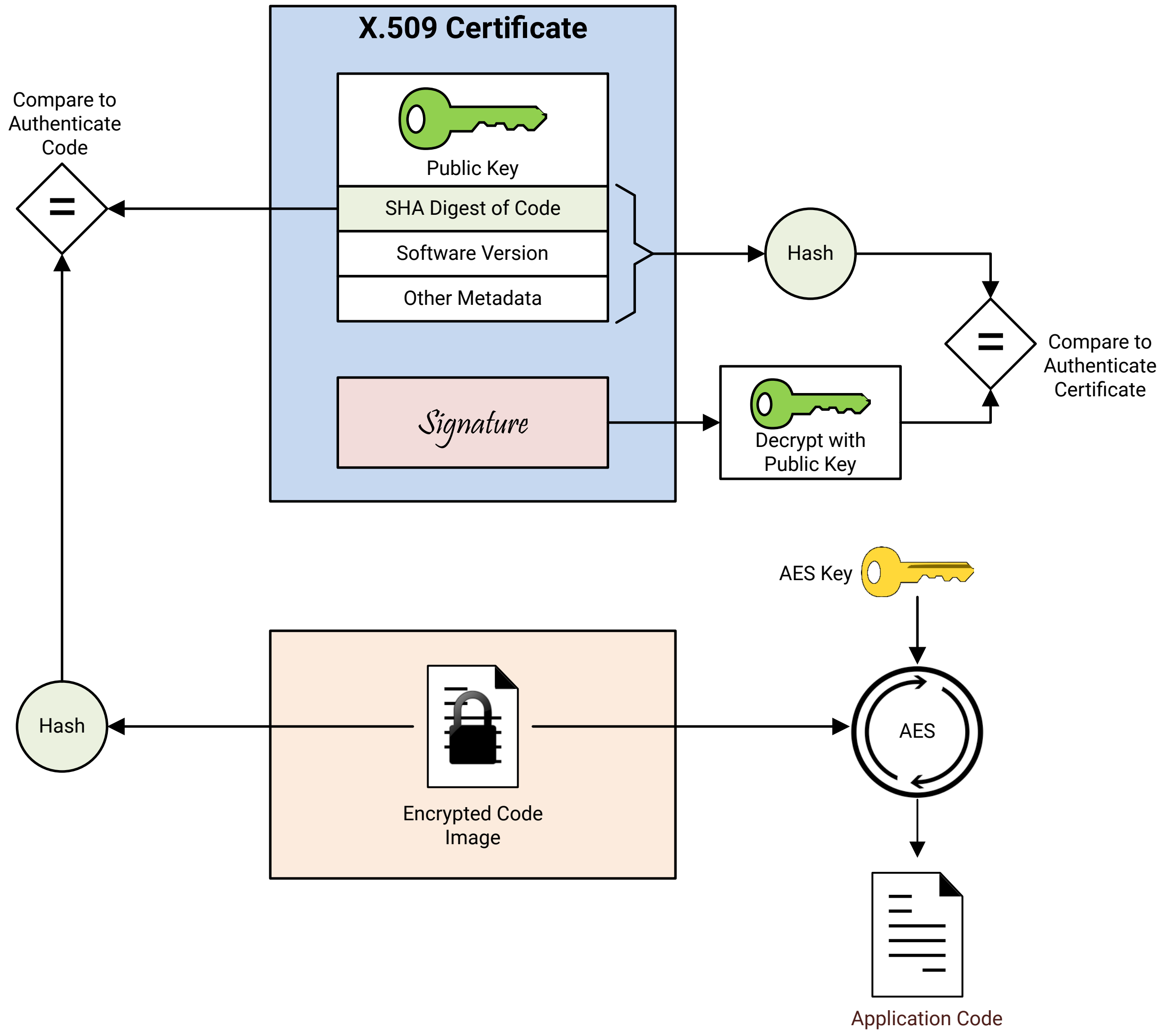

Figure 3-2. Authentication and Decryption Example

Cryptographic authentication schemes can also be used within the application itself to achieve run-time security, verifying the integrity and authenticity of data that is transmitted between devices. Using a secret key, a message authentication code, or MAC, is computed using the cryptographic algorithm, and appended to the message sent to the receiver. The receiver can then use the same key and algorithm to compute the MAC from the received message, and compare it with the one sent by the sender. If the two codes match, the message is verified to be authentic and unchanged.

The two most commonly used cryptographic authentication schemes are CMAC and HMAC. CMAC stands for Cipher-based Message Authentication Code, and is based on a symmetric algorithm such as AES. HMAC stands for Keyed-Hash Message Authentication Code, and uses a hash function such as SHA-256. An example of cryptographic message authentication can be found in modern automotive systems that employ the AUTOSAR Security On-board Communication (SecOC) architecture for data that is transmitted between electronic control units, or ECUs. SecOC uses CMAC to provide end-to-end protection for network messages transmitted over a vehicular network such as CAN, FlexRay or Ethernet. In the SecOC architecture, each message frame includes a security header and trailer, containing the MAC and other security metadata. The MAC can be used by each ECU to verify each message received, and shared keys can be managed and periodically distributed by a central host to preserve freshness. Such schemes can be used to protect against attack methodologies such as CAN injection that are often employed by car thieves.

3.3 Random Number Generators (RNGs)

Random number generation is an important element of many cryptographic services. Random numbers are used to initialize cryptographic sequences, generate keys, create authentication challenges, and more. However, if the random number comes from a source that is predictable or lacks sufficient entropy, this could become a weakness that can be exploited to break the encryption and expose secrets. For computational efficiency, many modern systems feature a Pseudorandom Number Generator (PRNG), sometimes also called a Deterministic Random Bit Generator (DRBG). A pseudorandom number generator generates a deterministic sequence of numbers using a mathematical algorithm, but is dependent on an initial random seed. A true random number generator (TRNG) uses physical sources of randomness such as noise or quantum phenomena to generate bits that are truly random and independent, but is typically much slower than a PRNG. A common practice is to use a TRNG to provide a high-entropy random seed to initialize the PRNG, which is then used to generate the random numbers for the cryptographic application.

4 Establishing a Root of Trust

As previously established, nominally securing the chip boundary is not sufficient to ensure code integrity, due to the myriad ways the application can be compromised or injected with malware. Establishing a root of trust provides two important benefits to bridge this gap:

- Ensures that the device always boots up with trusted code
- If the application is compromised, enables the device to return to trusted code, preventing malware from continuing to run.

The root of trust is a fundamental concept in the security of embedded systems. It refers to a set of hardware, firmware and software components that perform critical security functions, and are trusted by the rest of the system. These critical functions can include the storage of secrets such as encryption keys, authentication and attestation of secondary keys and user code, and cryptographic services. The root of trust forms the first link in a chain of trust that ensures the integrity of application software throughout code execution. The memory element containing root-of-trust functions must be immutable, that is, unchangeable and unmodifiable. ROM, e-fuses, and one-time-programmable or permanently locked Flash are examples of immutable memory.

4.1 Secure Storage of Secrets

An essential component of a secure root of trust is secure storage. Secure storage ensures the confidentiality, integrity, and availability of data assets that are essential to the operation and security of the device. These data assets can include encryption keys, certificates, device configuration settings, and more. Secure storage protects critical assets from unauthorized access or modification, preventing data leakage, tampering or corruption that could compromise the overall security or safety of the application system. Secure storage of encryption keys and credentials is required to establish a secure root of trust. In a secure microcontroller, these assets are typically stored in non-volatile memory such as Flash or e-fuse arrays. Additionally, hardcoded protections that block direct read or write accesses by runtime application software are required, restricting access to immutable firmware such as ROM code or hardware loaders.

4.2 Preserving Key and Code Security

One challenge many users face in deploying embedded systems is how to maintain the security of code, secrets, and intellectual property through the manufacturing process. A process called secure provisioning involves programming secret cryptographic keys into the microcontroller while in an unsecure environment. These keys are subsequently used to authenticate and decrypt incoming application code. In many cases, a third-party manufacturing and programming facility is involved in the provisioning process; and even with signed non-disclosure agreements, it is still possible for a rogue actor to intercept and steal secrets, or potentially install compromised software on the microcontroller. A process must therefore be established that preserves the confidentiality, integrity, and authenticity of secret keys, certificates and code at all stages, starting with the servers from which these assets are sourced, and ending with successful decryption and programming of the assets inside the microcontroller.

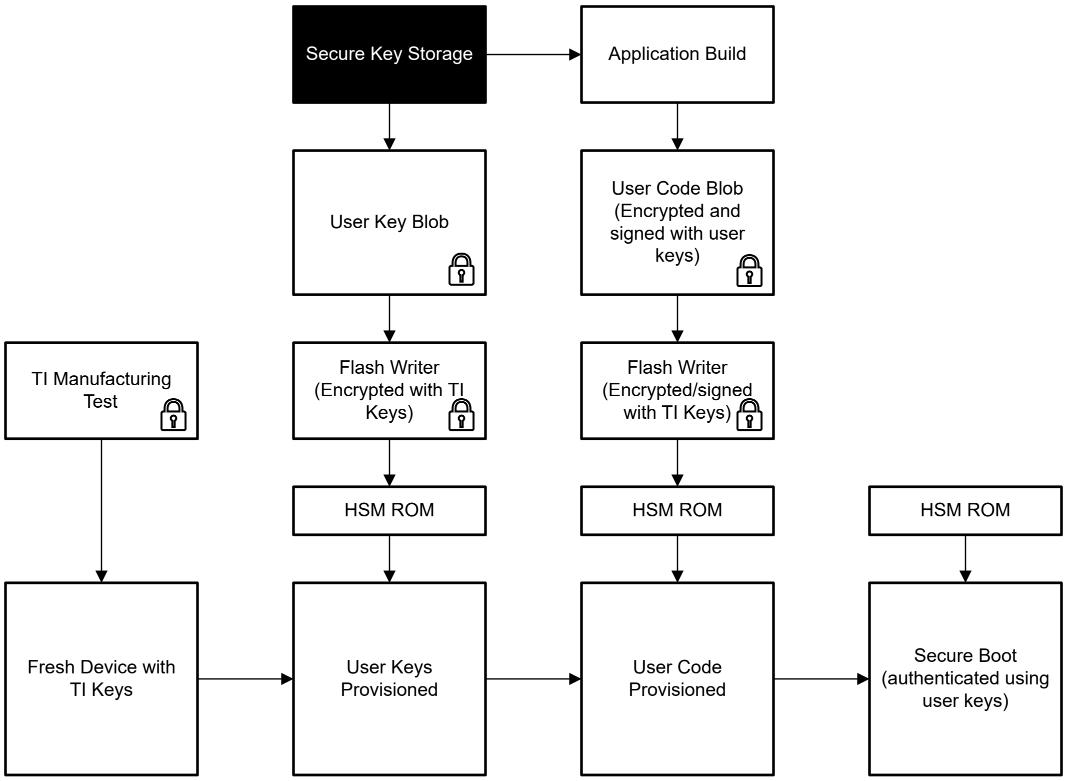
Secure Key Storage
Application Build
User Key Blob
User Code Blob (Encrypted and signed with user keys)
TI Manufacturing Test
Flash Writer (Encrypted with TI Keys)
Flash Writer (Encrypted/signed with TI Keys)
HSM ROM
HSM ROM
HSM ROM
Fresh Device with TI Keys
User Keys Provisioned
User Code Provisioned
Secure Boot (authenticated using user keys)

Figure 4-1. Establishing a Root of Trust with a Secure Provisioning Flow

A secure provisioning process begins with the IT infrastructure that is used to source encryption keys and data. Keys used to perform encryption and decryption must be stored in a secure container and handled within a trusted execution environment, accessible only by key personnel. Within this trusted execution environment, a user key provisioning package is prepared, with user keys encrypted and signed using the chip manufacturer's keys. This package is then securely transmitted to the factory and programmed into the device, preserving the confidentiality of the user keys throughout the process. Once user encryption keys are provisioned, application code can be programmed into the device. This process is similar to the key provisioning process, with the primary difference being that user code is now encrypted and signed using the user keys that were previously programmed into the device's secure storage. For devices with internal Flash memory, there is an additional security benefit: the code remains encrypted and inaccessible until it is programmed into the device. It can then be decrypted and stored in plain text to maximize code execution performance, as confidentiality is maintained using device-level security controls. On the other hand, devices that rely on external Flash chips can use a symmetric key that is unique to each device to encrypt and decrypt code being transmitted between external Flash and internal RAM. For additional security, a unique customer ID can be pre-programmed into the device before shipping to the customer programming facility. This unique ID can be further used to authenticate genuine parts, and help eliminate the risk of unauthorized clones created by rogue third parties.

4.3 Secure Boot

The boot process on an embedded microcontroller or processor represents the first opportunity for an attacker to compromise the security of the system. Securing the boot process is therefore critical to establishing a root of trust for the runtime operation of the system. With the cryptographic keys and certificates that have been securely programmed into the device using the provisioning process, the device hardware and on-chip firmware can verify the integrity of application code, security configuration settings and other data programmed into device Flash memory before commencing execution of the application code. For this process to securely establish a root of trust, all elements used to perform the secure boot operation must be immutable, including the on-chip firmware. Additionally, the chip architecture must be designed such that the secure boot process always happens at every startup or chip reset.

At chip startup, the device first securely loads cryptographic keys and certificates from the non-volatile secure storage medium into protected memory. Then, execution of on-chip boot code commences within an isolated, trusted secure environment that is inaccessible and uninterruptible by any external element, such as a debugger. This boot code decrypts and verifies the integrity of the code certificate, and then uses the certificate to authenticate application code and run-time security settings. The successful verification of code integrity establishes the root-of-trust, and control can be passed to the application to begin executing. The application can in turn perform further diagnostic and integrity checks, or use on-chip cryptographic services to authenticate debug and external communications interfaces using challenge-response schemes.

5 Secure Execution Environment

Establishing a root of trust on the device generally requires the creation of a secure execution environment, sometimes also referred to as a trusted execution environment (TEE). The essential idea behind secure execution environments is to create separate code execution domains, or “worlds”, that have different privileges and access rights to system resources. Secure execution environments can protect code and data in use from unauthorized modification, extraction, or tampering by malicious software or hardware. A hardware security module (HSM) is a typical example of a secure execution environment that is purpose-built for providing cryptographic services, secure storage, root of trust, and authentication for a host microcontroller or processor. TI’s AM26x and F29x microcontrollers include an available built-in HSM with these features, enabling a secure key and code provisioning process, secure boot, debug authentication and cryptographic services for the application.

In many cases, establishing a root of trust alone is not sufficient, as modern applications must deal with potential cybersecurity threats in a connected environment. External communication interfaces, such as a CAN bus or UART port, can potentially be vulnerable to exploits that end up compromising the integrity of the software tasks that operate these interfaces. Given the unpredictable nature of these threats, a complete cybersecurity threat assessment will necessarily include the presumption of compromised code modules, and the risk they pose to the confidentiality, integrity, availability and safety of the whole system. Run-time application security is therefore an important aspect of cybersecurity implementation in embedded systems.

Software-based measures for run-time security are undesirable in real-time control systems, due to the additional processing burden required, which typically means added latency and reduced overall control loop performance. In addition, software-based solutions are not immutable; compromised code could lead to a defeat of the entire run-time security apparatus. A hardware-based solution typically starts with some type of memory protection unit, or MPU. MPUs allow the application developer to define fixed memory regions, and configure access permissions (read, write and execute) for each region depending on the initiator attempting to access it. These initiators can include CPUs, DMAs and debuggers. In the TI F29x family of microcontrollers, memory and peripheral access protections are context-sensitive. The application can be divided into multiple code modules, based on the address range each application code module resides in. Any range of data or peripherals belonging to a code module can then be shared with other code modules, with individual read or write permissions defined. Because these hardware memory protections are specific to the code module performing the access, the need for a software OS layer managing the MPU is eliminated, thus maintaining code and data safety without compromising performance.

In addition to memory access protection, the security of data inside the CPU is also critical to establishing a secure execution environment. An intruder could snoop on the CPU as it executes code, reading out secrets from CPU registers and shared stack memory. One approach to dealing with this issue is software-based: a task scheduler in the operating system layer is responsible for isolating different application tasks or threads from each other. The OS maintains thread context by saving, clearing, and restoring registers when switching to a new task. The main drawback to this approach is that interrupt service routines, which are key to real-time processing applications, cannot be directly isolated using a task scheduler. Hardware features that help isolate application contexts within the CPU are therefore key to achieving run-time security in an embedded real-time system. Often this involves hardware handling of stack pointers, with a division between “secure” and “non-secure” worlds, or in CPU architectures such as C29x, multiple fully isolated stacks. Typically, this type of CPU instruction set includes gate instructions that are required to cross stack boundaries. The gate instruction can trigger processes such as register zeroing, and an associated access protection scheme that specifies what memories can be written or read. In architectures such as Arm® TrustZone®-M, a trampoline or veneer function is generally required to handle these protection changes during the transition from one context to another.

Texas Instruments C29x, on the other hand, mostly handles context switches in hardware, applying new protections in real time to maximize control performance.

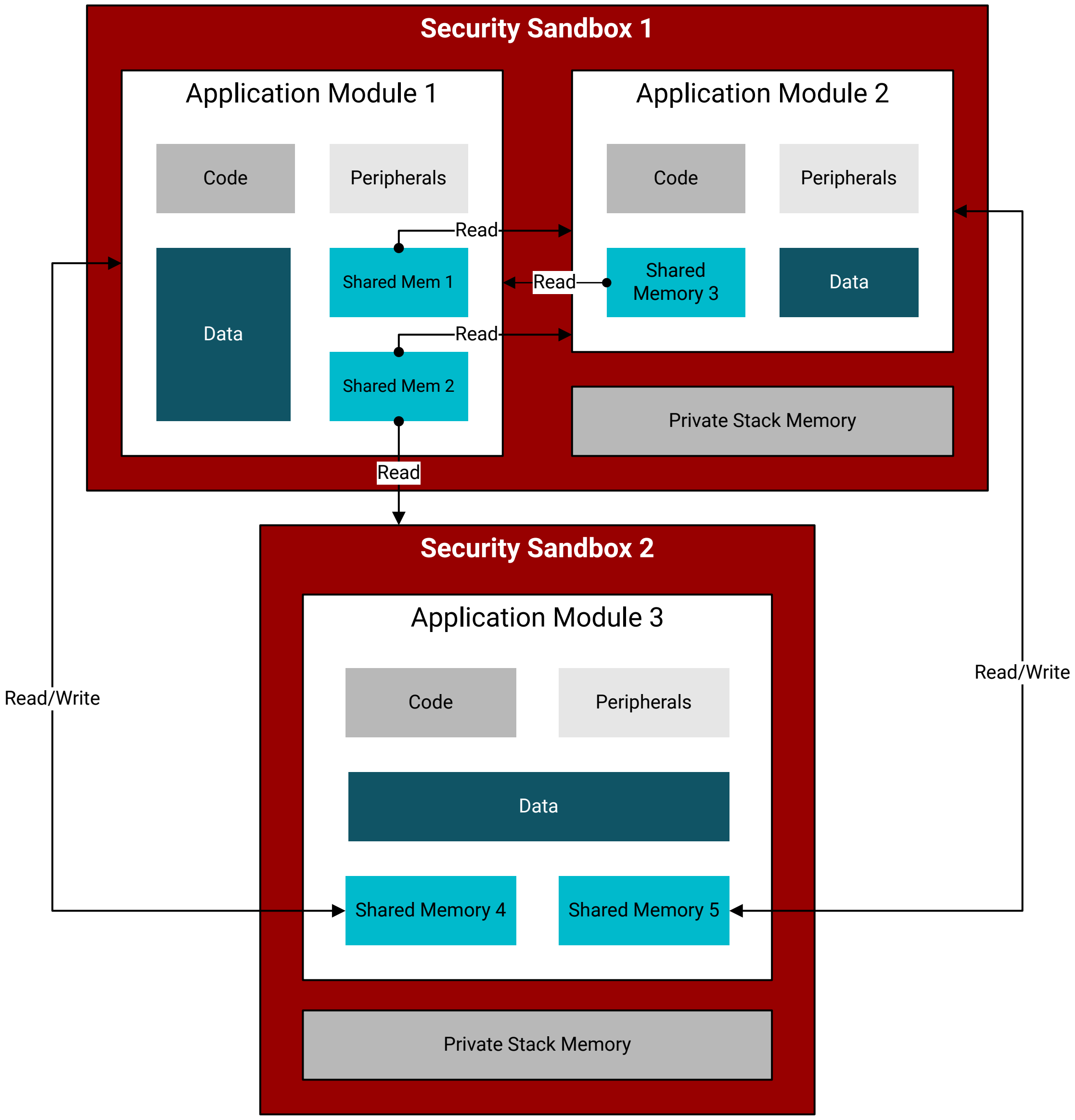

Figure 5-1. Application Run-Time Security Partitioning Example

With hardware-enabled protections for code and data in the CPU, the embedded system designer can partition the application into security sandboxes and application task modules, each with its own private code, data, and peripherals. Each of these task modules can also optionally share memory regions with other tasks. An example security partitioning scheme is shown in Figure 5-1. Often a real-time operating system layer is required to implement such schemes, but with isolated stacks and context-aware memory access protection in hardware, system designers can achieve complete isolation while avoiding the extra latency that comes with a software OS layer.

6 Security Countermeasures

An important aspect of implementing a cybersecurity strategy for embedded systems is identifying weaknesses in the system, enumerating potential attack types and scenarios, and implementing cybersecurity controls to mitigate those attacks. Embedded control systems tend to be more accessible to attack techniques that require local or physical access to the device. These include connections to the debug port, fault injection attacks such as power or clock glitching, and other side-channel attacks. A sound embedded security implementation identifies these various attack scenarios and implements countermeasures to mitigate them.

Fault injection attacks in principle attempt to redirect CPU execution by introducing a temporary anomaly in the device's power supply voltage or system clock signal. For example, each device data sheet includes specified operating voltage ranges for power supply rails such as core, I/O, and analog, as well as operating temperature ranges. This means that the device is designed to operate correctly and meet all timing requirements within these voltage and temperature ranges; operating the device outside the specified bounds could result in unspecified behavior. In practice, violating these specifications typically results in timing-related faults: undervoltage conditions cause setup time violations, and overvoltage conditions lead to hold time violations. Alternatively, an accessible input clock signal, such as a crystal oscillator input pin, could be used to directly inject timing faults at key points in time. A well-timed fault can result in skipped instructions or redirected execution at critical security decision points in firmware, leading to unauthorized access or exposure of embedded secrets. Tools for performing the required timing analysis to execute successful fault injection attacks are increasingly easy to obtain. Countermeasures against these types of attacks can include internal voltage and clock frequency monitoring circuits.

For protection against fault injection attacks, the ability to instantaneously detect single- or double-bit faults is a valuable tool. Error Correction Code logic (ECC) is an example of such a protection mechanism, providing the ability to automatically correct single-bit faults and detect double-bit faults in memory or on a bus. The mechanism operates using a pre-computed code that is typically written into memory alongside each data word. ECC is commonly used within functional safety contexts, but is also valuable as a security countermeasure, enabling the system to either reject injected faults, or respond by halting execution and sending out an error signal. In TI's AM26x microcontrollers, all ECC protection is available for all on-chip memories and caches, and also for various peripheral subsystems and interconnects. F29x microcontrollers feature ECC protection built directly into the C29 CPU and device interconnect, providing robust end-to-end protection against code and data faults across all memories and peripherals during application runtime.

7 Debug Security

A common way for attackers to compromise an embedded control system is to connect to the microcontroller's debug port using JTAG, SWD or some other standard protocol. If the debug port is unsecured, the attacker could easily redirect execution, extract secrets, or explore other ways to compromise the system. A microcontroller used in a secure application should offer password-based locking of the debug port as a base requirement. For added security, system designers who want to avoid the use of passwords, which are potentially exposable as shared secrets, an alternate authentication mechanism can be implemented, such as challenge-response or certificate-based authentication using public key cryptography.

In certain cases, it is desirable to demarcate the application into multiple domains, potentially outsourcing the development of one or more of those domains to a third party or otherwise untrusted software development environment. In this scenario, having a single authentication credential that is shared across all development parties is less than ideal from a security standpoint. To address this weakness, the device architecture can partition resource groups on the chip with independent debug enable signals. Multiple authentication credentials can then be created for a given device, with varying levels of access to device resource groups. For instance, one debug certificate could grant access to the entire chip, while another certificate blocks access to memory regions containing sensitive data.

8 Conclusion

As automotive and industrial real-time control systems become more complex and interconnected, cybersecurity threats continue to increase in scope and ingenuity. The modern threat landscape therefore drives a demand for more stringent cybersecurity features in real-time microcontroller products, without compromising loop processing performance. Texas Instruments C2000™ F29x and Sitara™ AM26x microcontrollers meet this need, enabling embedded designers to build highly secure real-time control systems for modern applications. These products include an embedded Hardware Security Manager (HSM) that provides firmware protection, secure debug authentication, secure storage, and modern cryptographic functions. Secure provisioning of user encryption keys and code, secure boot, and secure firmware update features can be used to establish a root of trust, and maintain the confidentiality and integrity of user code and assets. Additionally, the unique architecture of the TI C29x CPU enables dynamic context-aware security for applications during run-time, extending the concept of a secure execution environment beyond the boundaries of the HSM. Using this run-time security model, externally facing application functions such as communication stacks can be isolated from critical system functions, mitigating the risk to the overall safety and security of the system posed by outside threats. With this complete platform security solution, users can develop high-performance real-time control systems that meet the highest cybersecurity standards.

Visit ti.com/mcu to learn about embedded microcontrollers at Texas Instruments, including the C2000 and Sitara product families. For more information about product cybersecurity, visit ti.com/security.