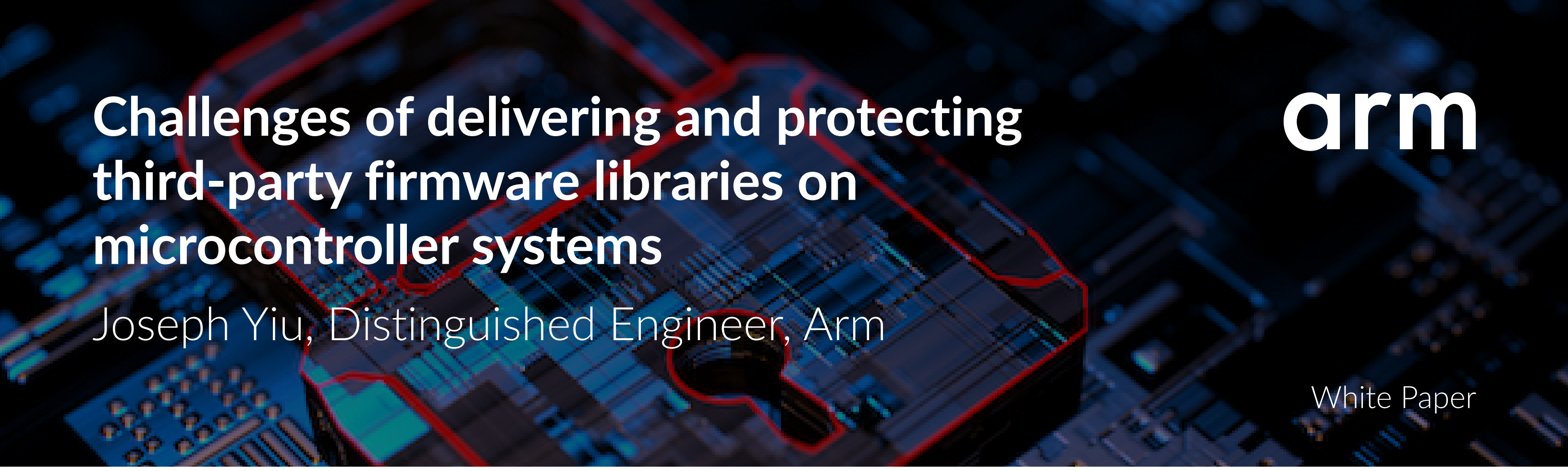

# Challenges of delivering and protecting third-party firmware libraries on microcontroller systems

Joseph Yiu, Distinguished Engineer, Arm

White Paper

Today's microcontroller products designed for the IoT may come preloaded with a range of firmware from different vendors. While TrustZone for Armv8-M helps protect devices from hackers and untrusted parties, there are cases where a fragmented supply chain can cause a higher risk of leaking trade secrets for firmware vendors. In these instances, devices need additional IP protection capabilities.

These new capabilities change the debug features, memory protection architecture, and system-level security features of the processor. This presentation explains the background of these requirements, and how new features in the Armv8.1-M architecture can help chip vendors address these challenges. For example, demonstrating how the Unprivileged Debug Extension (UDE) feature in Armv8.1-M restricts debug visibility to a specific software partition. To use the UDE, debug tools and software must be adapted and devices require debug authentication support.

## 1. Background

In the era of IoT, many mainstream microcontrollers carry a range of preloaded firmware to deliver various software solutions including security management, communication stacks and hardware drivers. As the number of software components in firmware increases, it might no longer be feasible for a microcontroller vendor to create all these solutions themselves. As a result, some microcontroller vendors work with one or multiple third-party middleware vendors to provide the software solutions that software developers need.

In addition to the integration of traditional middleware components, IoT applications introduce the needs for the integration and management of a diverse set of cloud services clients. Due to limited memory spaces in microcontroller devices, it is not possible to preload all cloud client software that software developers might need into the microcontrollers. As a result, there is a need to allow some of these cloud clients to be programmed into the device later on, based on the needs of the applications. For most connected devices it must be possible to update software (including middleware and cloud clients) to be able to fix bugs and possibly add new features.

All these requirements lead to the exposure of software components to software developers and other parties in various ways:

- Application developers need to be able to call Application Programming Interface (APIs) inside the on-chip middleware and cloud clients
- Application developers might be able to load their choice of middleware or cloud clients into the chips using chip programming tools (likely to be chip vendor-specific)
- After an IoT product is deployed, it might be able to update some of these third-party software components over-the-air (using a direct/indirect internet connection)

Meanwhile, for many providers of the middleware and cloud clients, their high value intellectual property (IP) must be protected. For example, some of the software components might contain proprietary algorithms of high value. In addition, some of the providers of the software components might be in direct competition with each other, and exposure of underlying operations inside these software components might lead to a leakage of trade secrets. As a result, there is a strong need to protect the confidentiality of the software components.

To make the situation slightly more complicated, it might also be needed to protect the distribution of the software components even if they are preloaded on the chip during chip production. In some cases, a part of the chip manufacturing process could be outsourced and software vendors might not trust the third-party companies that handle parts of the chip manufacturing, as these third-party companies might also deal with other companies that are competitors to the provider of the software components.

Overall, software vendors might see various security risks regarding intellectual property (IP) protection:

- Application developers could try to reverse engineer their software implementation. This also includes competitors or hackers that gain access to the devices containing the preloaded software

- If the software is already loaded on to the chip when other middleware vendors/cloud client providers develop their solution, there is also a risk that other middleware vendors/cloud client providers could try to reverse engineer the preloaded software image
- During part of the chip manufacturing process, the program image(s) on-chip could be readout
- During the delivery of software updates, competitors or hackers can access the updated image(s) and try to reverse engineer the code
- A software component from one company might attempt to extract secret information from another software component from a competitor

## 2. The Basic Protection

One of the most frequently mentioned solutions for these challenges is encryption of the compiled image before delivery. To prevent reverse engineering of the software image, the encrypted image is programmed to non-volatile memory directly without being decrypted. To allow execution of the encrypted software image, the memory interface needs to support on-the-fly decryption.

Using on-the-fly decryption does introduce extra memory latency. Usually, AES (Advanced Encryption Standard) decryption operations have to be done in blocks of at least 128-bits. However, in many existing microcontrollers, the use of flash memories with 128-bit wide I/O is already commonplace due to speed limitations of embedded flash memories (using a wide data bus allows higher data bandwidth). To hide the latency of embedded flash memories, many microcontrollers also include program caches. As a result, a small increase in memory access latency due to the on-the-fly decryption would only have an impact when there is a cache miss, which is not frequent.

To avoid information leakage between different middleware vendors, on-the-fly decryption can be arranged on a memory partition basis so that each partition can have its own decryption key. To allow the security state of each non-volatile memory (NVM) partition to be managed independently, the security management service of the chip (secure storage and lifecycle state management) must also be adopted.

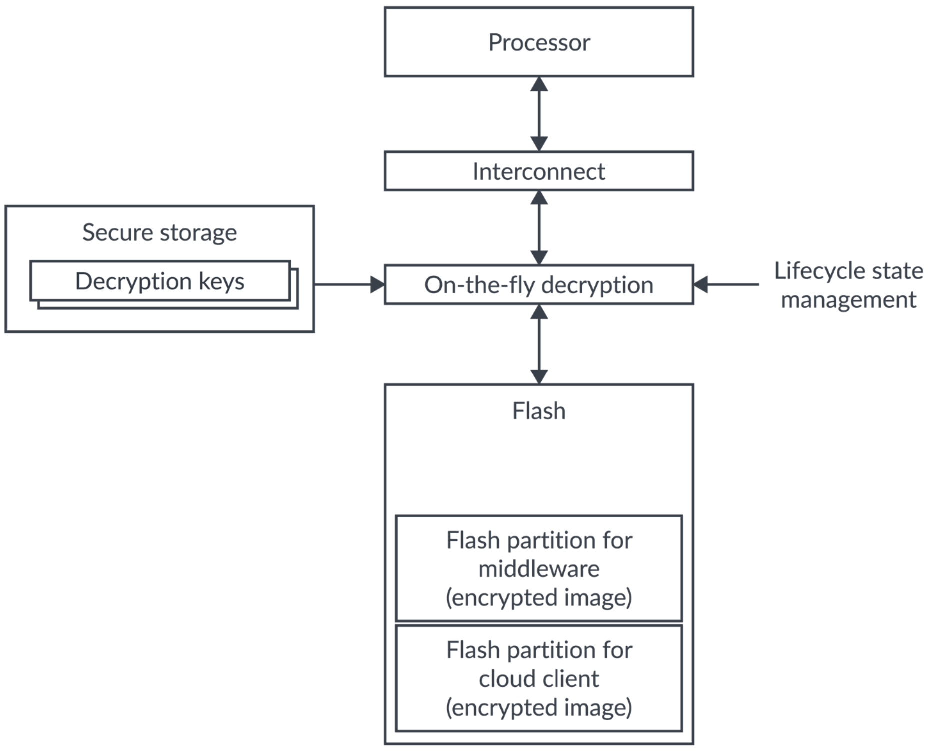
Processor
Interconnect
Secure storage
Decryption keys
On-the-fly decryption
Lifecycle state management
Flash
Flash partition for middleware (encrypted image)
Flash partition for cloud client (encrypted image)

Figure 1: Concept of using on-the-fly decryption engine to support encrypted firmware components

Just having on-the-fly decryption support by itself is not enough. For example, once the decryption key has been installed in the Secure storage, reading the encrypted image will reveal the raw image data. Without additional protection, an application developer could read the protected firmware.

Arm TrustZone technology, already available in a selection of Arm Cortex-M processors, can address this challenge. Within a TrustZone environment, the processor can operate in a normal (Non-secure) state which executes programs in Non-secure memories, and a protected (Secure) state which executes programs in Secure memories. By putting the encrypted images in Secure memories, and with debugging authentication support, application developers creating applications running in the normal (Non-secure) world can call Secure APIs in the encrypted images, but cannot:

- Read the protected software image (either by debugging connection or by running code that readout memories), or
- Branch into the middle of a Secure API

Arm TrustZone technology also protects the installation of decryption keys. With preloaded Secure firmware, a Secure connection between the microcontroller device and authentication server can be established via attestation. Entity Attestation Tokens (EAT) enable the decryption key to be transferred securely (for example, using an asymmetric cryptography method) after a device registration process. Attestation is one of the 10 security goals of PSA Certified. PSA Certified provides resources to make attestation possible, the Trusted Firmware-M project provides open source reference firmware for attestation. Without TrustZone, it is possible for the details of attestation and decryption key transfers to be observed by an application developer, who can then copy the decryption key.

Here we assumed that brute force attack against the encrypted program image is sufficiently expensive for the majority of hackers and this area is not considered in this paper.

We also assumed that using a unique crypto key for encrypted program image per device is too expensive to implement, in the sense that the servers hosting the firmware update images have to encrypt the firmware images on the fly (unique to each device), and the server needs to be trusted to handle such operations.

## 3. A Requirement for Additional Isolation within the Secure World

Protecting the Secure software from unauthorized application developers is a step forward, but not enough in the new scenarios where silicon vendors and Secure middleware/cloud client developers do not fully trust each other. For example, if the microcontroller contains cloud clients for cloud service providers A and B, where the two companies are in direct competition, there is a very strong need to make sure proper isolation is in place between these two cloud client software components.

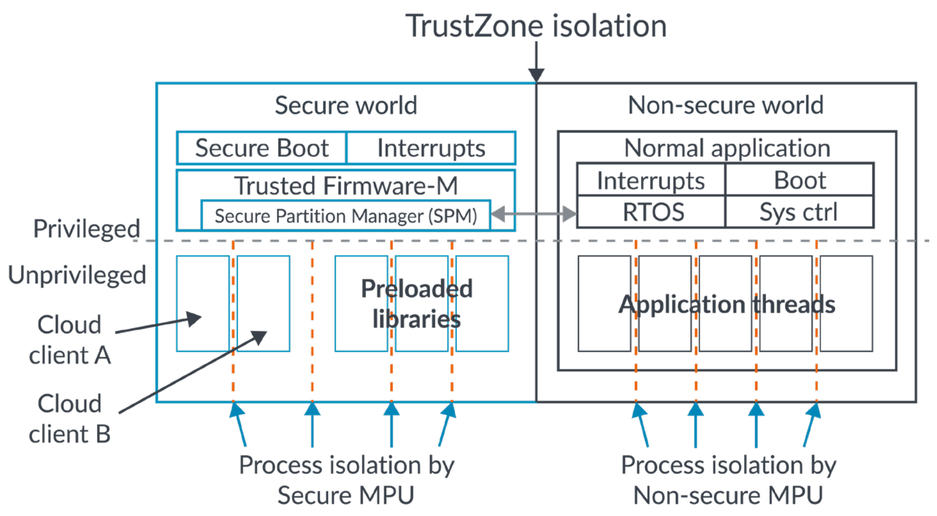

Figure 2: Various software components are isolated

TF-M provides the firmware reference code for isolation both between Secure and Non-secure processing environments and between different client software components. This ensures the product aligns with PSA Certified Level 1 security requirements. Using the Secure Memory Protection Unit (MPU), the Secure Partition Manager (SPM) inside TF-M can ensure that Secure unprivileged software components can only access the memory ranges that they are allowed to access. Similar to how an RTOS manages the Non-secure MPU, the SPM reconfigures the Secure MPU during context switching so that different application threads can call different Secure APIs in the Secure libraries and clients.

If a Non-secure thread which has previously interacted with one Secure library starts calling APIs from another Secure library, the SPM reconfigures the Secure MPU. This overhead happens only when the second library is called for the first time, and subsequent calls to the same library do not require additional MPU reconfiguration, enabling lower overhead. This is a significant advantage over Virtualization/Hypervisor where each function call between virtual machines requires considerable overhead in context switching.

Meanwhile, an RTOS running in the normal (Non-secure) world can use the Non-secure MPU for process isolation between application threads running in the RTOS. The configuration settings for the Secure and Non-secure MPUs are independent. If preferred, the application running in the Non-secure world could use an RTOS that does not have MPU support (Non-secure MPU not used in that case), or even running a bare metal application. This does not affect the isolation between Secure partitions, which uses the Secure MPU.

This solves the challenge of isolating and protecting Secure software components (including third-party firmware libraries) during software execution. However, to protect software assets of software components during software development, we need another mechanism.

## 4. Requirements for Debug Isolation During Software Development

Today, most of Arm microcontroller debug tools support halt mode debugging. While monitor mode debugging support is available in Armv7-M and Armv8-M Mainline processors, halt mode debugging is much more popular as there is no need for additional memories for debug agents running on the processors. In addition, monitor mode debugging presents limitations (since some exception handlers cannot be debugged using debug agent) which make it unsuitable for some applications.

With TrustZone technology introduced for Cortex-M processors in the Armv8-M architecture, debug authentication support is included so that we can control debug access permission for Secure and Non-secure worlds using a set of debug authentication control signals:

Table 1: Debug authentication signals on Armv8-M processors

| Signal | Name | Function |
|---|---|---|
| DBGEN | Invasive Debug Enable | When set to HIGH, enables invasive debug activities such as halting (breakpoint and watchpoint) and single-stepping for Non-secure software execution |
| NIDEN | Non-invasive Debug Enable | When set to HIGH, enables Non-invasive debug activities such as instruction and data trace, and profiling for Non-secure software execution |
| SPIDEN | Secure Privilege Invasive Debug Enable | When set to HIGH, enables invasive debug activities such as halting (breakpoint and watchpoint) and single-stepping for Secure software execution |
| SPNIDEN | Secure Privilege Non-invasive Debug Enable | When set to HIGH, enables Non-invasive debug activities such as instruction and data trace, and profiling for Secure software execution |

If **DBGEN** or **NIDEN** is HIGH, then a debugger can access Non-secure memories through a debug connection. Similarly, if **SPIDEN** or **SPNIDEN** is HIGH, then a debugger can access Secure memories. This adds protection to prevent normal software developers (who develop Non-secure software) from being able to see what is inside the Secure firmware and reverse engineer it. However, if a software developer is granted access to Secure debug (which is needed for the creation of cloud client A and B in Figure 2), this person (let's say this person is from cloud service provider B) would also be able to access Secure memories used by other Secure software. This might be a problem because:

- A secure software developer could look into Secure memories used by other software components, including memories used by TF-M and Secure boot. This could result in leakage of critical secret data like secret keys. To avoid problems, the development platform provided to company B for software development might be prepared specifically so that it does not contain preloaded critical secret data
- Software developers from cloud company B could look into the binary image of company A if the image was already loaded and the decryption key for cloud client A is already installed. Normally the decryption keys are expected to be installed at product activation, so this is less likely to be a real problem

Although these issues can be worked around, it would be nice to have a better solution. As a result, Armv8.1-M introduced a new feature called Unprivileged Debug Extension (UDE).

When this new debug mode is used, privileged software can control which unprivileged software is allowed to be debugged, without exposing data used by privileged software or other unprivileged software.

Privileged software controls UDE operations which are available for both Secure world and Non-secure world. It does not affect the existing halt mode debug when normal debug is permitted and UDE is turned off, so debug support in current toolchains can still work with Armv8.1-M processors. When a debugger is attached and when privileged software selected to use UDE debug:

- The processor can halt only when the processor is in the unprivileged state, and when UDE halting is allowed (controlled by privileged software and updated at each context switch)
- The debugger can access memories (excluding debug components, which can be accessed throughout the debug session) only when the processor is halted and access is monitored by the MPU

Because debug access is checked against MPU settings and can only be carried when the processor is halted at the software component that can be debugged, the debugger has exactly the same access permission as the unprivileged software component being debugged, outlined by the privileged software (either SPM on Secure side, or RTOS on Non-secure side). In this way, the software developer creating cloud client B is not able to access memories allocated for cloud client A or privileged software, satisfying the security requirements.

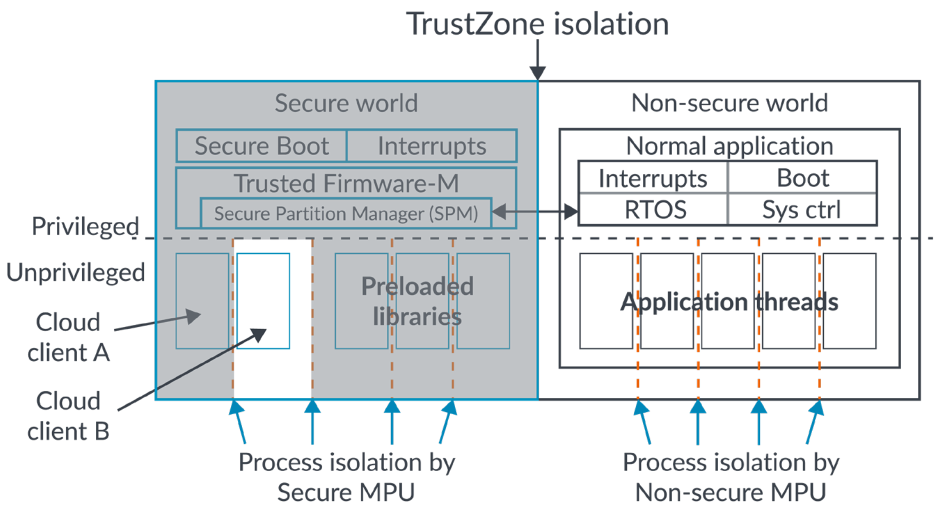

Figure 3: UDE allows Secure privileged software to control debug visibility to a specific unprivileged software partition (cloud client B)

As mentioned before, UDE is supported for both Secure and Non-secure worlds. While the example above shows the use of UDE in Secure world, it is possible for an embedded OS to implement UDE to restrict debug visibility to a specific unprivileged application. In this case, instead of using halt mode debugging, it is also possible to use debug monitor support so that the OS and other application threads that are not being debugged can continue to run without being affected by the debug operations.

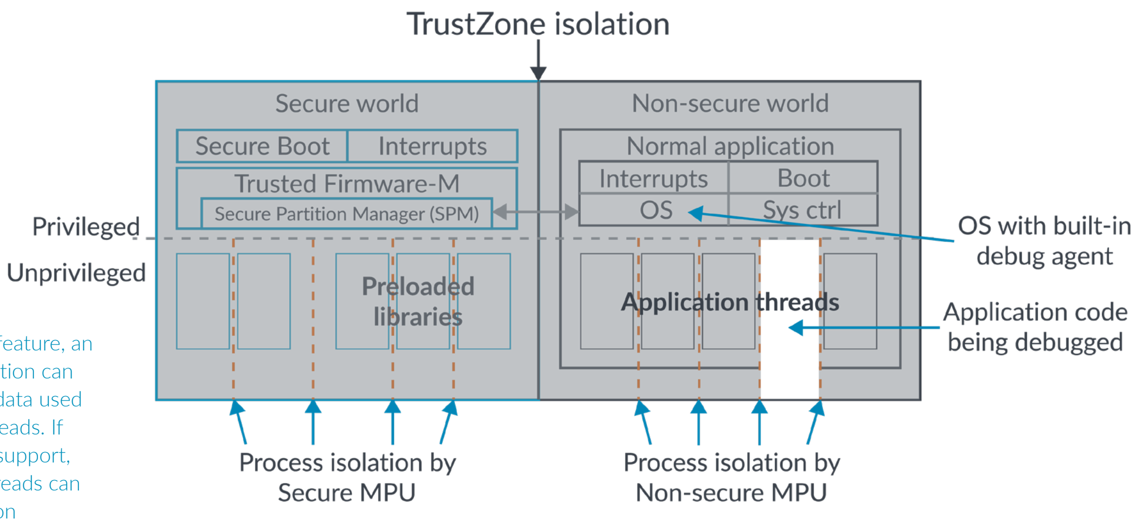

Figure 4: Debug With the UDE feature, an unprivileged Non-secure application can be debugged without exposing data used by privileged code and other threads. If using UDE with debug monitor support, the OS and other application threads can run throughout the debug session

## 5. Putting Multiple Pieces of the Puzzle Together

### A. Overview

So far, we have covered several pieces of the solution:

- On-the-fly decryption of encrypted software image enables a processor to execute an encrypted image
- TrustZone security enables installation of a decryption key to be handled securely during the product activation stage
- TrustZone security enables APIs in protected firmware to be called by normal application code, but at the same time protected from being reverse-engineered by software developers with debug access to the Non-secure world
- Secure MPU enables process isolation between various Secure software components
- Unprivileged Debug Extension (UDE) enables unprivileged Secure software components to be developed and debugged without exposing data for Secure privileged software or other Secure unprivileged components

To get the whole system to work, we need additional support:

- Microcontroller/SoC designs – In addition to on-thefly decryption, chips need to have additional functions:
  - Dedicated memory spaces (flash memory and SRAM) for third party software partitions
  - Ability to update the firmware of these memory partitions individually
  - Debug Authentication application on a debug host to handle debug authentication (e.g. communicate with debug authentication services running on the cloud)
  - An on-chip firmware software component (Debug Authentication Manager) to receive debug authentication information from the Debug Authentication application, and interface with the SPM (Secure Partition Manager) in TF-M (Trusted Firmware-M) to indicate if a Secure partition is allowed to be debugged
    - Potentially the Debug Authentication Manager needs to provide software interface for Non-secure world if UDE support on Non-secure side is needed.
  - A communication interface to allow Debug Authentication application on debug host to communicate with Debug Authentication Manager. The SDC-600 – Secure Debug Channel, is one possibility but the communication could also be handled by other physical interfaces like a UART
- Security firmware (e.g. Trusted Firmware-M)
  - Support of Unprivileged Debug Extension (UDE) in Security firmware is needed for isolating debug accesses to different software components in the Secure world so that each time a context switch of Secure unprivileged library takes place, the debug access permission for Secure unprivileged can be updated accordingly
  - A software interface between secure library management (e.g. Secure Partition Manager in Trusted Firmware-M) and the device vendor-specific Debug Authentication Manager is needed. This allows library management to access information about what software component should be allowed to be debugged
  - Secure privileged software needs to include error reporting code so that during the development of Secure unprivileged libraries, if the library codes caused a failure and resulted in fault exceptions, the error details (e.g. values of fault status registers) can be examined by the software developers
- RTOS running in Non-secure world
  - If the UDE support is needed in the Non-secure world, then an update to RTOS is needed so that each time a context switch of RTOS threads (unprivileged) takes place, the debug access permission for Non-secure unprivileged can be updated
    - The RTOS can communicate with the Debug Authentication Manager in a Secure world to determine the debug permission configuration
  - Similar to Security firmware, if UDE is introduced then fault reporting codes are needed to enable developers of unprivileged application codes to debug software failures

The microcontroller/SoC design should also have a range of hardware security features such as Secure storage for crypto key storage, crypto accelerators such as Arm CryptoCell, CryptoIsland and True Random Number Generator (TRNG). These are already commonplace in many IoT devices. While UDE can work without strong hardware-based security protection, these features are desirable from a security point of view.

### B. Initial installation of an encrypted image

Back in the early part of this paper, we covered the need for on-the-fly decryption support in program memory. This enables software vendors to deliver their middleware/library in encrypted forms. To protect the software asset, the encrypted software is programmed in its encrypted form. The encrypted software images could be preloaded during chip manufacturing or can be programmed to the chip at a later stage. The decryption key is installed into the device only when the product is being activated and deployed, or when an authorised Non-secure application developer needs access to the Secure library to install a key.

To support features like product activation, we need lifecycle state management. A processor-based device can have a number of lifecycle states such as (but not limited to):

- Manufactured – the chip has just been produced
- Software development – initial boot loader and preloaded firmware are loaded, and allow application software to be downloaded and tested
- Software finalised – the product might be in storage/transit
- Activated – product is activated by an end user and is actively used
- Return to manufacturer – might need to allow diagnosis of issues reported by customers
- Disposal – confidential data needs to be erased

To support lifecycle state management, the chip needs to have Non-volatile memory to hold the lifecycle state, and such storage needs to be able to prevent rollback of product state. Support for lifecycle state management is essential for controlling debug authentication. In addition, software operations such as crypto key management can also be made aware of the product lifecycle state.

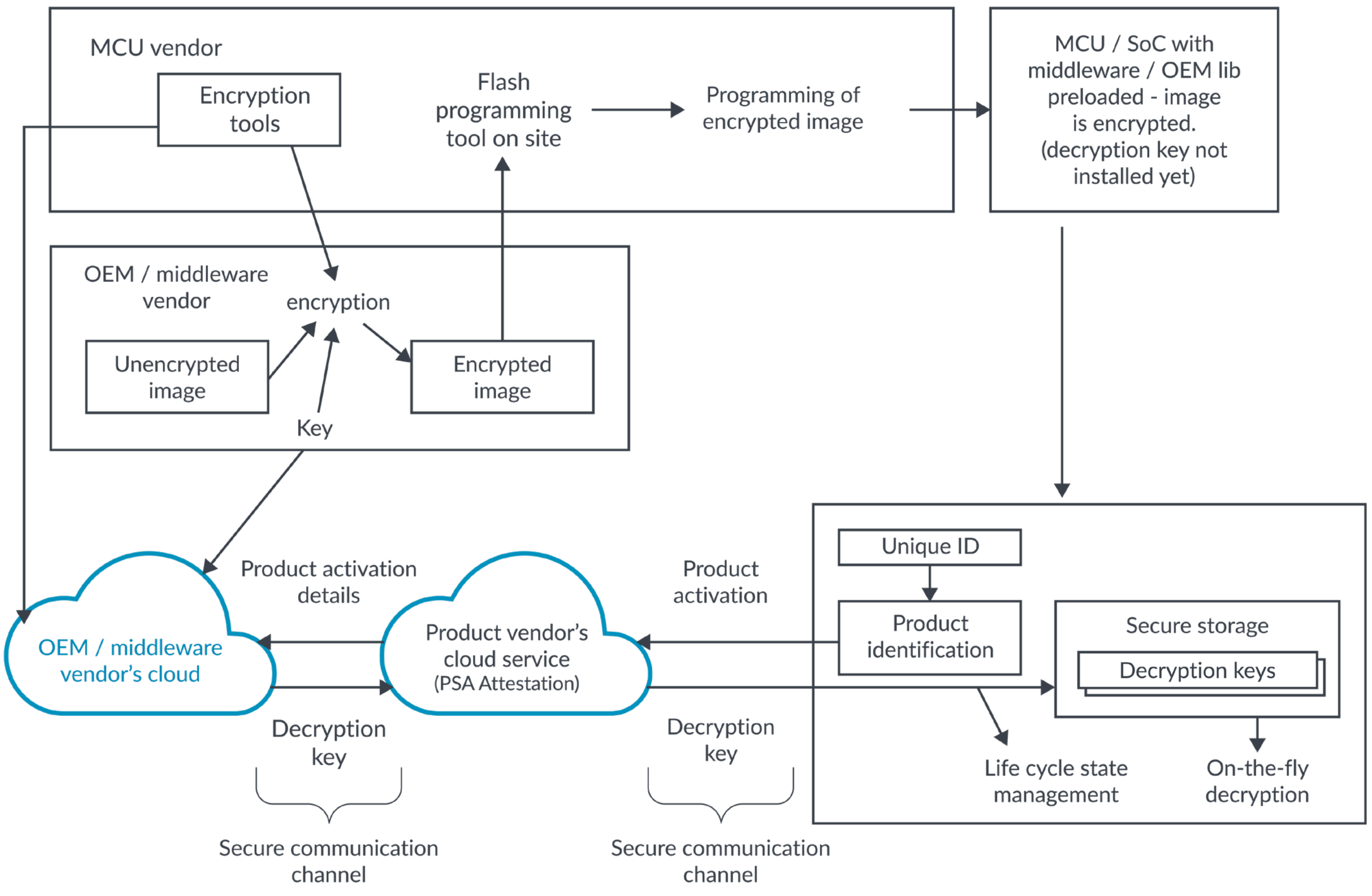
MCU vendor
Encryption tools
Flash programming tool on site
Programming of encrypted image
MCU / SoC with middleware / OEM lib preloaded - image is encrypted. (decryption key not installed yet)
OEM / middleware vendor
encryption
Unencrypted image
Encrypted image
Key
Product activation details
OEM / middleware vendor's cloud
Product vendor's cloud service (PSA Attestation)
Product activation
Unique ID
Product identification
Secure storage
Decryption keys
Decryption key
Decryption key
Life cycle state management
On-the-fly decryption
Secure communication channel
Secure communication channel

Figure 5: Creation of encrypted software library and installation of decryption key at product activation

During product activation, the processor can invoke authentication in middleware vendor's or OEM servers (such as cloud services) to provide the decryption key. The decryption key would need to be transferred securely to the device (such secure connection should be established for product activation anyway, and here we assume that the device has another preload key to help facilitate this need). Once the key is installed, the secure library is then available. However, the secure library might include its own authentication process as the service/feature provided in the library might require a separated service subscription arrangement.

Lifecycle state management on the chip should ensure that

- Unless authorised by the middleware/OEM library vendors, the decryption key(s) cannot be installed until the product gets to product activation stage – this prevents third-party software developers that have Secure debug access from installing the decryption key and gaining read access to the library. Attestation support in PSA (Platform Security Architecture) can be used to ensure that the product's lifecycle state allows the decryption key to be installed

- The usages of program memory partitions are tracked so that once a partition is programmed with a software component from vendor A, vendor B cannot overwrite it
- The decryption key(s) is erased if the product is deactivated (end-of-life) and needs to be returned to the factory. This prevents the key being read out by a person in the device's manufacturer

This of course assumes that the lifecycle state management of the device is implemented with strong protection around it.

## C. Development of Non-secure software

For Non-secure software developers that develop applications for end users, they will also need the key so that they can call the Secure APIs in those libraries. As a result, these software developers will need authorisation from the vendors of Secure libraries to install the decryption key before normal product activations.

The installation of the decryption key does not allow the Non-secure software developers to reverse engineer Secure code, as this is protected by TrustZone.

## D. Over-the-air firmware update

Over the product's lifecycle, bugs in products could be found and additionally there might be a need to enhance the product with new features. As a result, there is a need to allow the protected firmware to be updated. To protect the software libraries, the new program image needs to be delivered in encrypted form if software asset protection is required. To enable wider distribution of the update images, the encrypted software image could be hosted by third-party servers.

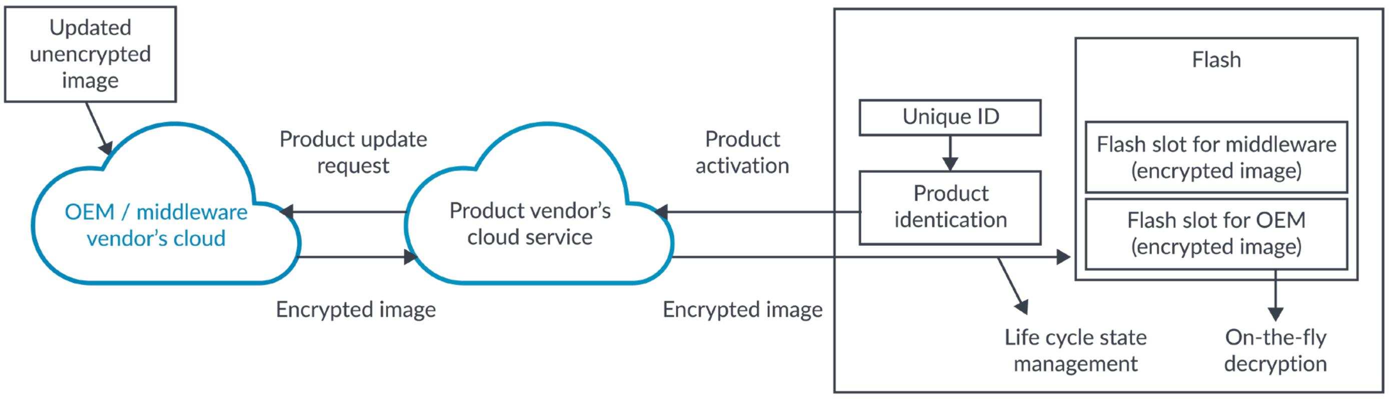
Updated unencrypted image
OEM / middleware vendor's cloud
Product update request
Encrypted image
Product vendor's cloud service
Product activation
Encrypted image
Unique ID
Product identication
Flash
Flash slot for middleware (encrypted image)
Flash slot for OEM (encrypted image)
Life cycle state management
On-the-fly decryption

Figure 6: Over-the-air firmware update is possible for encrypted software libraries

It is possible to use new decryption key for update image, but that means the product would need to get the new key (via a secure connect as in device activation, and the key needs to match the specific firmware version) for each firmware update. If this is not feasible, the program encrypted could be encrypted with the previously installed key, but it means if the key has been leak/cracked, future firmware is also leaked.

## 6. Debug Protection

Before the middleware/library is released and encrypted, it needs to be developed and the debug protection can be handled by UDE. To enable the UDE to be deployed (assumed it operates in a Secure world), we need various software components:

- Debug authentication manager (silicon vendorspecific) need to be able to communicate with the debug authentication application running on the debug host to receive debug authentication information
- The library management component (e.g. Secure partition manager in Trusted Firmware-M) need to be available and can communicate with the debug authentication manager software so that UDE operations can be based on debug authentication configuration
- The debug authentication application on the debug host could be connected to authentication services running on the cloud, or can be a local authentication service

One possible system architecture is illustrated below:

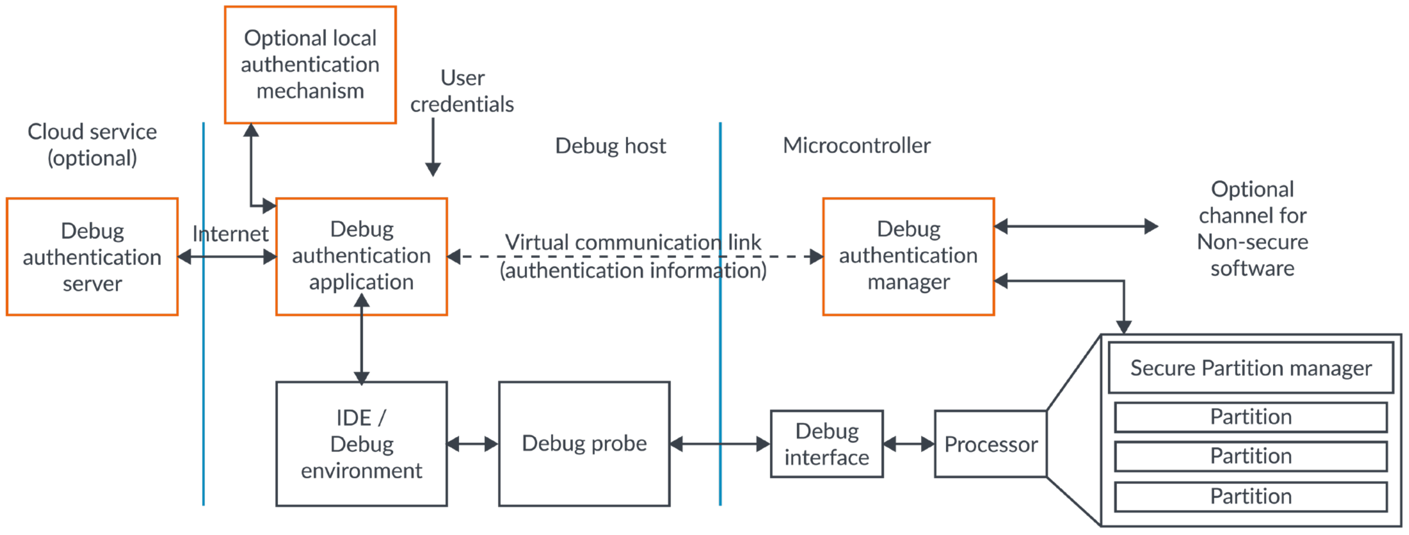
Optional local authentication mechanism
User credentials
Cloud service (optional)
Debug host
Microcontroller
Debug authentication server
Internet
Debug authentication application
Virtual communication link (authentication information)
Debug authentication manager
Optional channel for Non-secure software
Secure Partition manager
Partition
Partition
Partition
IDE / Debug environment
Debug probe
Debug interface
Processor

Figure 7: High-level representation of how UDE can be deployed in a debug environment

After a debugger is connected to the target development platform, the following sequence could be used to enable debug authentication:

- The debug authentication first happens with verification of user credentials at the debug authentication application. If successful, the authentication generates a debug authentication token (or similar form of authentication information), which is then transferred to the debug authentication manager on-chip either through the debug connection or another interface
- The debug authentication manager verifies the debug authentication token and determines the debug permission configuration (i.e. to indicate which secure partition is allowed to be debugged)
- At context switches of Secure library context, the Secure partition manager checks with the debug authentication manager to see if the next partition being switched into is allowed to be debugged. If yes, the UDE is enabled. Otherwise, UDE is disabled

To enable the ecosystem to work together, some standardisation effort is needed.

- The interface between toolchain/IDE and Debug Authentication application running on the debug host need to be standardised so that a Debug Authentication application can work with multiple toolchains
- The interface between Secure partition manager and debug authentication manager running on the silicon device need to be standardised to allow the Secure partition manager to work with debug authentication manager from different silicon vendors

Ideally, alignment or event standardisation of some of the technologies around debug authentication would help the tool ecosystem to deploy this UDE solution. However, the standardisation of debug authentication itself is not essential for the operations of UDE. All these standardisation and alignment of technologies, update of development tools and enhancement of security firmware (e.g. Trusted Firmware-M) will require effort and time, and collaboration between silicon vendors, debug tool vendors, software vendors and Arm.

## 7. Summary

Combining various new technologies such as system-level features like on-the-fly decryption, lifecycle state management and processor architectural features like TrustZone for Armv8-M, debug authentication and UDE (Unprivileged Debug Extension) in Armv8.1-M, a chip vendor can provide an SoC product that enables software vendors to create and deliver software libraries securely:

- The software asset (of the software libraries) is protected during delivery by encryption of the software image
- The encryption of the software library asset also protects it from untrusted parties during chip manufacturing and packaging
- Security keys for decrypting image is protected by TrustZone
- The software library asset is protected from Non-secure software developers, but at the same time, the Secure APIs inside these libraries can be called by Non-secure applications
- Using Unprivileged Debug Extension (UDE), it is possible to define and restrict debug permission to a single unprivileged software partition, preventing untrusted developers of Secure unprivileged software libraries from stealing secrets from silicon vendors and other vendors of Secure unprivileged software libraries

The UDE solution requires many technologies to work together. Due to the complexity, it is expected to take a while for all the required pieces of technologies to be developed. However, the goal of protecting on-chip software libraries is achievable and will enable a new market for embedded software vendors and OEMs.

To learn more about Armv8.1-M and the UDE, access the Armv8.1-M Architecture Reference Manual here.

## Acknowledgement

I would like to express my gratitude to Arm research team for their effort in enhancing the security of Cortex-M processors, and Arm security researchers in reviewing this paper and providing useful suggestions.

All brand names or product names are the property of their respective holders. Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given in good faith. All warranties implied or expressed, including but not limited to implied warranties of satisfactory quality or fitness for purpose are excluded. This document is intended only to provide information to the reader about the product. To the extent permitted by local laws Arm shall not be liable for any loss or damage arising from the use of any information in this document or any error or omission in such information.

© Arm Ltd. 2020