Renesas RA Family

Injecting and Updating Secure User Keys

Introduction

Cryptography is important because it provides the tools to implement solutions for authenticity, confidentiality, and integrity, which are vital aspects of any security solution. In modern cryptographic systems, the security of the system no longer depends on the secrecy of the algorithm used but rather on the secrecy of the keys.

Renesas MCU security revolves around integrated security engines. There are different types of security engines across the RA MCU. Users can find the specific engine used in a particular MCU from its hardware user's manual.

The security engines can operate in two different modes, called Compatibility Mode and Protected Mode. The application note "Renesas Security Engine Operational Modes" (R11AN0498) explains the definition of the two modes and their use cases. In the context of this application note, the operational differences are:

- In Compatibility Mode, the security engines can inject secure keys as well as plaintext keys. Key injection must be performed using RA Family Flexible Software Package (FSP) APIs. All security engines support Compatibility Mode.
- In Protected Mode, the security engines can inject only secure keys. Key injection must be performed using the MCU's boot firmware. Only the RSIP security engines and SCE9 support Protected Mode.

With this release, this application project demonstrates the following secure key injection processes:

- RSIP-E51A Compatibility Mode AES-128 secure key injection using the RA8M1 MCU
- SCE9 Protected Mode AES-256 and ECC secp256r1 public key secure key injection using the RA6M4 MCU
- SCE7 Compatibility Mode AES-128 secure key injection using the RA6M3 MCU.
- RSIP-E50D Protected Mode ECC secp256r1 public key and private key secure key injection using the RA8P1 MCU

Example keys are provided with the projects. **These keys must not be used in an end-product**, as key reuse constitutes a major security risk. This application note describes how to modify the projects to use custom keys.

Required Resources

Target Devices:

RA8M1/RA8D1/RA8T1 (with RSIP-E51A)

RA8P1, RA8T2 (*), RA8M2 (**), RA8D2 (**) (with RSIP-E50D)

RA4C1 (with RSIP-E31A)

RA4L1 (with RSIP-E11A)

RA6M4/RA6M5/RA4M2/RA4M3 (with SCE9)

RA6M1/RA6M2/RA6M3 (with SCE7, Compatibility Mode only)

RA6T2 (with SCE5_B, Compatibility Mode only)

RA4M1/RA4W1 (with SCE5, Compatibility Mode only)

(*) These devices will be supported starting from FSP v6.1.0 and later.

(**) These devices will be supported starting from FSP v6.2.0 and later.

Development tools and software

- e^2 studio IDE v2025-04.1
- Renesas Flexible Software Package (FSP) v6.0.0
- SEGGER J-Link® USB driver and RTT Viewer
- Renesas Flash Programmer (RFP) v3.19
- Renesas Security Key Management Tool v1.09

The FSP, J-Link USB drivers, and e^2 studio are bundled in a downloadable platform installer available on the FSP webpage at renesas.com/ra/fsp. SEGGER RTT Viewer is available for download free-of-charge from https://www.segger.com/products/debug-probes/j-link/tools/rtt-viewer/. RFP is available for download from https://www.renesas.com/software-tool/renesas-flash-programmer-programming-gui. The free-of-charge edition can be used for the functionality required by this Application Project. The Security Key Management Tool can be downloaded at https://www.renesas.com/software-tool/security-key-management-tool.

Hardware

- EK-RA8P1, Evaluation Kit for the RA8P1 MCU Group (https://www.renesas.com/ek-ra8p1)
- EK-RA8M1, Evaluation Kit for the RA8M1 MCU Group (http://www.renesas.com/ra/ek-ra8m1)
- EK-RA6M4, Evaluation Kit for the RA6M4 MCU Group (http://www.renesas.com/ra/ek-ra6m4)
- EK-RA6M3, Evaluation Kit for the RA6M3 MCU Group (http://www.renesas.com/ra/ek-ra6m3)
- Workstation running Windows® 10
- One USB device cable (type-A male to micro-B male)
- One USB device cable (type-A male to type-C male) (for the EK-RA8P1 board)

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e^2 studio IDE and Arm®-TrustZone®-technology based development models with e^2 studio. In addition, the application note assumes that you have some knowledge of RA Family MCU security features. You can reference the section "Security Features" in the hardware user's manual for background knowledge preparation for the cryptographic key injection. The intended audience are product developers, product manufacturers, product support, or end users who are involved with any stage of injecting or updating secure keys with Renesas RA Family MCUs.

Contents

1. Wrapped Key Creates Root of Trust

1.1 Introduction to Root of Trust

Roots of trust are highly reliable hardware, firmware, and software components that perform specific, critical security functions (https://csrc.nist.gov/projects/hardware-roots-of-trust). In an IoT system, a root of trust typically consists of identity and cryptographic keys rooted in the hardware of a device. It establishes a unique, immutable, and unclonable identity to authorize a device to exist in the IoT network.

Secure boot is part of the services provided in the Root of Trust in many security systems. The application is authenticated using public key encryption. The associated keys are part of the Root of Trust in the system. Device Identity, which consists of Device Private Key and Device Certificate, is part of the Root of Trust for many IoT devices.

From the above Root of Trust discussion, we can see that leakage of cryptographic keys can bring the secure system into a risky state. Protection of the Root of Trust involves limiting key accessibility within the cryptographic boundary only, with keys that are securely stored and preferably unclonable. The Root of Trust should be locked from read and write access by unauthorized parties.

The Renesas user key management system and the MCUs can provide all the above desired protection.

1.2 Introduction to Security Engine and Associated Keys

The security engine (RSIP or SCE) is an isolated subsystem within the MCU. The security engine contains hardware accelerators for symmetric and asymmetric cryptographic algorithms, as well as various hashes and message authentication codes. It also contains a True Random Number Generator (TRNG), providing an entropy source for cryptographic operations. The security engine is protected by an Access Management Circuit, which can shut down the security engine in the event of an illegal external access attempt. Figure 1 shows the conceptual diagram of the security engine. Refer to Table 1 for exactly what cryptographic operations are supported by each type of security engine.

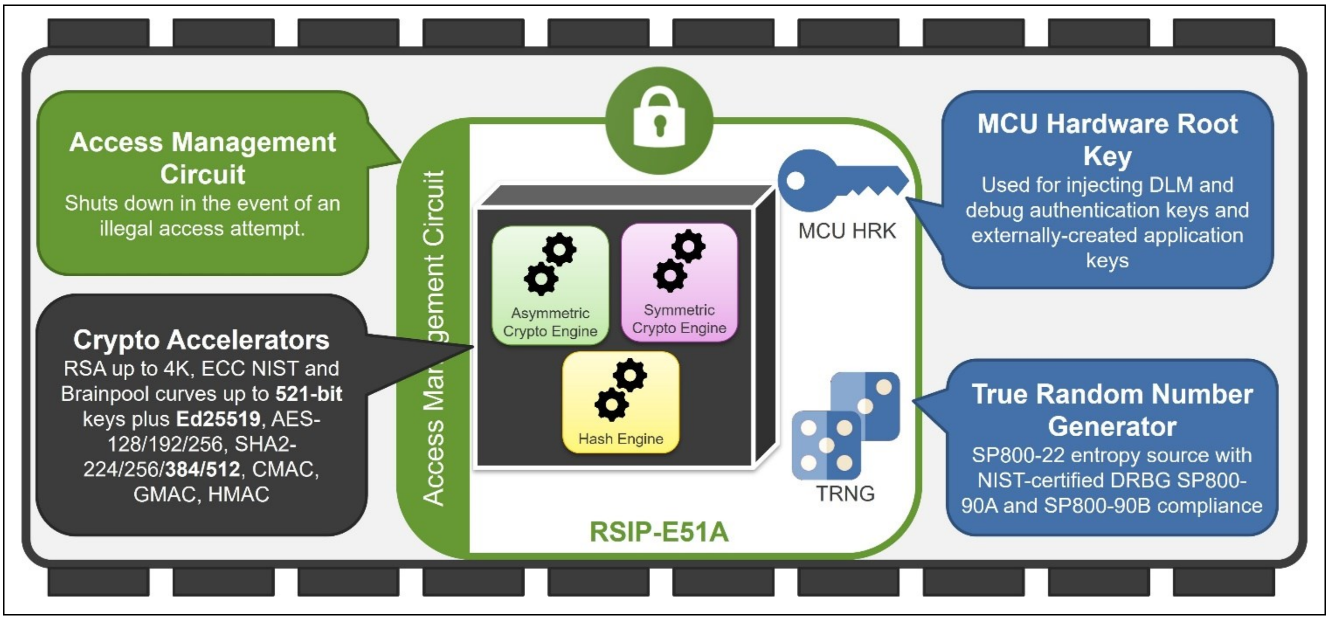


Figure 1. Security Engine RSIP-E51A

The Hardware Root Key (HRK) is not a single key that is physically stored. It is represented in this presentation as such for simplifying the description of the concepts. The security engine contains internal RAM for operations that deal with sensitive material such as plaintext keys. This RAM is not accessible outside the security engine.

The security engine has its own dedicated internal RAM, enabling all crypto operations to be physically isolated within the security engine. This, combined with advanced key handling capability, means that it is possible to implement applications where there is no plaintext key exposure on any CPU-accessible bus.

Secure key storage and usage is accomplished by storing application keys in wrapped format, encrypted by the MCU's Hardware Unique Key and tagged with a Message Authentication Code. Since wrapped keys can only be unwrapped by the security engine within the specific MCU that wrapped them, the wrapping mechanism provides unclonable secure storage of application keys. The RA Family also provides a secure key injection mechanism to securely provision your devices.

The security engine is packed full of cryptography features that you can leverage in your higher-level solutions, giving you the option to use hardware acceleration to reduce both execution time and power consumption. Table 1 summarizes the different security engines and their associated cryptographic functionalities.

Table 1. Renesas Secure IP and Security Engine Cryptographic Capabilities

Functions			RA8P1, RA8x2	RA8x1	RA4L1	RA4C1	RA6M4, RA6M5 RA4M2, RA4M3	RA6M1, RA6M2 RA6M3, RA6T1	RA6T2	RA4M1, RA4W1
		Cryptographic Isolation								
	Security Engines	Security Engine	RSIP-E50D	RSIP-E51A	RSIP-E11A	RSIP-E31A	SCE9	SCE7	SCE5_B	SCE5
		Identity & Key Exchange (Asymmetric)								
	RSA	Key Gen, Sign/Verify	Up to 4K	Up to 4K	-	-	Up to 4K	Up to 2K	-	-
	ECC	Key Gen, ECDSA, ECDH	Up to 521 bits	Up to 521 bits	Up to 256 bits	Up to 384 bits	Up to 512 bits	Up to 384 bits	-	-
	Ed25519	EdDSA	Y	Y	-	Y	-	-	-	-
	DSA	Sign/Verify	-	-	-	-	-	Y	-	-
		Privacy (Symmetric)								
	AES	ECB, CBC, CTR	128/192/256	128/192/256	128/256	128/256	128/192/256	128/192/256	128/256	128/256
		GCTR	128/192/256	128/192/256	128/256	128/256	128/192/256	128/192/256	128/256	128/256
		XTS	128/256	128/256	-	-	128/256	128/256	-	-
		CCM, GCM, CMAC	128/192/256	128/192/256	128/256	128/256	128/192/256	128/192/256	128/256	128/256
	ChaCha20	Poly1305	Y	-	-	-	-	-	-	-
		Data Integrity								
	Hash	GHASH	Y	Y	Y	Y	Y	Y	Y	Y
		HMAC	Y	Y	Y	Y	Y	Y	-	-
		SHA-2 (224/256)	Y	Y	Y	Y	Y	Y	-	-
		SHA-2 (384/512)	Y	Y	-	Y	-	-	-	-
	TRNG	HW Entropy, SP800-90B	Y	Y	Y	Y	Y	Y	Y	Y
		Key Handling								
	Wrapped	Secure key storage	256-bit HUK	256-bit HUK	256-bit HUK	256-bit HUK	256-bit HUK	128-bit DHUK	128-bit DHUK	128-bit DHUK
	Plaintext	Legacy compatibility	Y	Y	Y	Y	Y	Y	Y	Y

The features of the various Security Engines are as follows:

- SCE5 and SCE5_B provide hardware-accelerated symmetric encryption for confidentiality.
- SCE7 adds asymmetric encryption and advanced hash functions for integrity and authentication. SCE7 AES, SHA, and random number generation DRBG are NIST CAVP certified.
- SCE9 extends asymmetric encryption support for RSA up to 4K and enhanced key storage capability with a Hardware Unique Key (HUK). The full complement of algorithms is NIST CAVP certified.
- RSIP-E11A is the most basic RSIP type, offering all necessary core functionalities including ECC. RSIP-E31A builds on this by expanding ECC and SHA-2 support up to 384 bits and adding Ed25519 support.
- RSIP-E51A offers a broader set of cryptographic features. It supports RSA, including signature and key generation, up to 4K bits, ECC up to 521 bits, and a wide range of symmetric AES modes with all three key lengths. Additionally, it provides SPA/DPA resistance for both Protected Mode and Compatibility Mode.
- RSIP-E50D offers the widest range of functions among all types, encompassing all functionalities of the earlier versions. It further enhances data integrity with SHA3 and extends symmetric cryptographic capabilities with support for ChaCha20-Poly1305.

The MCU-unique Hardware Unique Key (HUK) is a 256-bit random key for RSIP and SCE9 and a 128-bit random key for SCE5_B, which is injected into the Renesas factory. This key is stored in a wrapped format using an MCU-unique key wrapping mechanism.

The derived MCU-unique Hardware Key (DHUK) for SCE5 and SCE7 is a derived MCU unique key that serves the same purpose as the HUK in terms of user key wrapping. The derived HUK for SCE7 and SCE5 is never stored and is accessible only by the SCE and not by application code.

Since the HUK/DHUK is unique to each MCU, even if an attacker were able to extract a stored application key, another MCU would not be able to use it.

In Protected mode, a Key Update Key (KUK) can be used to securely update the user keys when a device is deployed in the field. The KUKs are injected during end-product manufacturing via the MCU's programming interface or using FSP Crypto Driver. To update keys in a device that is deployed in the field, the new key must be wrapped with one of the previously injected KUKs. Compatibility Mode does not support KUK usage, so key updates must be performed using plaintext key injection. In addition to replacing keys that have been compromised, many security policies require key rotation or key update (re-keying) on a regular basis. It is recommended to consider injecting multiple KUKs.

1.3 Renesas Secure Key Injection Advantages

Secure key injection and update, combined with the security engine's support of wrapped keys, address many vulnerabilities associated with using plaintext keys:

- Plaintext keys are never stored in code flash. In the event of a program memory breach, the sensitive key material is protected.
- Plaintext keys are never stored in RAM. In the event of malicious code executing on the system, the sensitive key material is still protected.
- Keys can be securely stored in code flash, data flash, MRAM or even copied into external memory, enabling unlimited secure key storage.

In addition, the Renesas key wrapping techniques protect against device cloning, as discussed below.

1.3.1 Advantages of Key Wrapping over Key Encryption

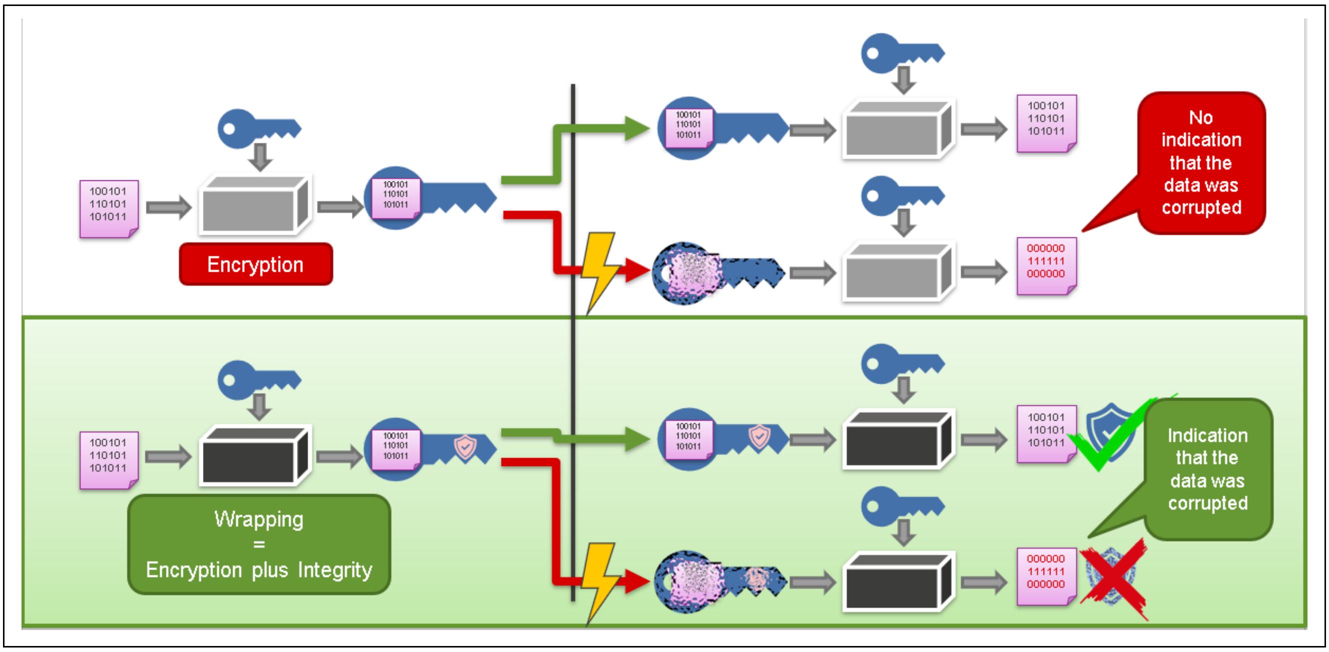


Figure 2. Key Wrapping versus Key Encryption

It is important to understand the difference between wrapping and encrypting for secure asset storage.

When data is encrypted and sent to another recipient, if that recipient has the same key, they can decrypt the data. This results in a confidential exchange of information. However, what if there was a problem with the transmission of the encrypted data? If the recipient unknowingly receives corrupted information, the decryption algorithm will generate garbage data with no indication that the original data has been corrupted.

Wrapping solves this problem by appending a Message Authentication Code to the encrypted output for integrity checking.

1.3.2 Advantages of Key Wrapping using MCU HUK

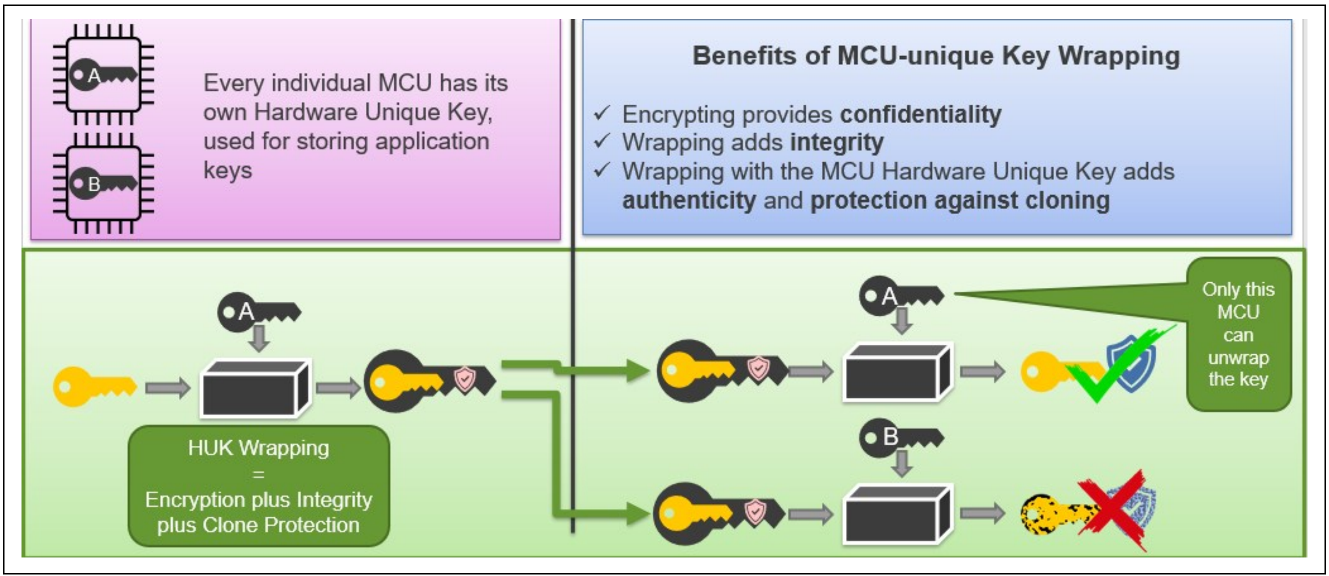


Figure 3. Key Wrapping using the HUK

Using the MCU Hardware Unique Key (HUK) to wrap the stored keys adds another protection feature – clone protection. If the wrapped key is transmitted or copied to another MCU, that MCU's HUK will not be able to either unwrap or use the copied key. Even if all of the MCU contents are copied onto another device, the keys cannot be used or exposed.

1.4 Renesas RA MCU Factory Boot Firmware Limitations for SCE9

Secure key injection via the serial programming interface is not supported for RSA 3K, RSA 4K, ECC secp256k1, and Key-Update Keys on some older versions of the Renesas RA MCUs due to factory Boot Firmware limitations. The user needs to use a Renesas Flash Programmer (RFP) to read out the Boot Firmware version and confirm the support for the Secure Key Injection of the above-mentioned keys. Refer to the RFP user's manual Flow of Operations section to access the Bootloader Firmware version by using the **Read Device Information** menu.

- V1.2.04 – WS1: secure user key inject command is not supported
- V1.3.10 – WS2: user key inject command is not supported
- V1.5.22 – CS: user key inject command is supported, but it does not support RSA 3K, RSA 4K, secp256k1, or KUK
- V1.6.25 and above – MP: no limitations

The part information silkscreened on the device can also be checked, though it is recommended that the boot firmware version be confirmed as described above. Boot firmware limitations exist for the following MCUs:

- RA4M2 - All WS and ES devices
- RA4M3 - All WS, ES, and CS devices (date code 014AZ00)
- RA6M4 - All WS, ES, and CS devices (date code 014AZ00). MP device with date codes 028AZ00, 031AZ00
- RA6M5 - All WS and ES devices

Please note that some EK-RA6M4 and EK-RA4M3 Evaluation Kits may contain affected silicon. The following list shows the serial numbers of the affected kit. Note that all early adopter kits with WS or ES silicon are also affected.

- EK-RA4M3 – Serial numbers 219243 – 219542
- EK-RA6M4 – Serial numbers 215938 – 216237 and 218497 - 218996

If your application requires secure key injection of RSA 3K, RSA 4K, ECC secp256k1, or Key-Update Keys and your evaluation kit does not support it, please contact your local Renesas Sales representative.

2. Wrapped Key Injection Use Cases and Injection Procedure Overview

This section provides an overview of the wrapped key injection use cases and the general steps for the injection procedure of each use case. A step-by-step walkthrough of the wrapped key injection procedures is provided in later sections.

2.1 Wrapped Key Types

Renesas RA Family MCUs have the unique ability to store and use cryptographic keys in wrapped format. Wrapping involves encrypting and signing the key with either the MCU's Hardware Unique Key (HUK) or a derived key based on the MCU's Unique ID. Since these Key Encryption Keys are unique for each individual MCU, even if an attacker were able to extract the wrapped key, another MCU will not be able to use it. For information on the supported key types for each MCU, please refer to the Hardware User's Manual or the FSP User's Manual.

2.2 General Steps for Secure Key Injection and Update

Secure Key Injection for RSIP/SCE Protected Mode and SCE5_B is performed via the MCU boot interface, demonstrated here with the Renesas Flash Programmer (RFP). Secure Key Injection for RSIP/SCE Compatibility Mode is performed through the FSP. Key preparation steps where key material is exposed in plaintext must be performed in a secure environment.

2.2.1 Key Injection

There are three high-level steps for key injection. Section 3 guides the user in establishing the PGP encrypted communication channel between the user and the Renesas DLM Server. Sections 4, 5, and 1.1 provides step-by-step walkthroughs of how to perform the three high-level steps for the secure key injection.

1. The first step in the secure key injection process is to use the Renesas Device Lifecycle Management (DLM) service to wrap an arbitrary User Factory Programming Key (UFPK) (in green) using the Renesas Hardware Root Key (HRK) (in blue). The UFPK is a 256-bit value selected by the user. The same UFPK can be used to inject any number of keys.

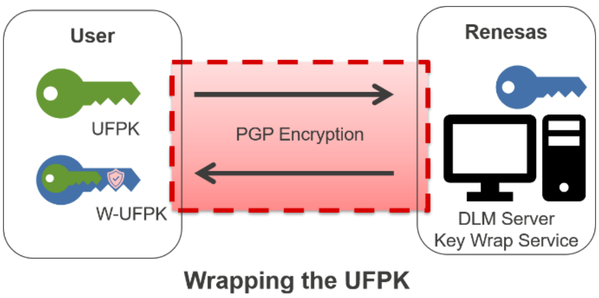


Figure 4. Wrapping the UFPK using DLM Server

2. Next, the user key (in yellow) must be wrapped with the UFPK.

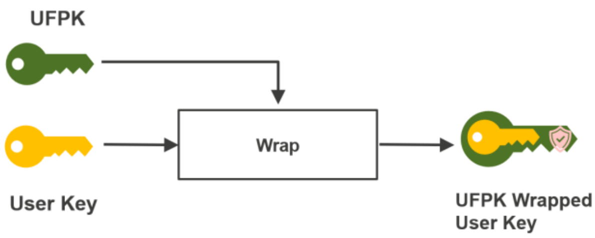


Figure 5. Wrap the User Key with the UFPK

3. Finally, the user key is injected by providing the wrapped UFPK (W-UFPK) and the wrapped user key to the secure key injection mechanism of the security engine. Note that this is a conceptual representation of the secure user key injection. Once the wrapped user key is generated, the tool generates one secure key injection file, which includes the wrapped user key and the W-UFPK. This file (.rkey file) will be used in the secure key injection project.
   For the Protected mode, injecting the wrapped user key should be performed using the MCU boot interface. For compatibility mode, injecting the wrapped user key should be performed using the FSP key injection PSA Crypto API. Note that keys injected via the MCU boot interface (that is, the factory boot firmware) cannot be used in Compatibility Mode with RSIP and SCE9. SCE5_B supports only Compatibility Mode; therefore, SCE5_B key injection via the MCU boot firmware interface is performed in Compatibility Mode. MCUs with the SCE7 and SCE5 security engines do not support key injection via the MCU boot firmware interface.

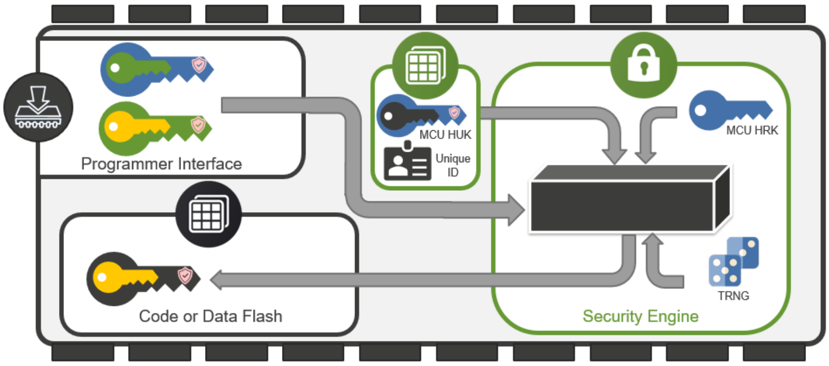


Figure 6. Inject User Key over the Serial Programing Interface

2.2.2 Key Update

Since injecting new keys in the field is usually done to replace older keys (key rotation or re-keying), this process is referred to as a “key update”. In Protected mode, to enable secure key updates in the field, one or more Key-Update Keys (KUK) must be injected during production programming/provisioning, as described above. Compatibility Mode does not support KUK usage, and key updates must be performed using plaintext key injection.

KUKs, like other cryptographic keys, can be stored in either code flash or data flash (if available on the MCU). Injection of the KUK uses the same procedure as injecting other user keys, as described in the section 2.2.1. Since the KUK is the only mechanism by which new keys can be injected/wrapped, it is highly recommended that multiple KUKs be injected during production provisioning. This enables the KUK to be rotated or revoked to adhere to an infrastructure security policy or to respond to a key exposure security breach.

Additional KUKs CANNOT be injected after the programming interface is disabled. Once a product is in the field with its programming interface disabled, new keys can ONLY be injected via a pre-existing KUK.

The KUKs may be stored in any code or data flash location during production. This location will be passed to the key update API for the injection of the new user key. A user can inject multiple KUKs and provide a scheme to rotate the keys based on a timed schedule or key leakage event. For security reasons, we recommend that users disable the programming interface prior to deploying to the field.

There are two high-level steps for key updates. Note that the KUK must already reside in the MCU.

1. Use the KUK (in grey) to wrap the new user key (in yellow).

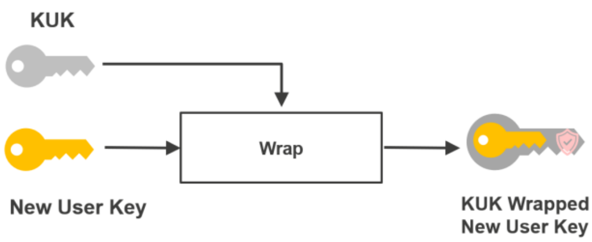


Figure 7. Wrap the New User Key with a KUK

2. Use the FSP and the previously injected KUK to inject the new user key. The new user key is wrapped by the MCU HUK (in black). Note that the APIs for the two modes are provided by different FSP modules.

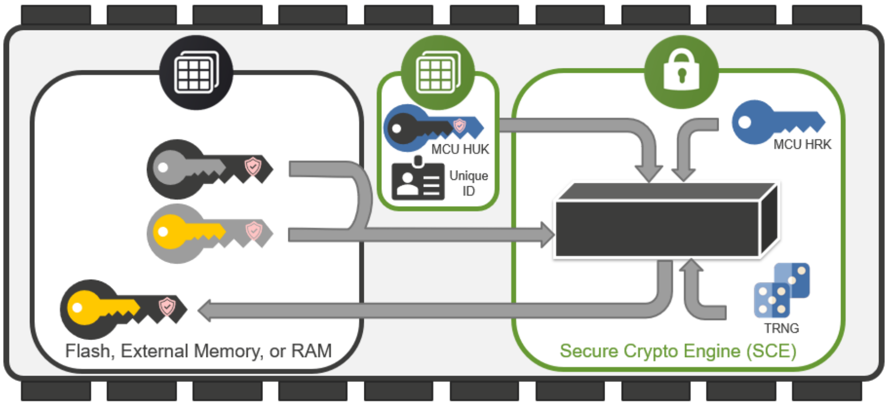


Figure 8. Update the User Key

2.3 Overview of the Operations for Evaluating the Example Projects

The example projects in this application project demonstrate the secure key injection and update capabilities of Renesas RA Family MCUs using sample keys. Sections 3, 4, and 5 describe the steps needed to replace these sample keys with custom keys.

The following graphic shows the flow of this preparation work, plus the example project for SCE9 (RA6M4 example) and RSIP-E50D (RA8P1 example). The block outlined in red is the scope of the functionality of the example project.

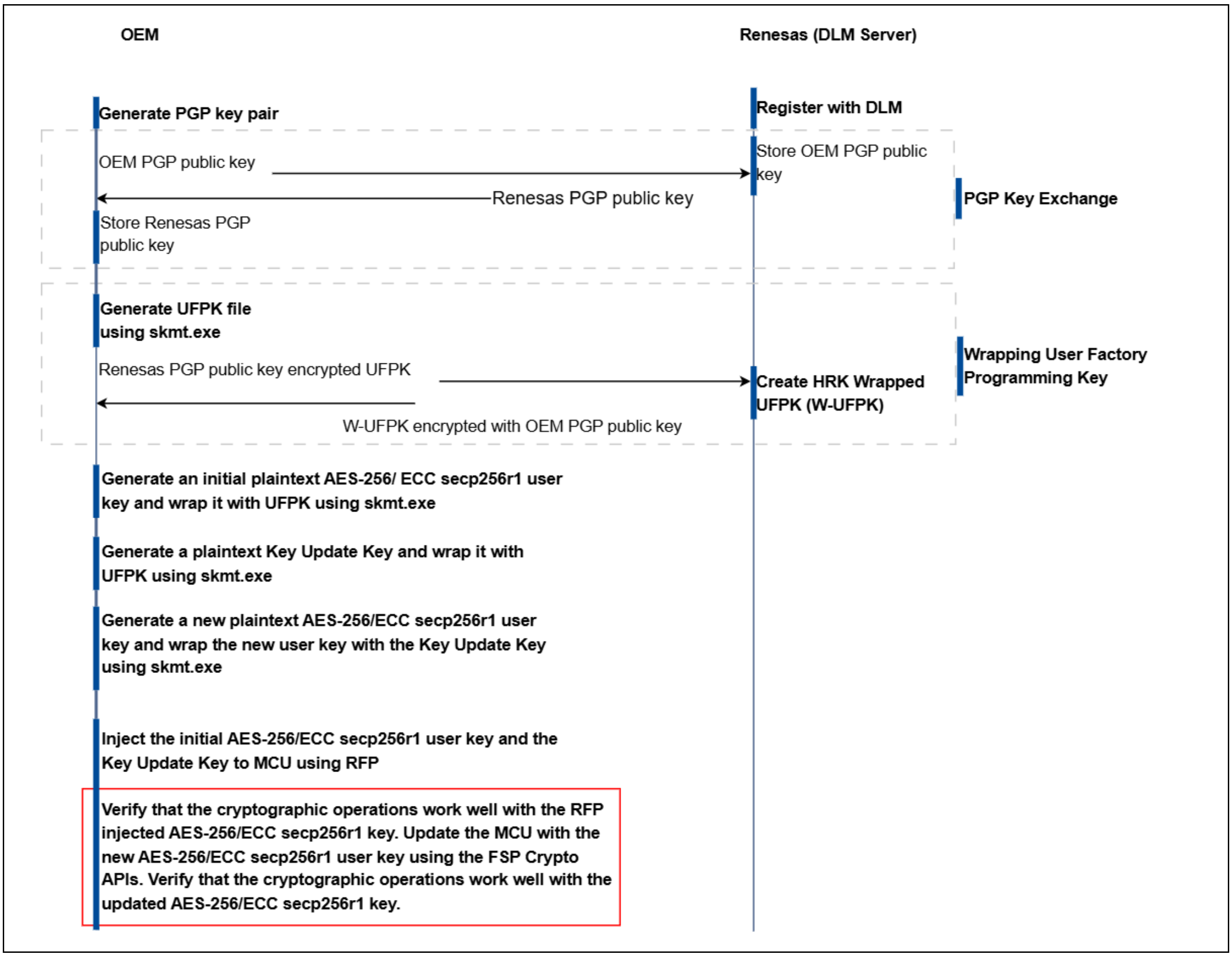


Figure 9. Operational Flow Injecting and Updating User Key for SCE9 and RSIP-E50D Protected Mode

The following graphic shows the flow of this preparation work plus the example project for SCE7 (RA6M3 example) and RSIP-E51A (RA8M1 example). The block outlined in red is the scope of the functionality of the example project.

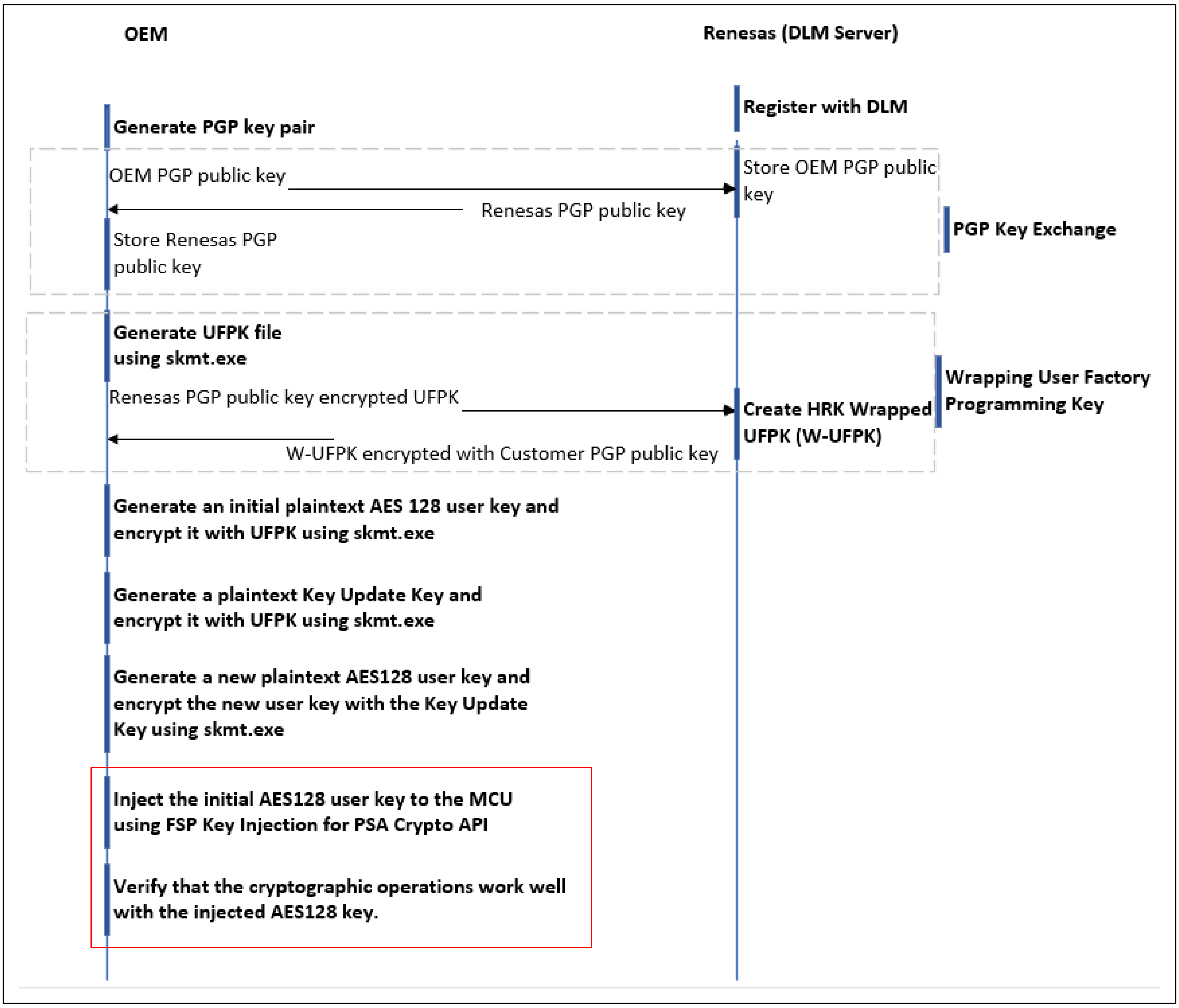


Figure 10. Operational Flow Injecting User Keys for SCE7 and RSIP-E51A Compatibility Mode

2.4 Tools Used in the Secure Key Injection and Update

There are three tools used in the secure key injection and update besides e² studio, which is used as the software project development environment. Refer to the corresponding section mentioned below for details on obtaining, setting up, and using these tools.

- **Gpg4win**
  This tool is used in section 3 to establish a PGP encrypted communication channel between user and the Renesas Key Wrap server. Using this tool, the user can generate a user PGP key pair, perform key exchange with the Renesas DLM server, and assist the reception of the W-UFPK.
- **Renesas Security Key Management Tool (SKMT)**
  This tool is used in section 4, section 5 and section 6 to generate the following three key files:
  - User key: to be injected to MCU via RFP or FSP API
  - Key update key: to be injected to MCU via RFP
  - New user key wrapped using the KUK: to be updated by an FSP API
- **Renesas Flash Programmer (RFP)**
  This tool is used in section 54 to inject the User key and KUK when using the security engine Protected Mode.

3. Using the Renesas Key Wrap Service

The Renesas Key Wrap Service must be used to obtain a wrapped UFPK (W-UFPK) for the specific MCU Group and security engine operational mode. All key material exchange is performed with PGP encryption. This section explains the steps to establish this PGP-encrypted communication channel between the user and the Renesas Key Wrap Server. This is a one-time process and does not need to be repeated for different MCUs.

3.1 Create PGP Key Pair

If you already have a PGP key pair, that key can be used for the key exchange process. Otherwise, the instructions below describe one method for creating a PGP key pair.

The PGP software demonstrated here is GPG4Win, which can be downloaded from this URL: http://www.gpg4win.org/

The screenshots included in this application note are based on `gpg4win-4.0.0.exe`. There may be minor graphic interface updates with later versions. However, the functionality used in this application note should persist.

Download and install Kleopatra:

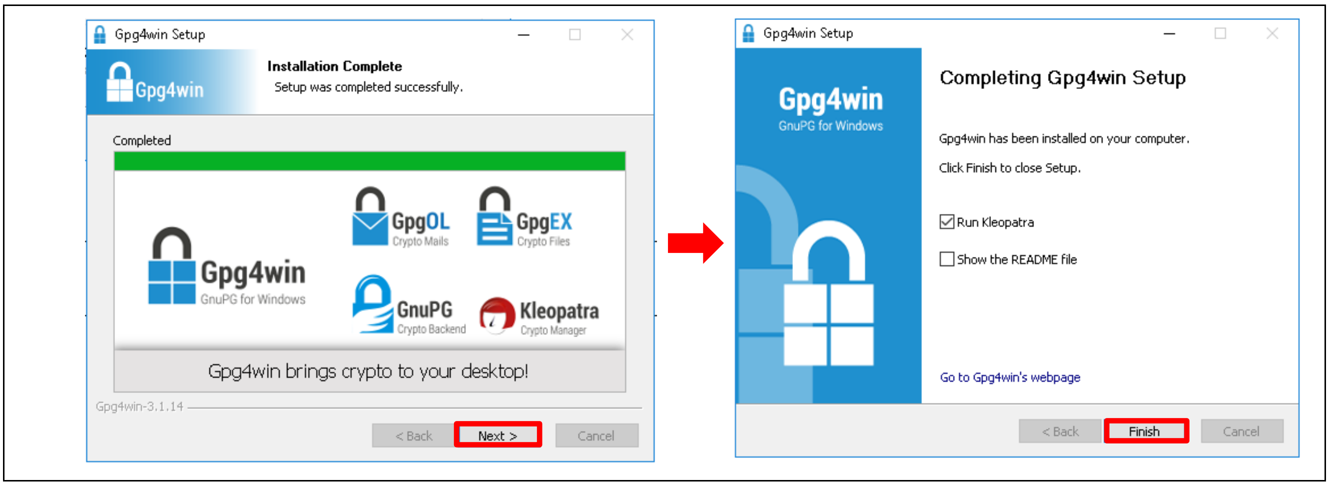


Figure 11. Download and Install Kleopatra

Launch Kleopatra and create a PGP Key Pair.

1. Click **File > New Key Pair**
2. Choose **Create a personal OpenPGP key pair**.

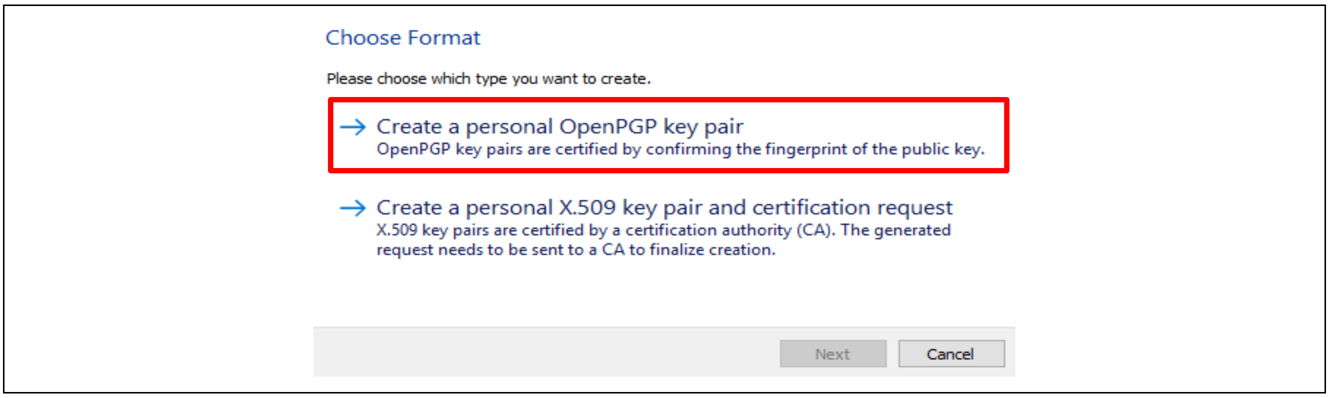


Figure 12. Create a Personal Open PGP Key Pair

3. Provide a **Name** and **Email**. **Note that even though these are marked as optional, at least one entity must be provided to move to the next stage.** Check **Protect the generated key with a passphase**.

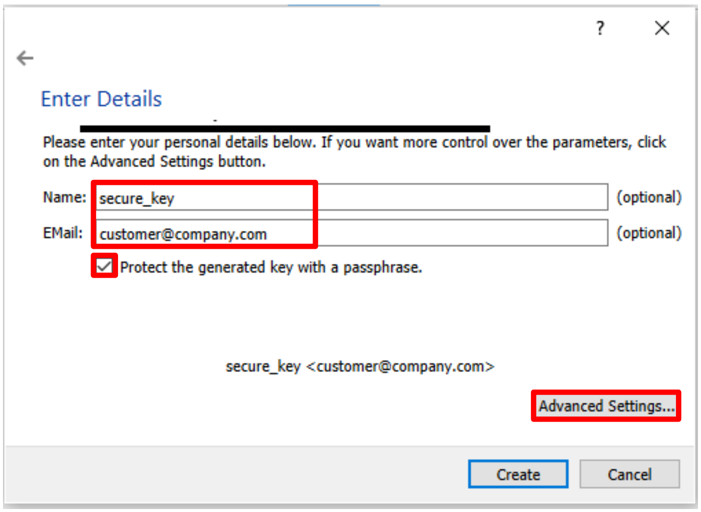


Figure 13. Provide Name and Email

4. Click **Advanced Settings** and select **RSA** as the key type.

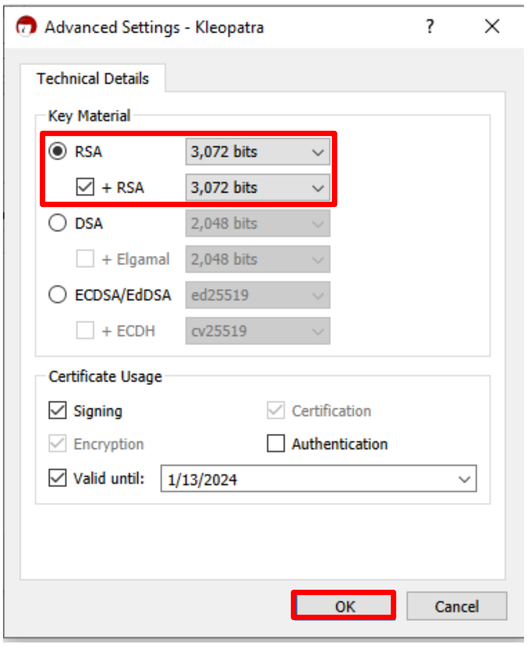


Figure 14. Select RSA Encryption

5. Click **Create** and provide a passphrase twice to protect the private key. Then click **OK**. **Be sure to save your passphrase.**

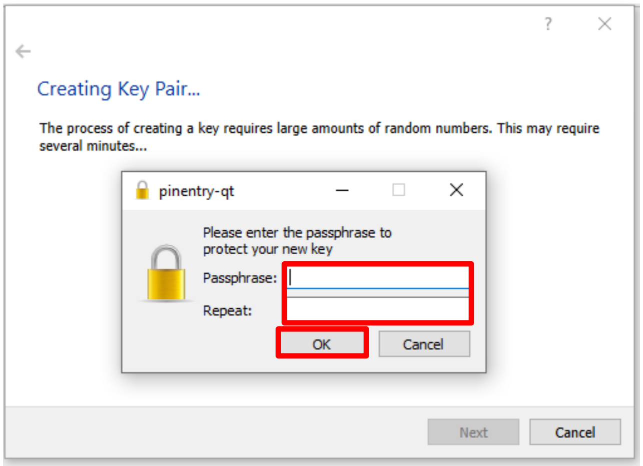


Figure 15. Define a Passphrase

6. The PGP key pair should be created successfully. Click **Finish**.

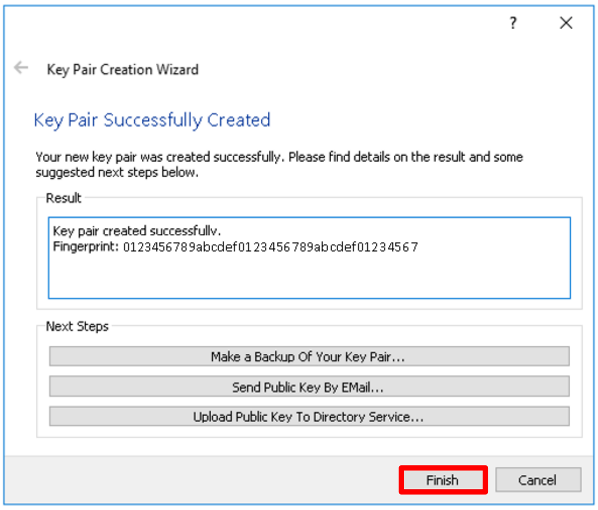


Figure 16. PGP Key Pair Created

7. A new item will be created in Kleopatra. Right-click on the keypair just created and select **Export**.

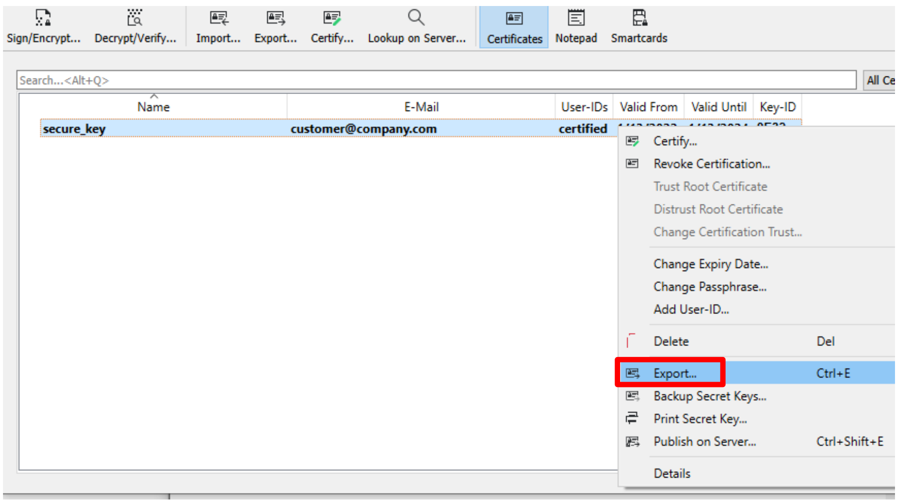


Figure 17. Export the User PGP Public Key

8. Save the public key to a file with an `*.asc` extension. In this example, this file is renamed to `customer_public.asc`. Click **Save**.

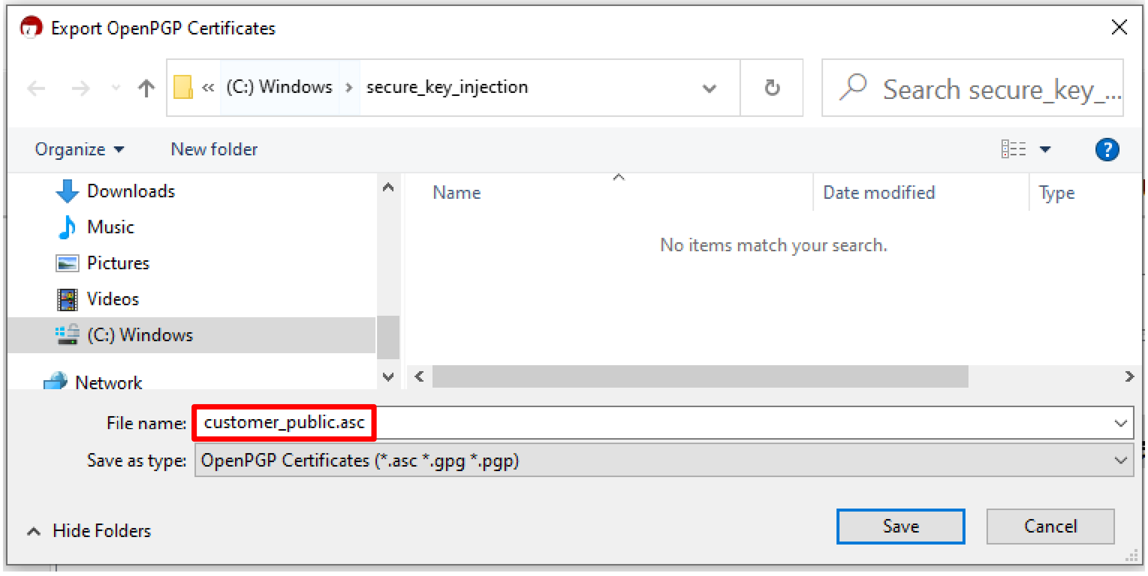


Figure 18. Save the PGP Public Key to a Folder

3.2 Registration with DLM Server

The first time you use the Renesas Key Wrap service, you will have to register with the Renesas DLM Server.

1. Open the URL https://dlm.renesas.com/keywrap in a browser and click **New registration**.

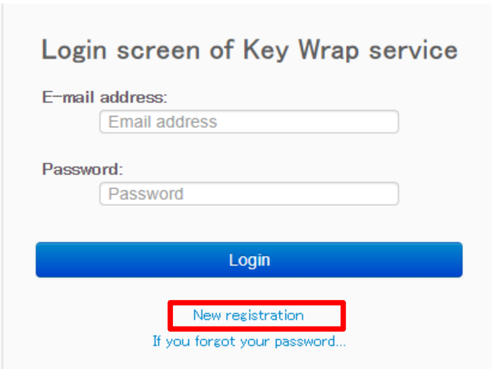


Figure 19. Start Registration with Renesas DLM Server

2. Follow the prompt to provide a **valid** email address and click **Send mail**.

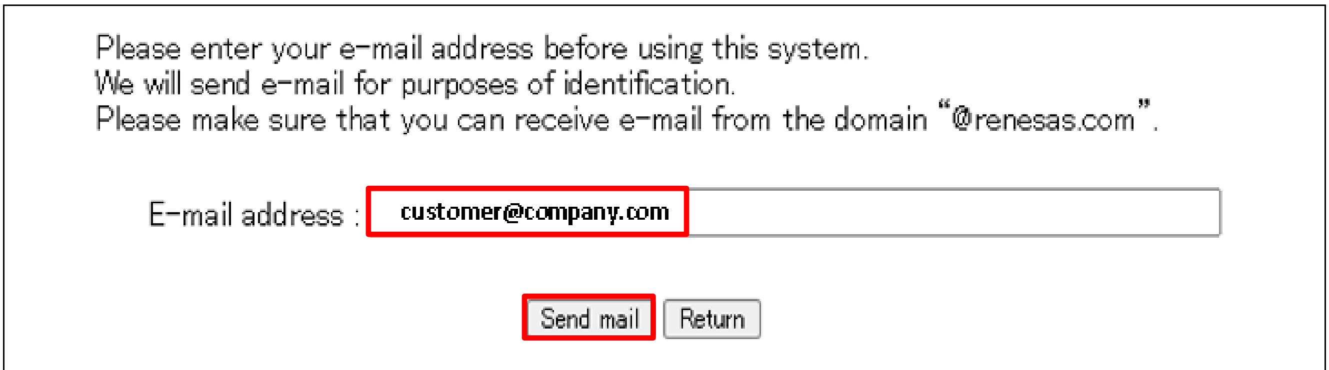


Figure 20. Register User Email Address

After clicking **Send mail**, the following screen will appear. Click **Return**.

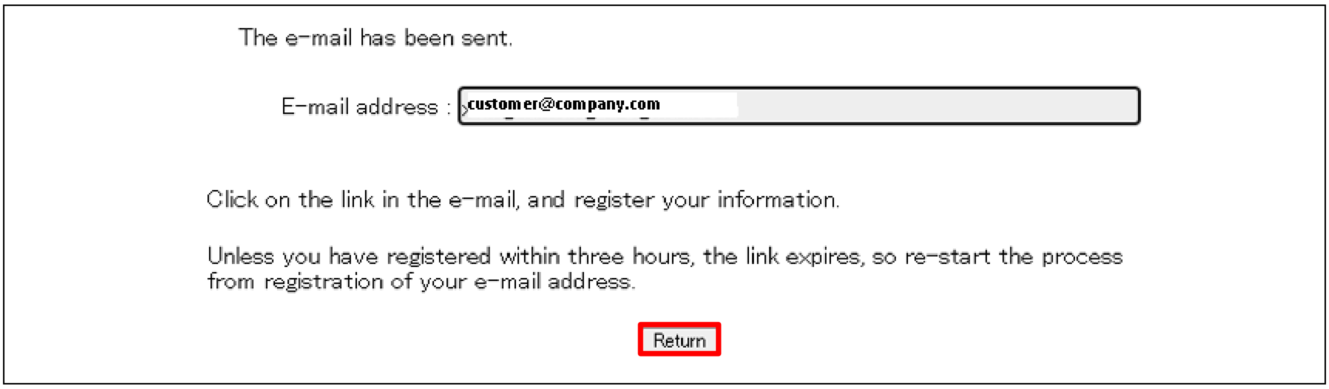
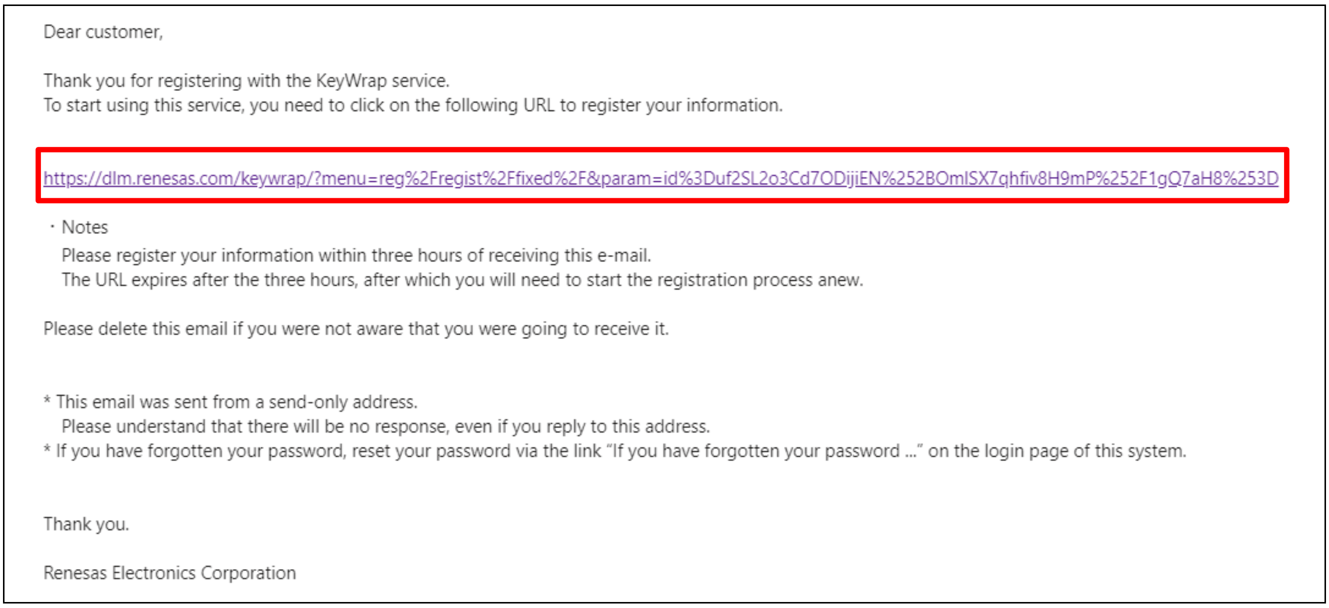


Figure 21. Acknowledge Email Transmission

3. You should receive an email similar to the one shown below. Click on the URL provided to confirm your registration.



Dear customer,

Thank you for registering with the KeyWrap service.
To start using this service, you need to click on the following URL to register your information.

https://dlm.renesas.com/keywrap/?menu=reg%2Fregist%2Ffixed%2F&param=id%3Duf2SL2o3Cd7ODijiEN%252BOmlSX7qhfiv8H9mP%252F1gQ7aH8%253D

・Notes
Please register your information within three hours of receiving this e-mail.
The URL expires after the three hours, after which you will need to start the registration process anew.

Please delete this email if you were not aware that you were going to receive it.

* This email was sent from a send-only address.
Please understand that there will be no response, even if you reply to this address.
* If you have forgotten your password, reset your password via the link "If you have forgotten your password ..." on the login page of this system.

Thank you.

Renesas Electronics Corporation

Figure 22. Registration Confirmation Email

4. Follow the prompts to provide your name and company name and create a password. Click the **Next (confirmation)** button. Note that the password must consist of 8 to 32 alphanumeric characters and may include the symbols "!" and "@".

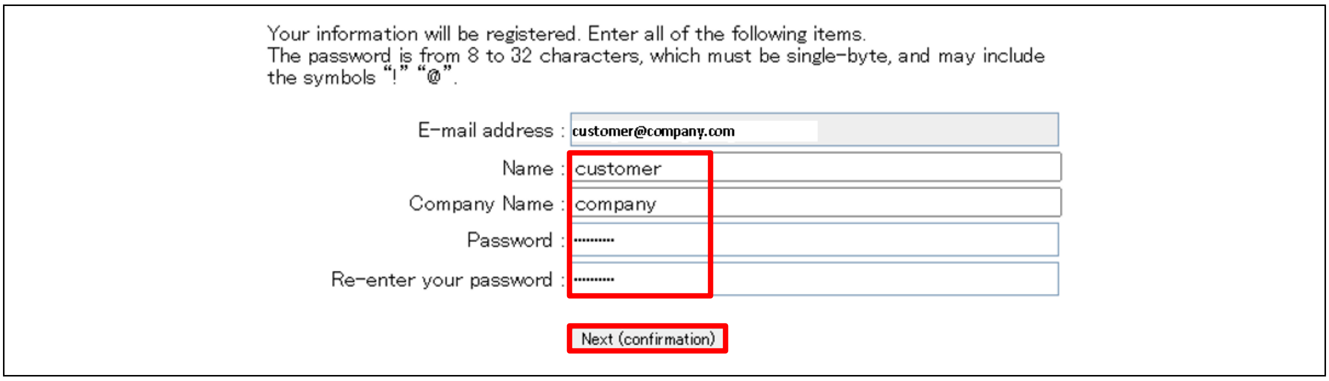


Figure 23. Confirm Registration

After the confirmation screen is displayed, click on the **Register** button to complete the user registration.

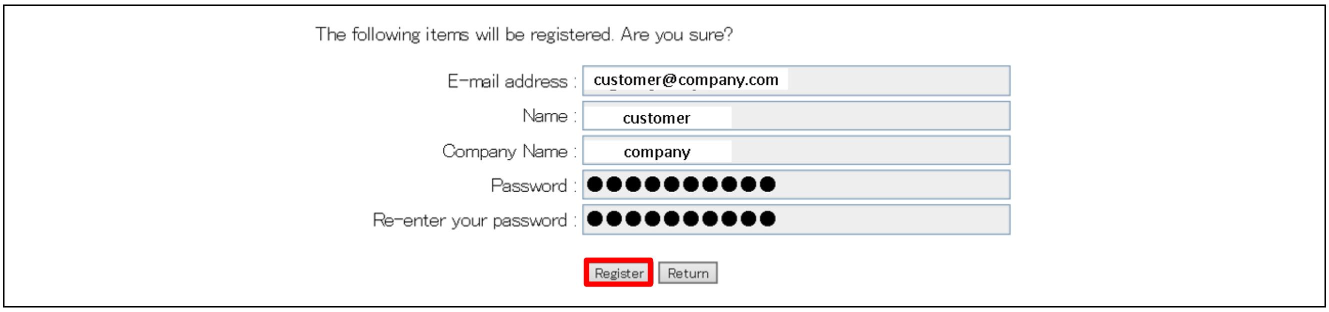


Figure 24. Finish the Registration

3.3 Exchange User and Renesas PGP Public Keys

If you have not already exchanged PGP keys with the Renesas DLM server, follow the steps below.

1. After successfully registering the user information, the following screen will open. Click the **Start service** button to start using the key encryption system.

Figure 25. Start DLM Key Wrapping Service

2. When the agreement warning shows up, scroll down to the bottom of the **Trusted Secure IP Key Wrap Agreement** and click **I agree**. You will then be logged into the DLM server. Note that the Agreement will come up every time you log into the DLM server.

Figure 26. Agreement for Using the Renesas DLM Server

3. When you log into the DLM system, the window below appears. Click **PGP key exchange**.

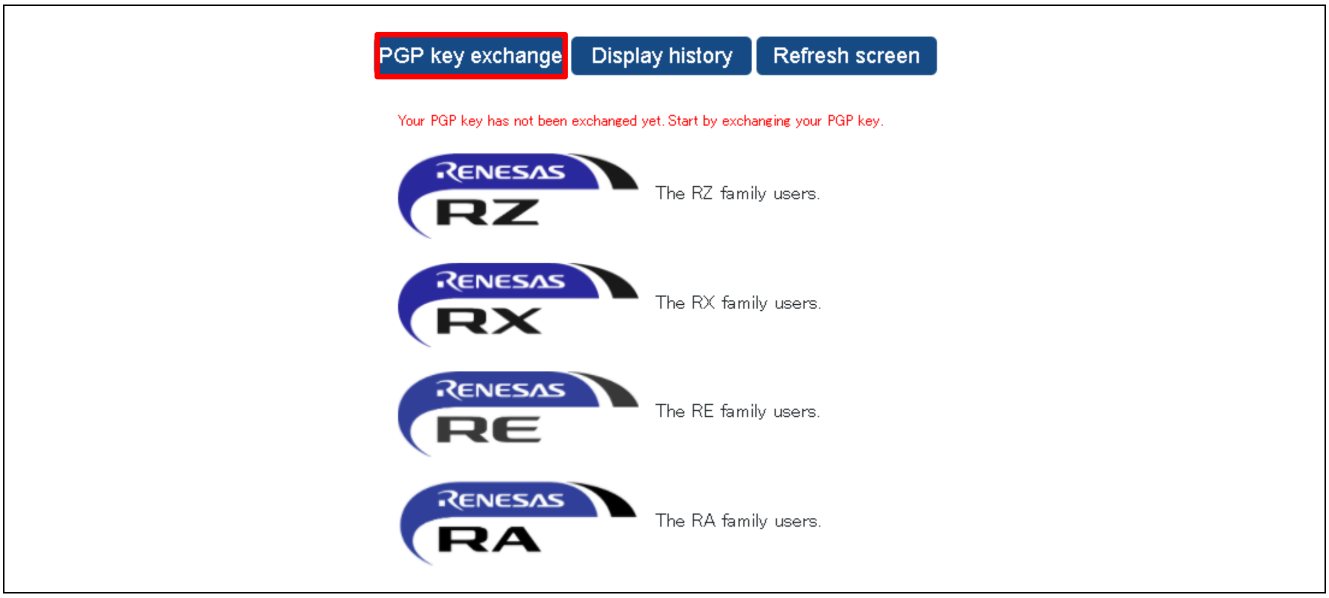


Figure 27. Start PGP Key Exchange

4. Click **Reference** and select the public key generated earlier (`customer_public.asc`). Notice that the fingerprint of the Renesas PGP public key is displayed. This will be used to certify the Renesas public key after you receive it.

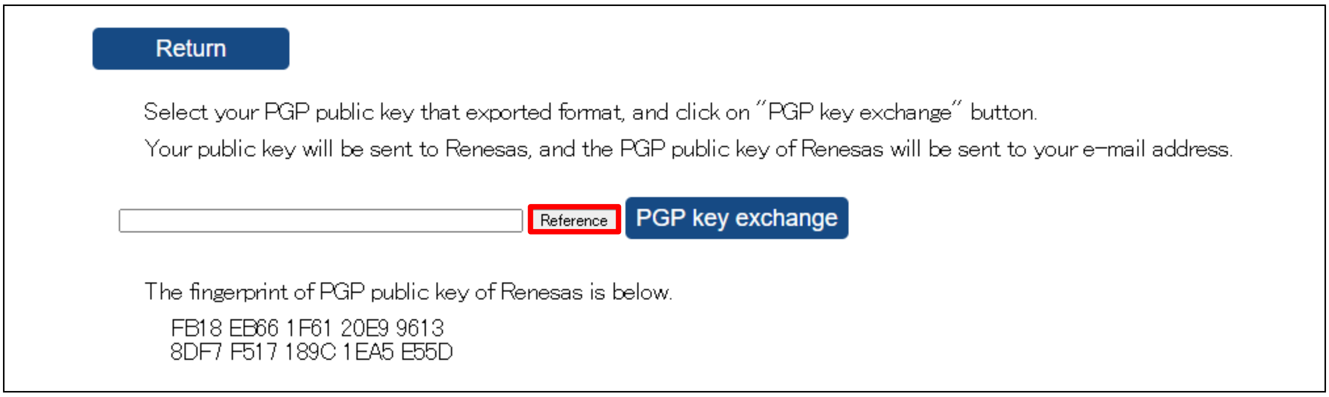


Figure 28. Browse the Customer PGP Public Key

5. Click the **PGP key exchange**.

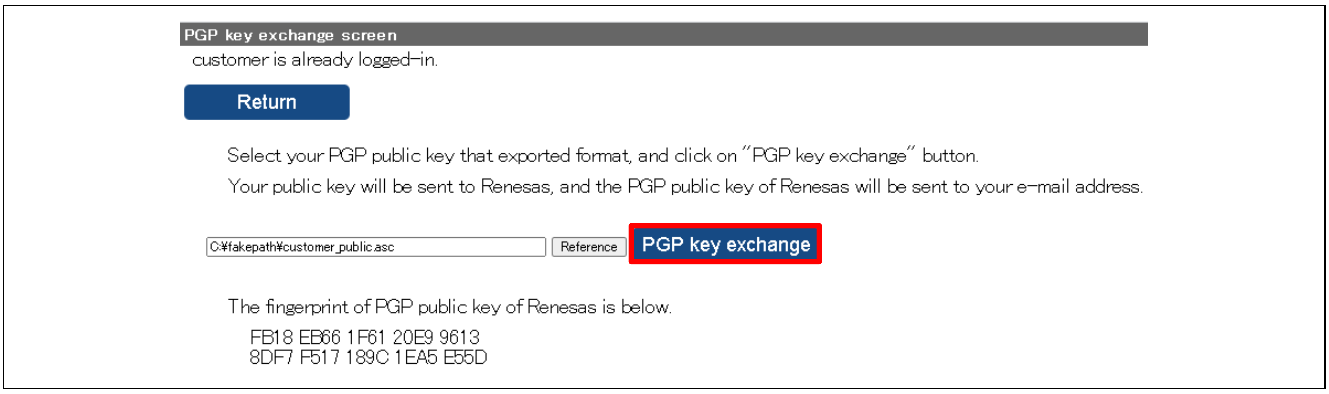


Figure 29. Exchange Keys

6. Once the PGP public key is submitted, click **Return**.

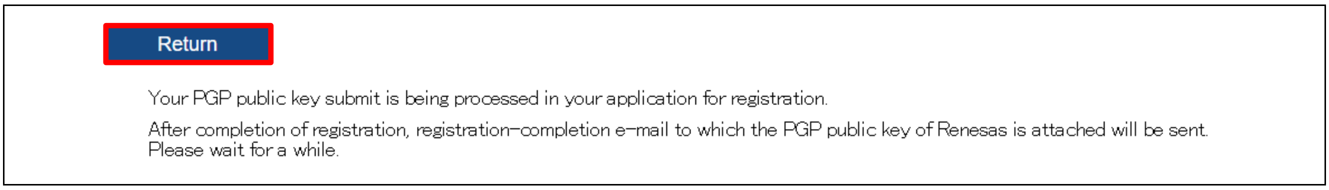


Figure 30. Wait for Renesas's PGP Public Key

7. You will receive an email from Renesas at the email address registered with the DLM server with the contents as shown below if the key exchange is successful. It typically takes about one to two minutes to receive this email.
   Note that a PGP public key can be registered any number of times. The latest PGP public key that has been registered successfully is used for encryption. All previously registered PGP public keys are discarded.

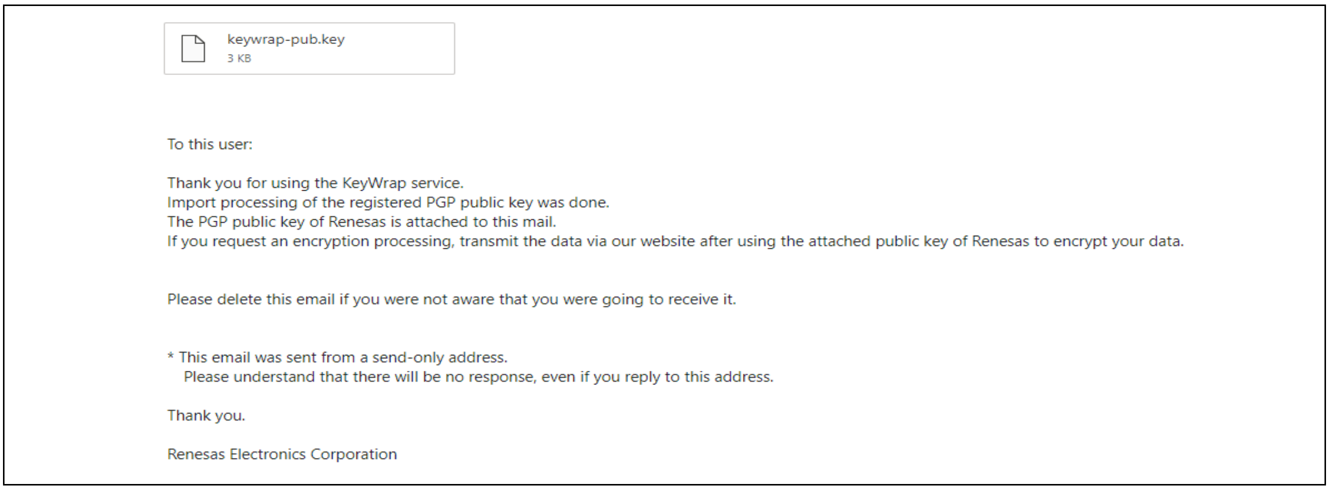


Figure 31. Receive the Renesas PGP Public Key

Save the Renesas PGP public key file (`keywrap-pub.key`).

8. Go back to the Kleopatra application and import the Renesas PGP Public key to Kleopatra as shown below.

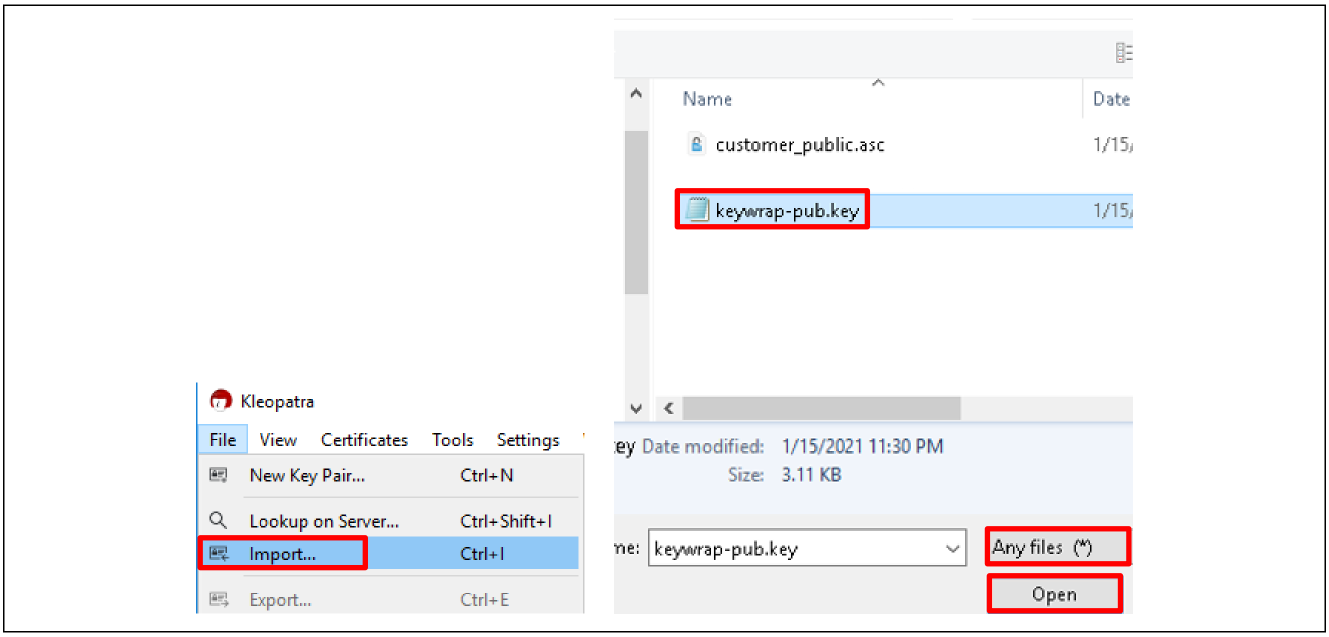


Figure 32. Import Renesas Public Key

9. After **Open** is clicked, a new item is added in Kleopatra as **not certified**.

Figure 33. Renesas Public Key is Imported

10. Confirm that the Fingerprint displayed is the same as what is shown on the screen represented in Figure 29. Click **Certify**.

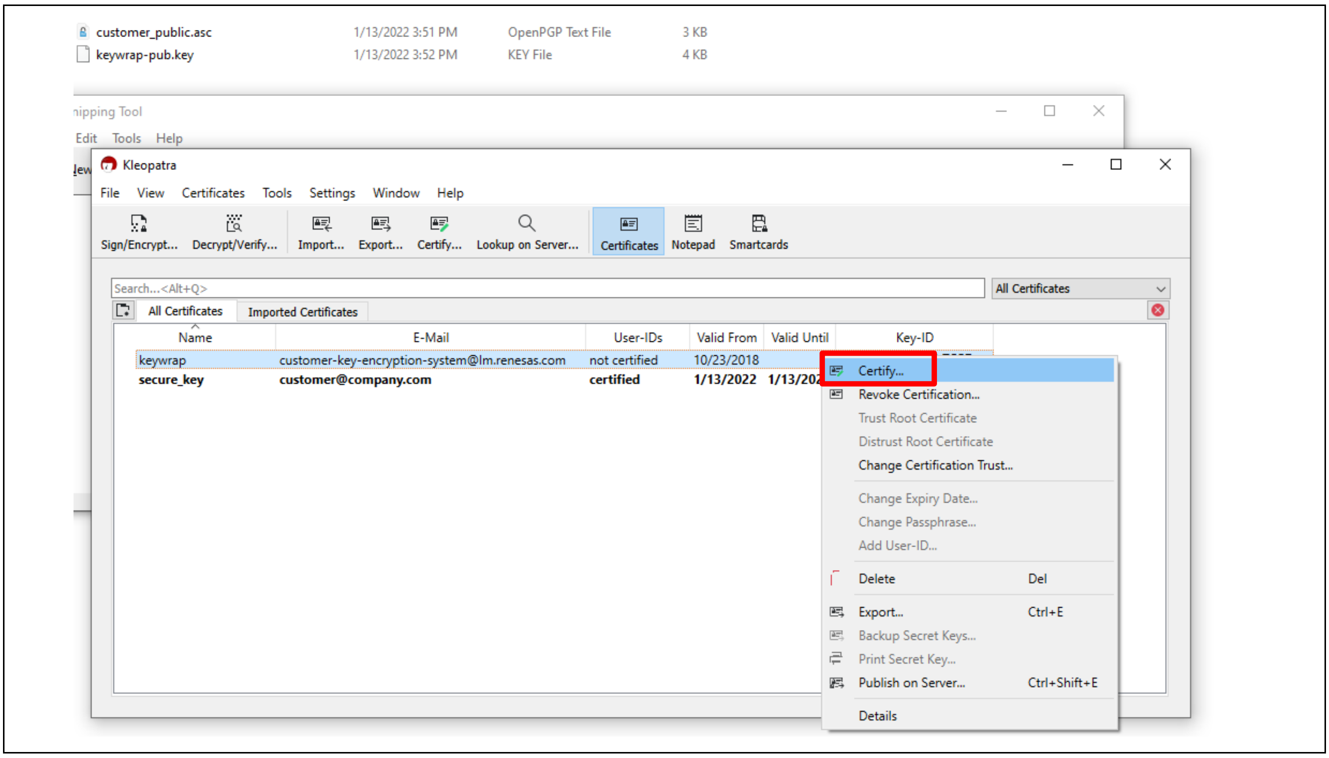


Figure 34. Confirm the Fingerprint and Certify the Renesas Public Key

11. Click **Certify** again from the following screen.

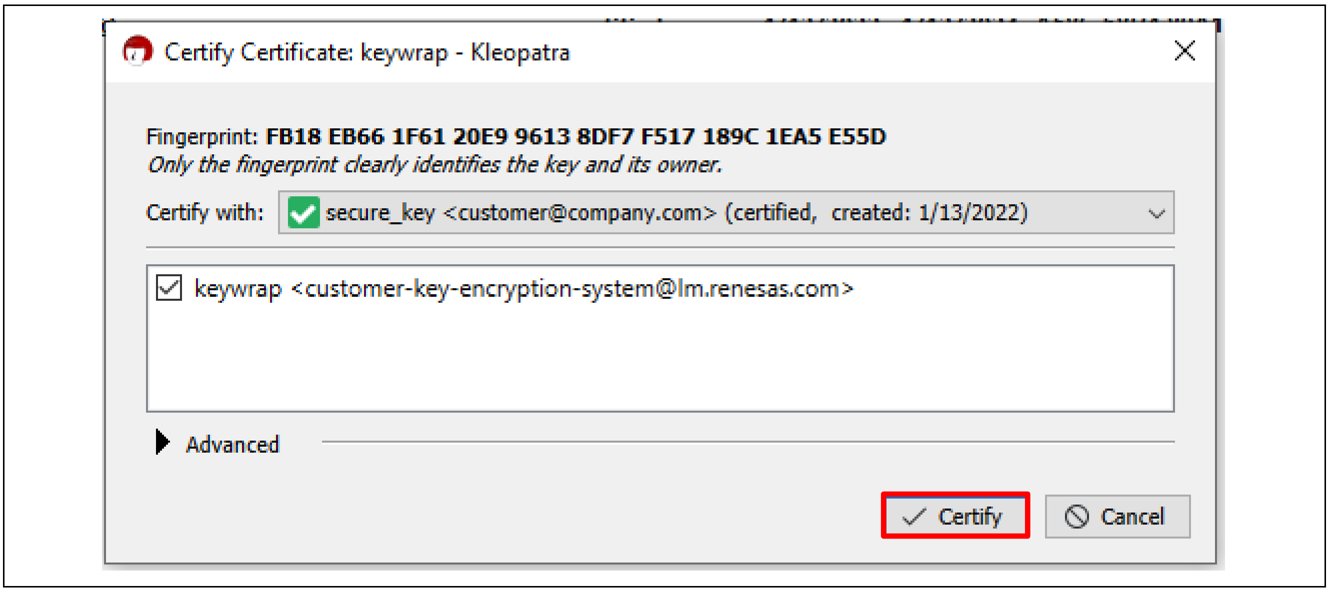


Figure 35. Certify the Certificate

12. Provide the passphrase to unlock the secure key.

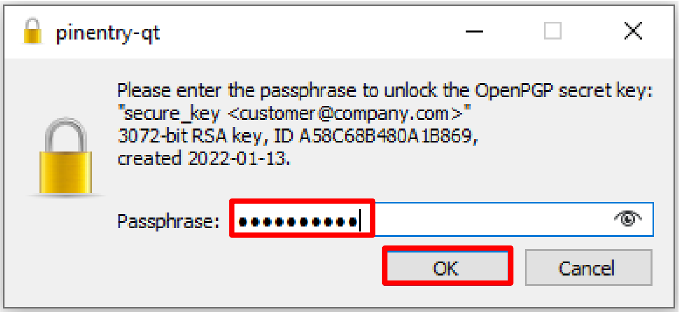


Figure 36. Provide the Passphrase

13. The following item will pop up upon successful certification. Click **OK**.

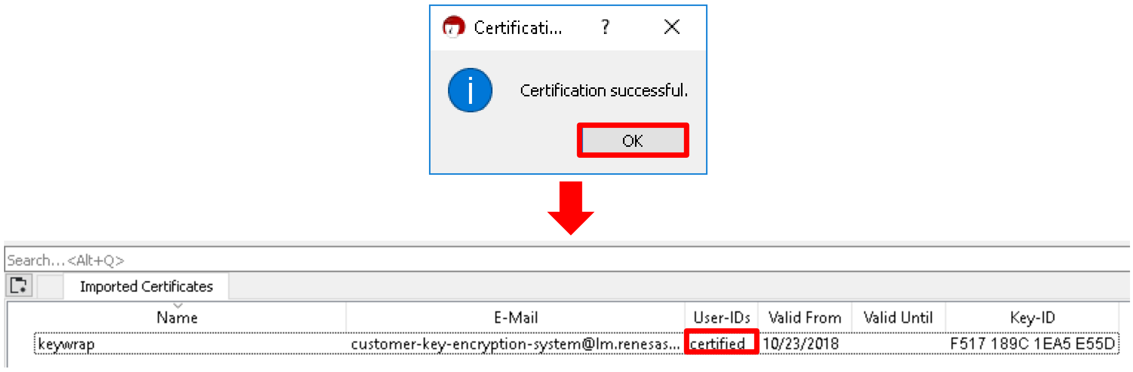


Figure 37. Successful Certification

4. Wrapping the User Factory Programming Key Using the Renesas Key Wrap Service

If you do not already have a W-UFPK for your target MCU Group, follow the steps below to wrap a UFPK with the Renesas Hardware Root Key as described by Figure 4.

4.1 Renesas Security Key Management Tool

The Renesas Security Key Management Tool (SKMT) performs several functions during the secure key injection process. Open the following link to access the latest SKMT:

https://www.renesas.com/software-tool/security-key-management-tool

From the above link, find the **Downloads** area and download the latest Security Key Management Tool installer. This tool supports Windows, Linux, and macOS. The screenshots in this document came from the Windows environment.

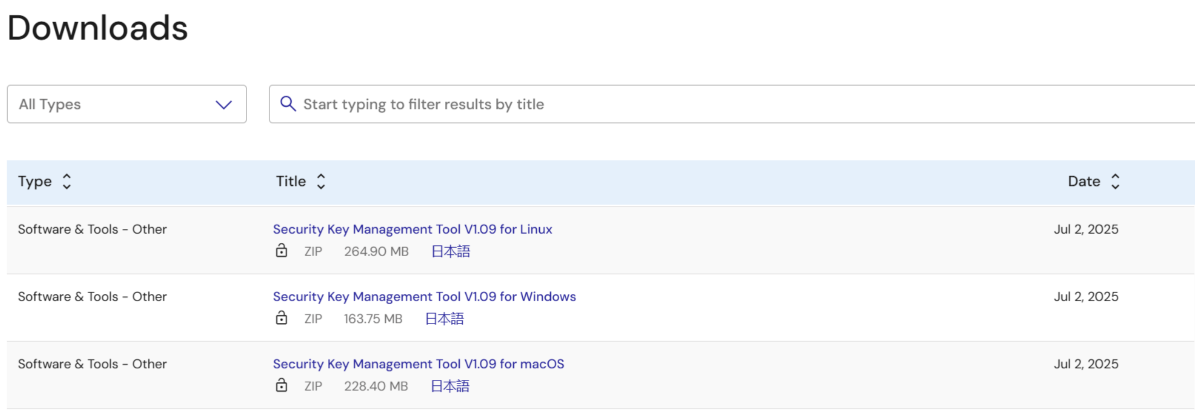


Figure 38. Download the Security Key Management Tool for Windows, Linux or macOS

Once the installer executable is downloaded, right-click on the installer and select **Run as administrator** to install this tool. Follow the prompt to select the **Setup Language**. Currently, both English and Japanese are supported. Next, select the installation folder. By default, it will be installed into `C:\Renesas\SecurityKeyManagementTool\`. If a previous version is installed, the old version will be overwritten.

The User's Manual of this tool is located in the `\DOC` folder. We recommend that you read through the user's manual before proceeding to the following section.

The SKMT provides two interfaces for users: a Command Line Interface (CLI) and a Graphic User Interface (GUI). The CLI interface is typically used for production support and the GUI interface is primarily intended for development usage. This application note will explain how to use both interfaces to perform key injection and update.

4.2 Creating the User Factory Programming Key using the SKMT GUI Interface

Define a UFPK and convert it to a binary format that is compatible with the Renesas Key Wrap Service. This can be done using the Renesas Security Key Management Tool (SKMT).

The same UFPK can be used for all RA Family MCUs. However, the corresponding W-UFPK may be different as it depends on the specific MCU Group. To avoid confusion and mistakes, it is recommended to choose the correct RA MCU Family when generating the UFPK using the SKMT GUI interface and name them different based on the MCU family.

Double-click **SecurityKeyManagementTool.exe** to launch the GUI interface.

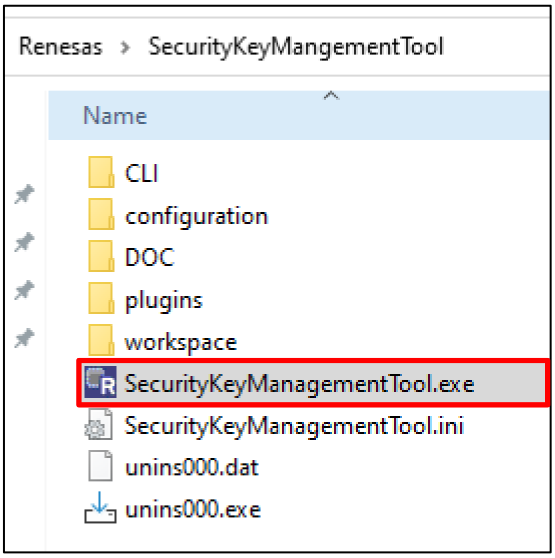


Figure 39. Launch SKMT GUI Interface

To use the example projects included this application project, set the UFPK to
`000102030405060708090A0B0C0D0E0F000102030405060708090a0b0c0d0e0f`

Note that the 32-byte UFPK must be provided in big-endian format.
It is important to select the correct MCU family and security engine mode when using the SKMT tool.

- RA8P1 has RSIP-E50D, for the RA8P1 protected mode example project included, in the **Overview** window, select **RA Family, RSIP-E50D Security Functions and Protected Mode**

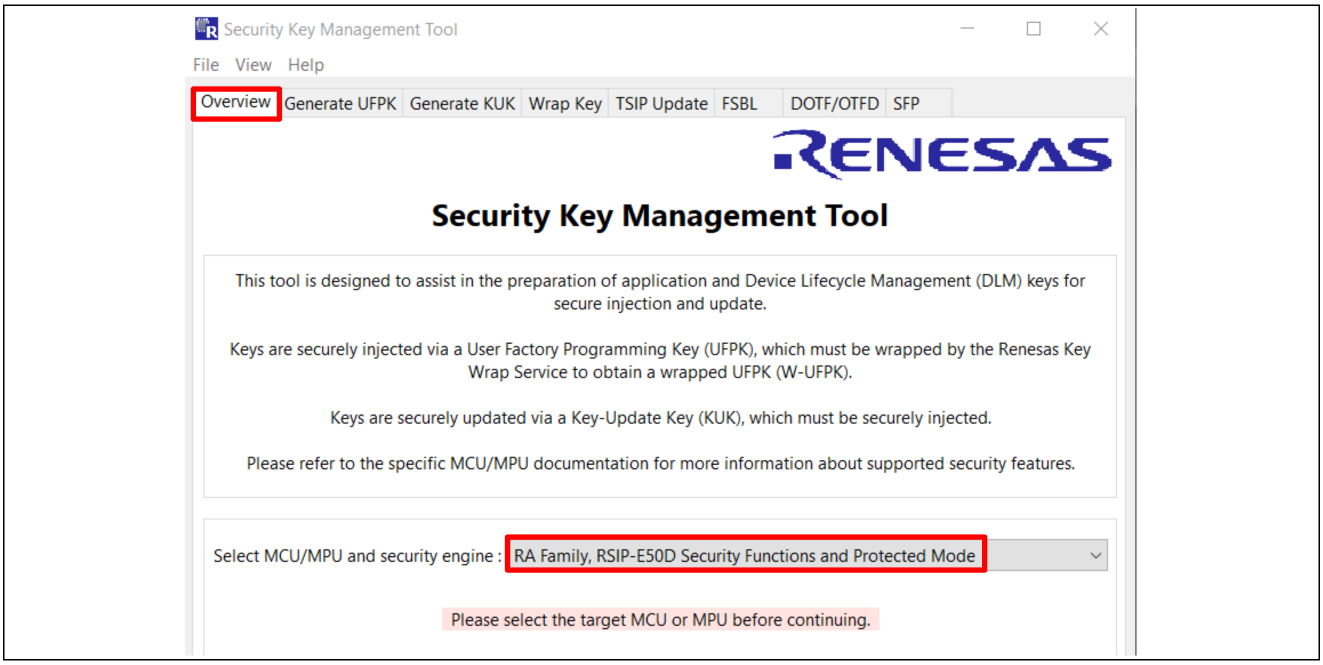


Figure 40. Select RA Family, RSIP Protected Mode

- RA8M1 has RSIP-E51A, for the RA8M1 compatibility mode example project included, in the **Overview** window, select **RA Family, RSIP-E51A Compatibility Mode**.

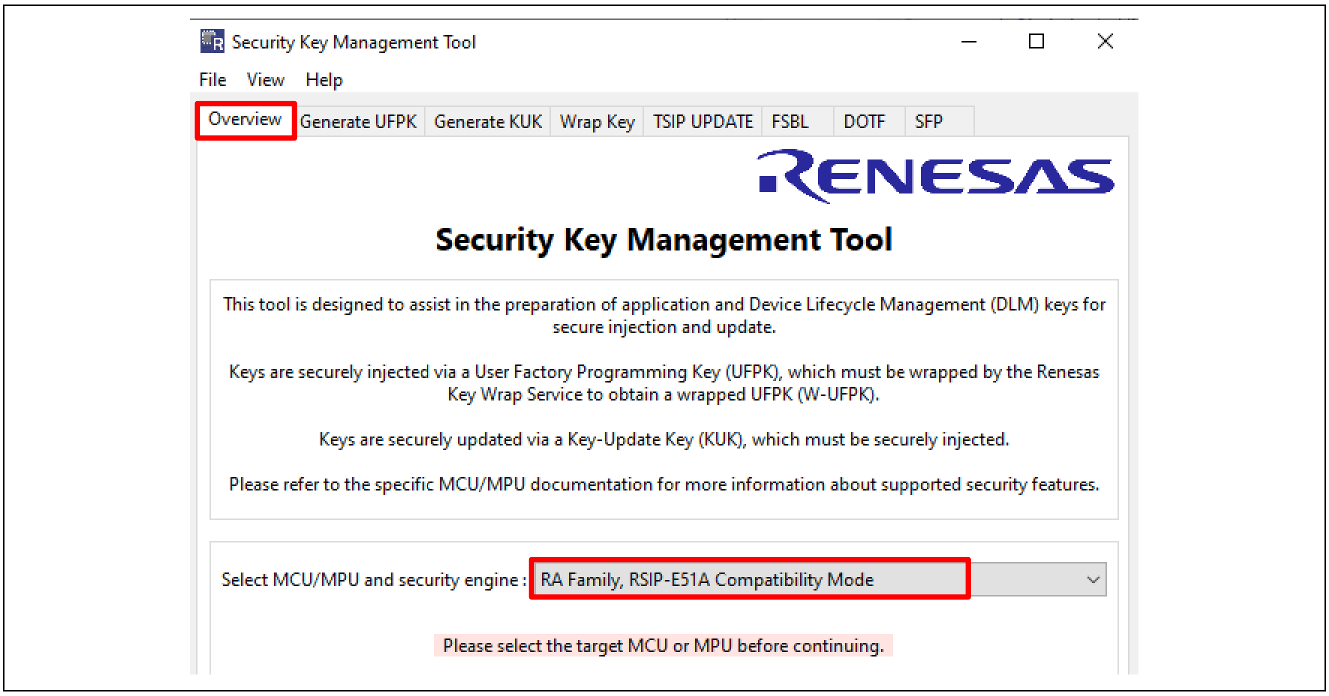


Figure 41. Select RA Family, RSIP-E51A Compatibility Mode

- RA6M4 has SCE9, for the protected mode example project included, in the **Overview** window, select **RA Family, SCE9 Security Functions, and Protected Mode**.

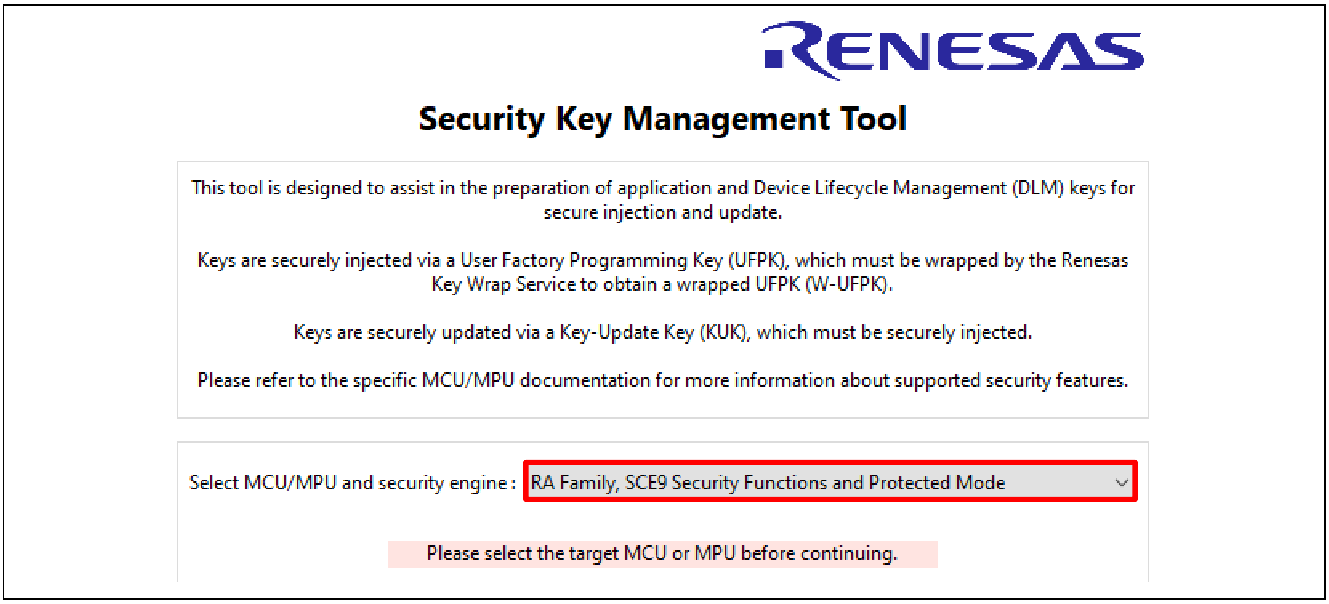


Figure 42. Select RA Family, SCE9 Protected Mode

- RA6M3 has SCE7, for the SCE7 example project included, in the **Overview** window, select **RA Family, SCE7**

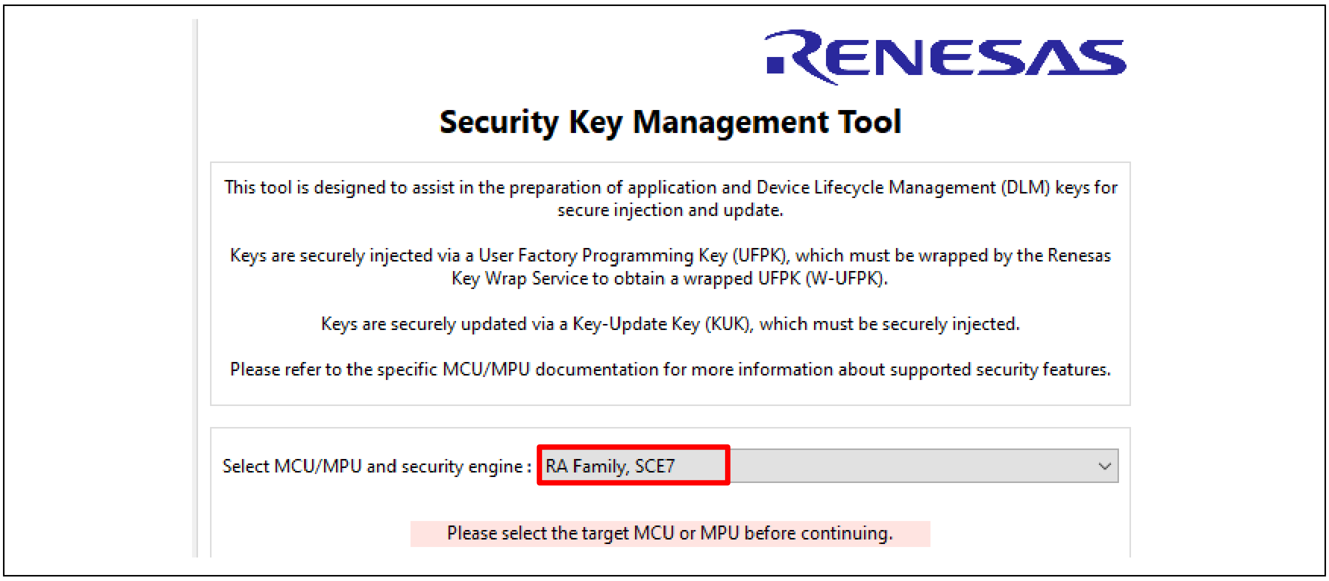


Figure 43. Select RA Family, SCE7

Once the correct MCU Family, Security Engine, and Mode are selected, navigate to the **Generate UFPK** page.

- For the **User Factory Programming Key**, select **Use specified value**.
- Click the **Browse** button to select a folder to store the key and name the resulting file.
  It is recommended that users choose different file names for the different MCU families to avoid confusion at the UFPK wrapping stage. In this example, we name the file `ra8x1_ufpk.key`.

  Click **Generate UFPK key file**. The `ra8x1_ufpk.key` file will be generated. Similarly, the UFPK for RA6M4, RA6M3 and RA8P1 can also be generated.

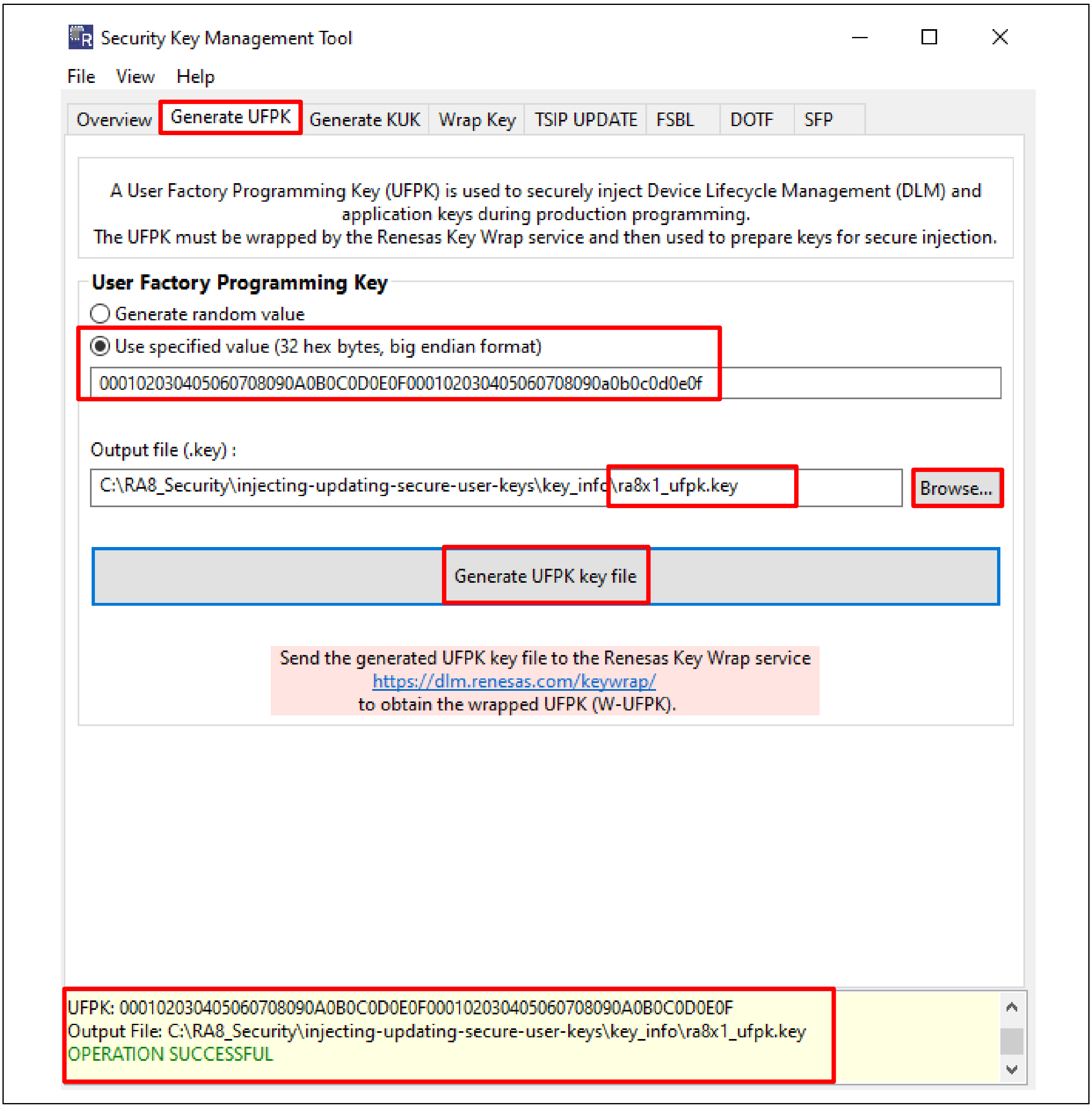


Figure 44. Generate Fixed UFPK using GUI for RSIP-E51A

Optionally, the user can also choose the **Generate random value** option to generate the UFPK.

4.3 Creating the User Factory Programming Key using the CLI Interface

Open a Command Prompt window and navigate to the folder where `skmt.exe` resides, typically under `\Renesas\Security Key Management Tool\CLI\`.

Use the following command to generate a random UFPK and place it in a key file (`ufpk.key`). If desired, a complete file name with a path may be specified. Refer to the Security Key Management Tool user's manual to understand the usage of `/genufpk` option.

```
skmt.exe /genufpk /output "C:\User_key_injection_protected_mode\keys\ufpk.key"
```

This command will generate a random 256-bit UFPK as shown below.

```
UFPK: E8AB23E99C9AD42823DA4215549A41496720F7243680A4715F4B944ACC94B691
Output File: C:\User_key_injection_protected_mode\keys\ufpk.key
```

Figure 45. Create a Random UFPK Using SKMT CLI

It is also possible to specify a specific UFPK, as shown by the following command:

```
skmt.exe /genufpk /ufpk
"000102030405060708090A0B0C0D0E0F000102030405060708090a0b0c0d0e0f" /output
"C:\User_key_injection_protected_mode\keys\ufpk.key"
```

```
UFPK: 000102030405060708090A0A0C0D0E0F000102030405060708090a0b0c0d0e0f
Output File: C:\User_key_injection_protected_mode\keys\ufpk.key
```

Figure 46. Create a Fixed UFPK Using SKMT CLI

4.4 Wrapping the UFPK

The next step is to obtain a W-UFPK from the Renesas Key Wrap Service based on the selected UFPK. Note that if the UFPK is changed, a new W-UFPK must be obtained.

1. Launch the Kleopatra program.
2. Encrypt the UFPK with the Renesas public key. This key was imported earlier to Kleopatra. Using Kleopatra, select **Sign/Encrypt…** and select the UFPK file. In this screenshot, an example file named `ufpk_ra6m3.key` file is used for demonstration purposes. Then click **Open**.

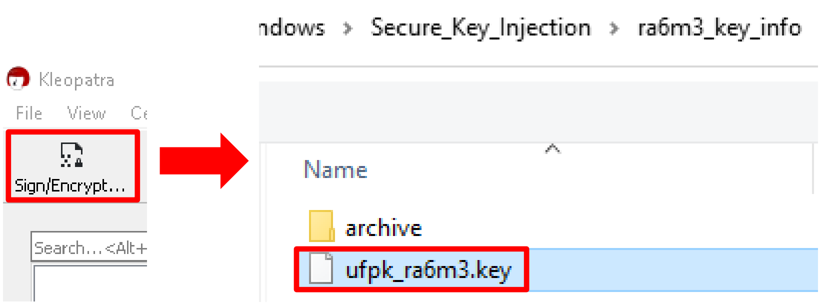


Figure 47. Encrypt the UFPK File for PGP Transfer

3. When asked which entity this file is to be encrypted for, (optionally) uncheck **Encrypt for me** and check **Sign as, Encrypt for others**, and **Encrypt / Sign each file separately**.

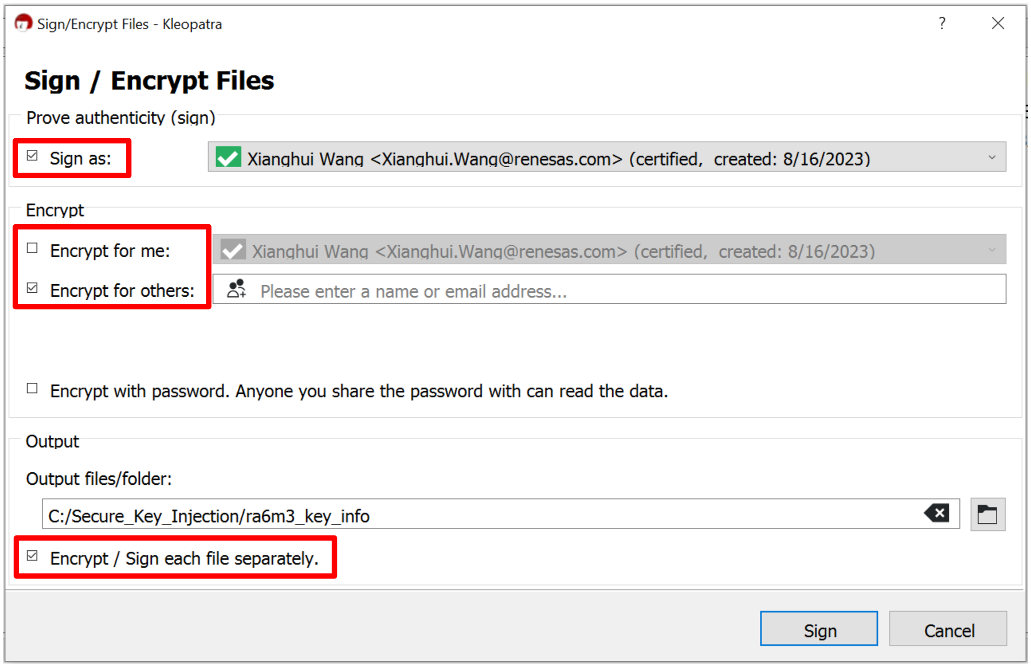


Figure 48. Select PGP Encryption Options

4. Click the **Open Selection Dialog** (the icon). This will open a **Certificate Selection** dialog box.

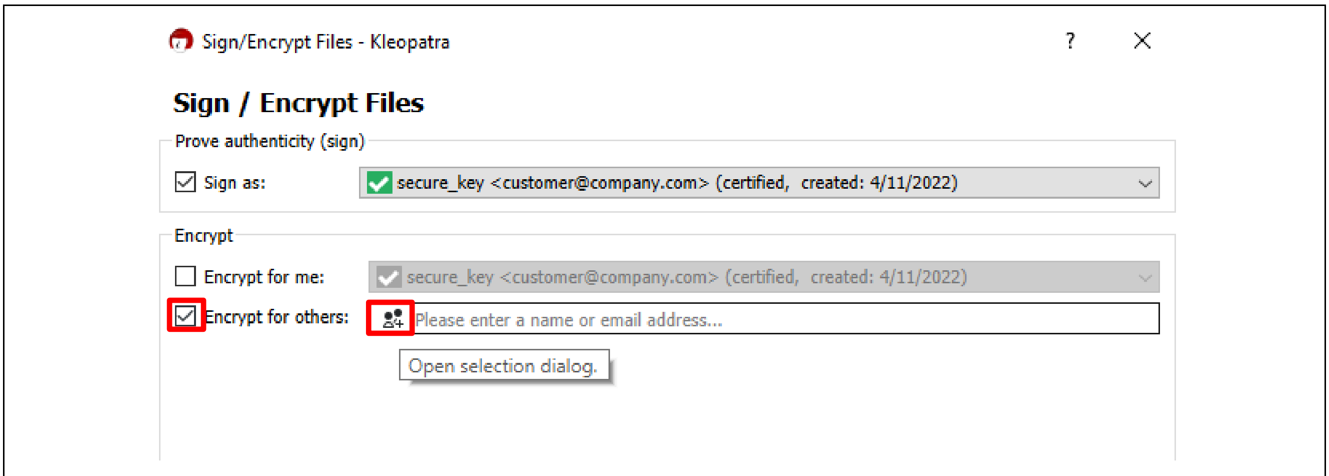


Figure 49. Open the Selection Dialog

5. In this window, select **keywrap** to select the Renesas public key, then click **OK**

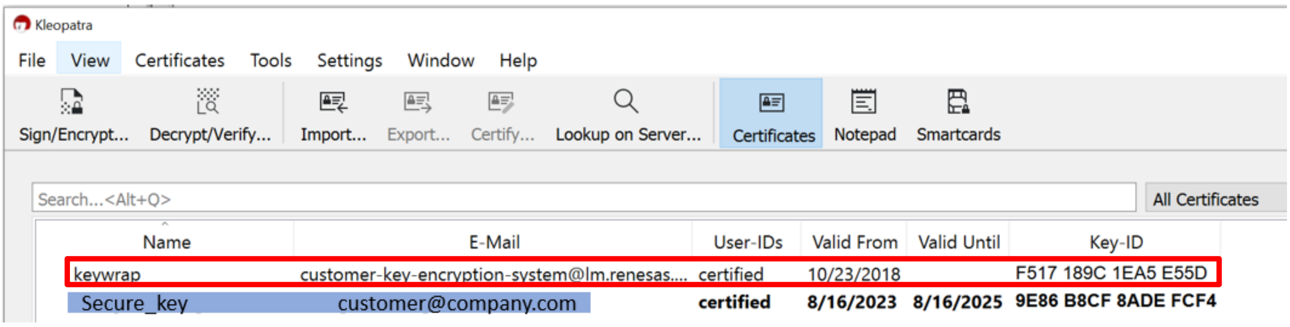


Figure 50. Select the Renesas PGP Public key

6. Ensure that the correct destination folder for the encrypted key is selected under **Output.** Finally, click **Sign/Encrypt**. It is a good practice to keep UFPK and W-WUPK for different MCU families in different folders and under different names.

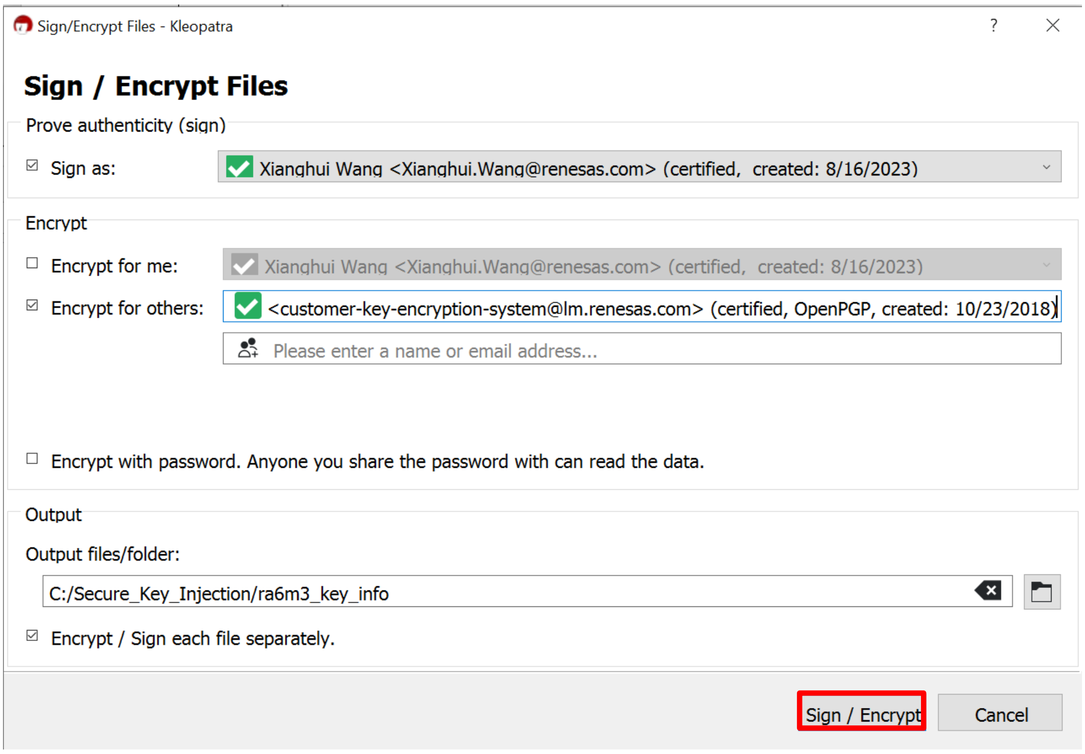


Figure 51. Encrypt UFPK using Renesas PGP Public Key

7. If you do not check **Encrypt for me**, you will get an **Encrypt-To-Self Warning** that you cannot decrypt the data. Click **Continue**.

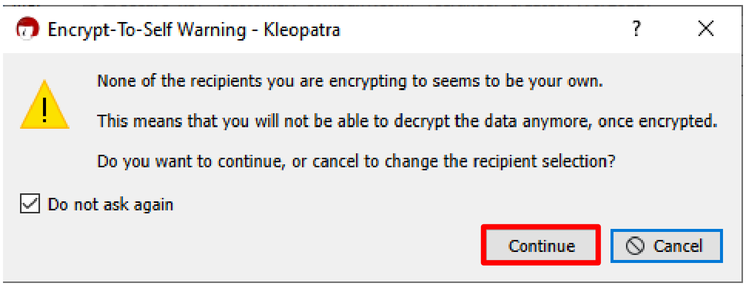


Figure 52. Start the UFPK Encryption process

8. Provide your private key passphrase, then click **OK**.

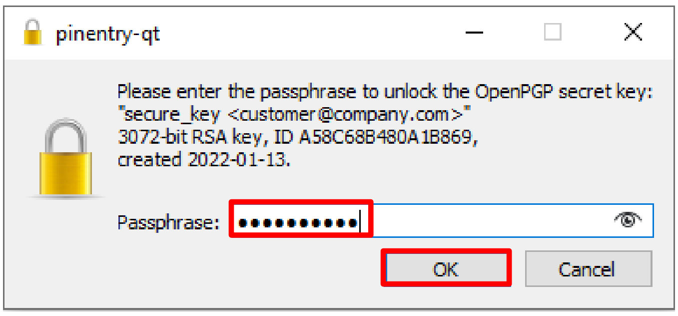


Figure 53. Provide Passphrase

9. The UFPK encrypted with the Renesas public key will be generated, with the `.gpg` added to the extension of the key. In this case, the file `ufpk_ra6m3.key.gpg` is generated. Click **Finish**.

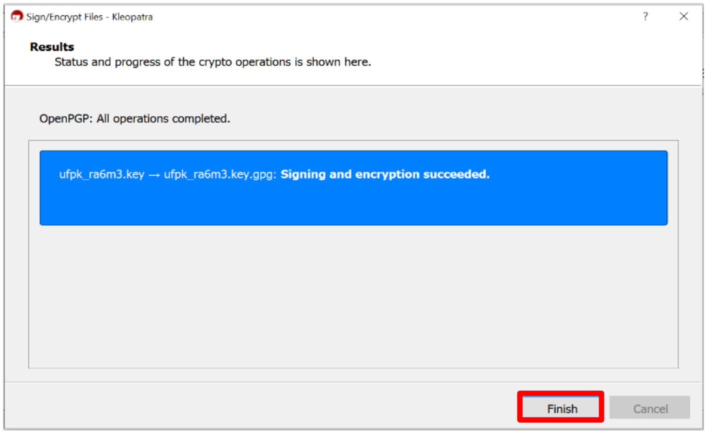


Figure 54. Encrypted Key is Generated

10. Now, we can send the UFPK that has been encrypted with Renesas Public Key to the Renesas DLM Server for wrapping. Return to the DLM Server web page:

Figure 55. Select the MCU Family

When generating the Wrapped UFPK, it is important to select the correct MCU family and security engine mode.

Note: A W-UFPK generated for Compatibility Mode cannot be used to inject keys in DLM or Protected Mode, and vice versa.

- To create a W-UFPK for the RA8P1 Protected Mode secure key injection example project, select the **Renesas RA** Family and click Protected Mode **RA8P1 Encryption of customer's data**.

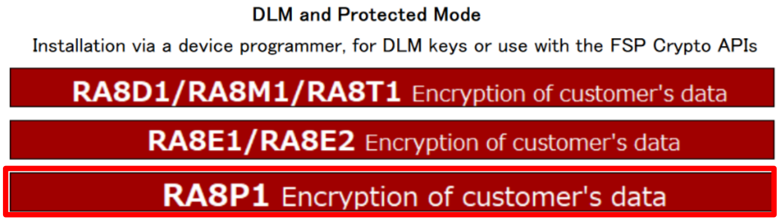


Figure 56. Select the RA8P1 MCU Group DLM and Protected Mode

- To create a W-UFPK for the RA8M1 Compatible Mode secure key injection example project, select the **Renesas RA** Family and click Compatibility Mode **RA8D1/RA8M1/RA8T1 Encryption of customer's data**.

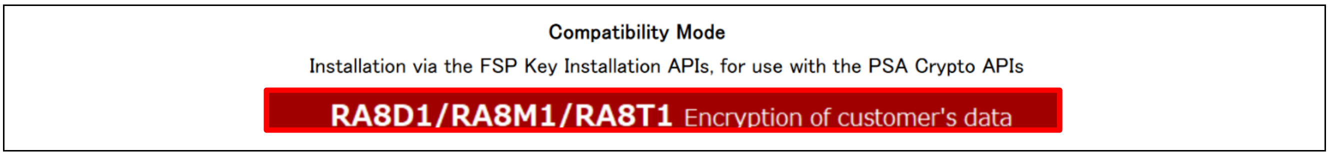


Figure 57. Select the RA8M1 MCU Group Compatibility Mode

- To create a W-UFPK for the RA6M4 example project, select the **Renesas RA** Family and click Protected Mode **RA6M4/RA6M5 Encryption of customer's data**.

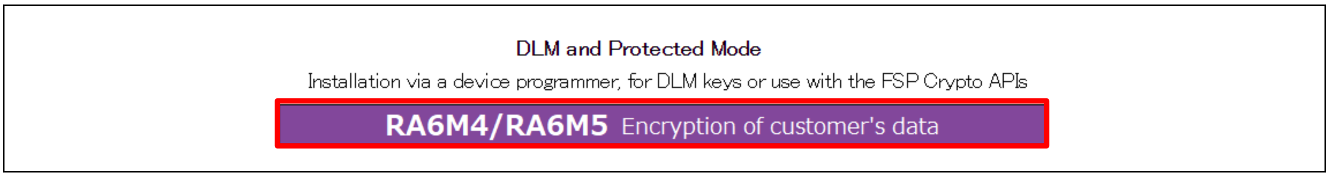


Figure 58. Select the RA6M4/RA6M5 MCU Group DLM and Protected Mode

- To create a W-UFPK for the RA6M3 example project, select the **Renesas RA** Family and click Compatibility Mode **RA6M1/RA6M2/RA6M3/RA6T1 Encryption of customer's data**.

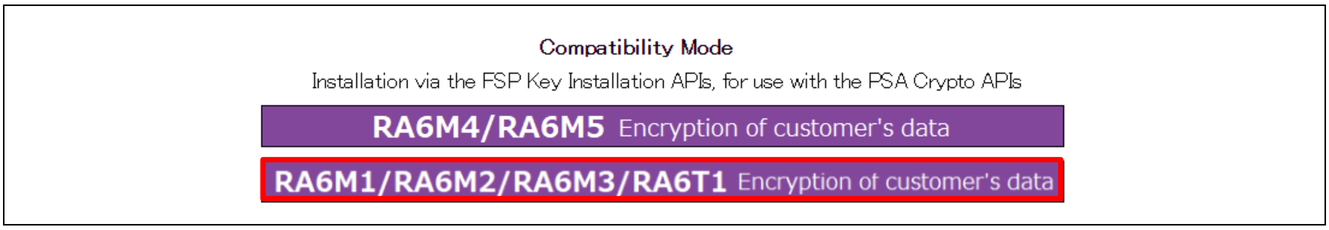


Figure 59. Select the RA6M1/RA6M2/RA6M3/RA6T1 MCU Group Compatibility Mode

11. Click **Encryption service for products** on the next screen. Here, the screenshot uses RA6M3 as an example; for other MCU families, a similar screen will be presented.

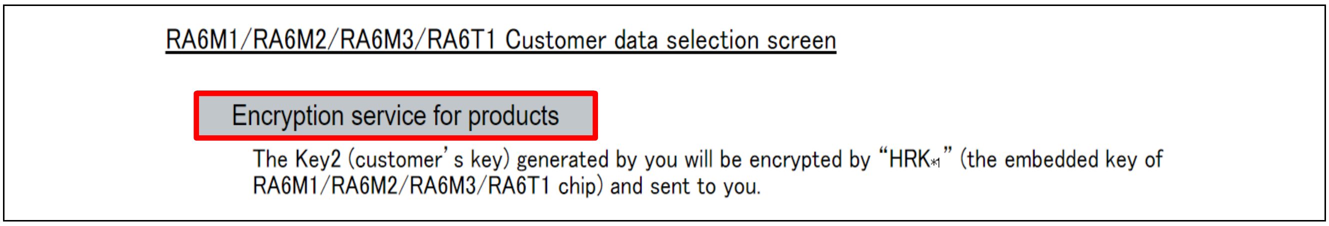


Figure 60. Choose Encryption service for products

12. Click **Reference** and select the corresponding encrypted UFPK; example shown is `ufpk_ra6m3.key.gpg` created previously and click **Open**. Note that in the DLM server description, **Key2** refers to the UFPK.

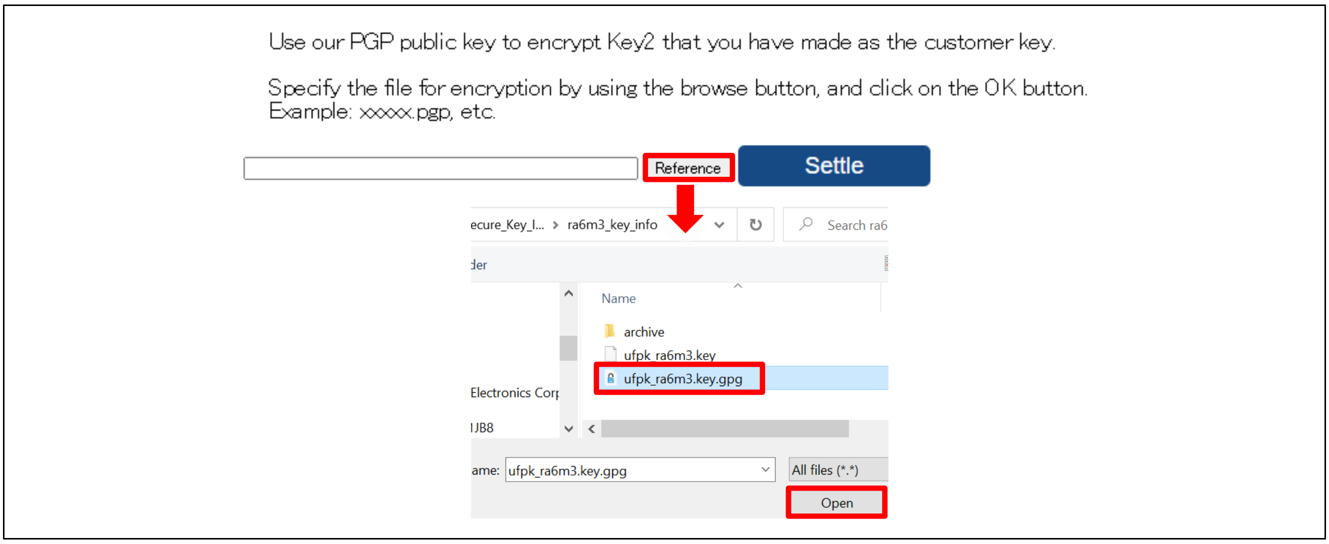


Figure 61. Select the PGP-Encrypted UFPK file

13. Click **Settle**. The following message will be printed. Then click **Return to the menu**. You can now log out of the Renesas Key Wrap Service.

Figure 62. Return to the DLM Server Main Menu

14. The wrapped UFPK Key (W-UFPK) encrypted with your PGP public key should arrive in your email typically in about 1-2 minutes. Save the attached file.

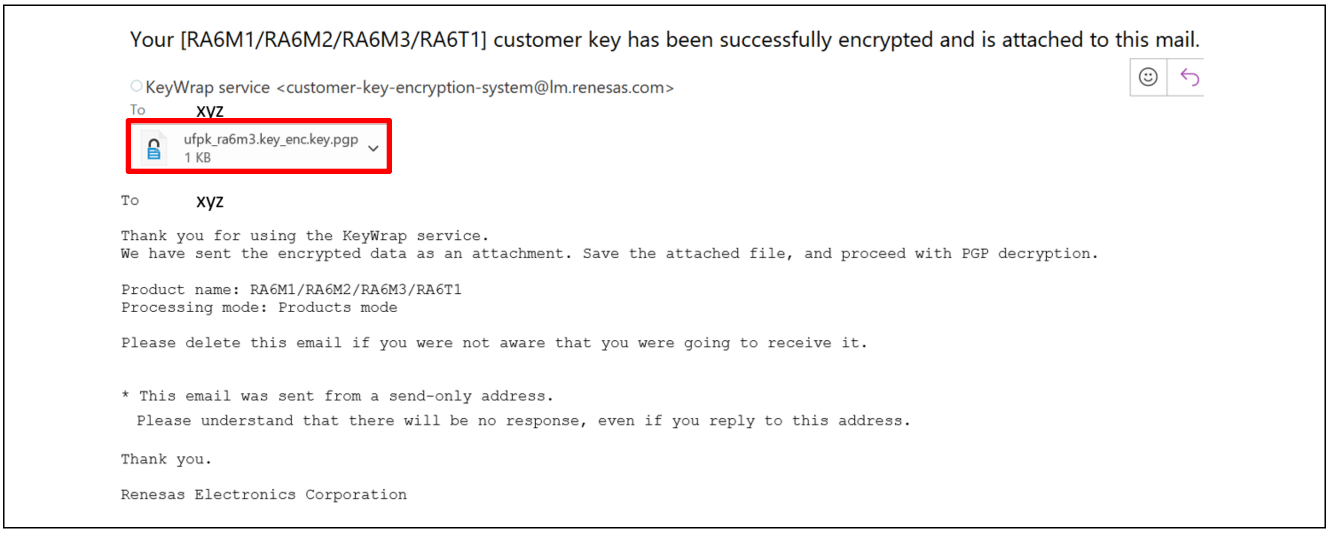


Figure 63. Receiving the W-UFPK via Email

15. With the Kleopatra program, click **Decrypt/Verify,** select the W-UFPK file, and click **Open**.

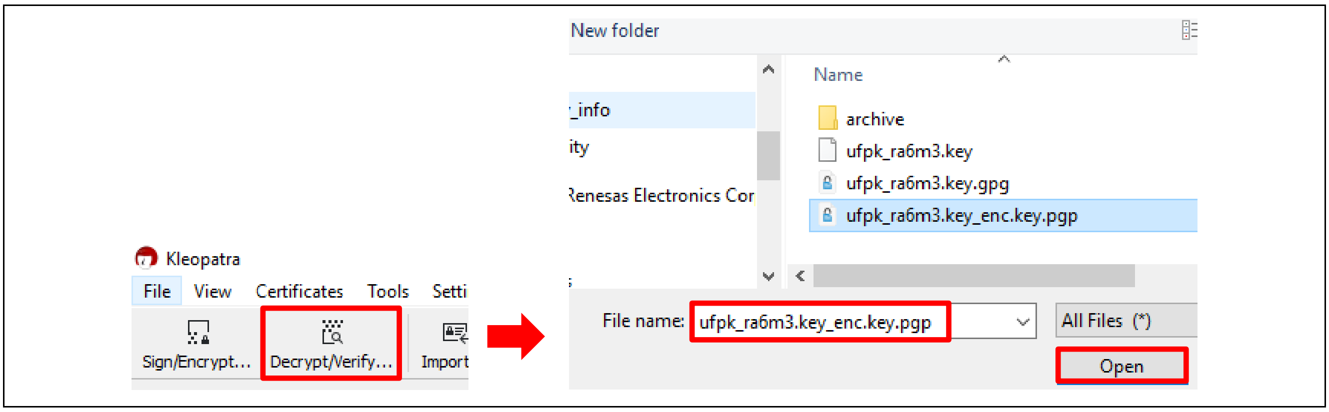


Figure 64. Decrypt the W-UFPK

16. Follow the prompt to provide your PGP private key passphrase and click **OK.** The decrypted W-UFPK is generated in the folder specified.

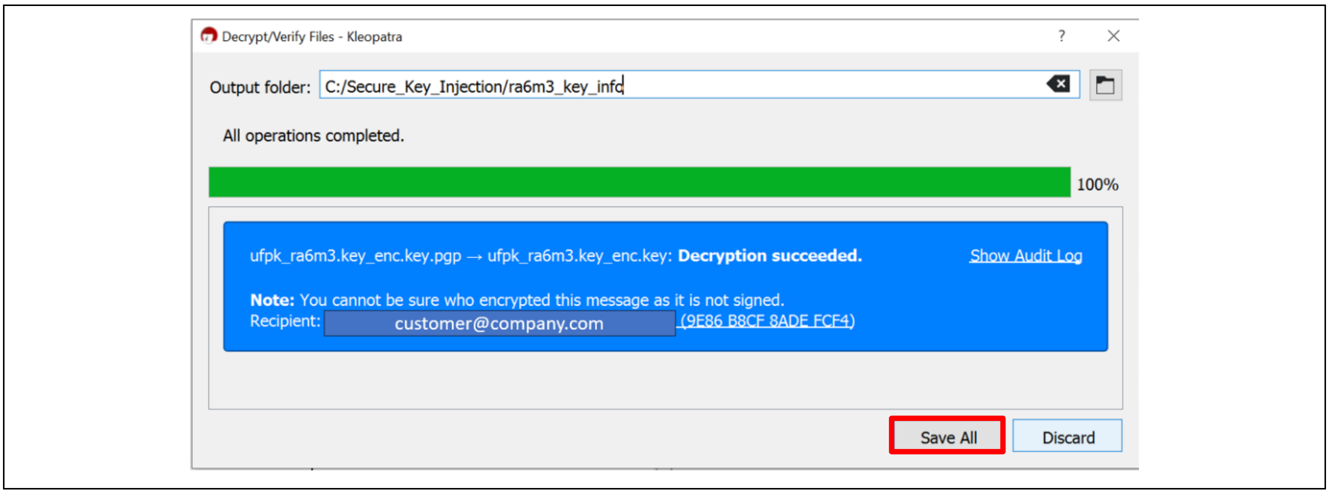


Figure 65. Decrypting the Encrypted W-UFPK

17. Click **Save All** to save the decrypted W-UFPK key file `ufpk_ra6m3.key_enc.key` to the same folder as the UFPK key file. Both key files are required to generate key injection bundles.

5. Secure Key Injection for RSIP and SCE9 Protected Mode

This section walks the user through the wrapping process required for secure key injection and update. The SKMT tool is used to perform this key-wrapping process.

Step-by-step instructions for generating the three types of keys are provided using both the CLI and GUI interfaces of the SKMT.

- **User Key** wrapping with the UFPK for secure key injection of the user key
- **Key-Update Key** wrapping with the UFPK for secure key injection of the KUK
- **User Key** wrapping with the KUK for secure key update of the user key

This application project provides examples of user key wrapping of both AES-256 and ECC secp256r1 public keys for SCE9 Protected Mode on RA6M4 MCU, and also provides an example of user key wrapping an ECC secp256r1 key pair (both private key and public keys) for RSIP-E50D Protected Mode on RA8P1 MCU.

5.1 Wrap Keys with the UFPK and W-UFPK for SCE9 Protected Mode using the SKMT GUI Interface

To prepare a Protected Mode user key to inject using RFP, we need the UFPK, W-UFPK, and the user key as input to the SKMT GUI interface.

Launch the SKMT GUI and select **RA Family, SCE9 Security Functions, and Protected Mode** on the **Overview** tab. On the **Wrap Key** tab, open the submenu **Key Type**. This page can be used to choose which key type to prepare.

5.1.1 Wrap an Initial AES-256 Key with the UFPK

A NIST CAVP test vector is used for this purpose.

https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Block-Ciphers

```
KEY = 8000000000000000000000000000000000000000000000000000000000000000
IV = 00000000000000000000000000000000
PLAINTEXT = 00000000000000000000000000000000
CIPHERTEXT = e35a6dcb19b201a01ebcfa8aa22b5759
```

Figure 66. NIST AES 256 Test Vector

In the **Key Type** area, choose **Key Type** and specify **AES** with **256 bits**.

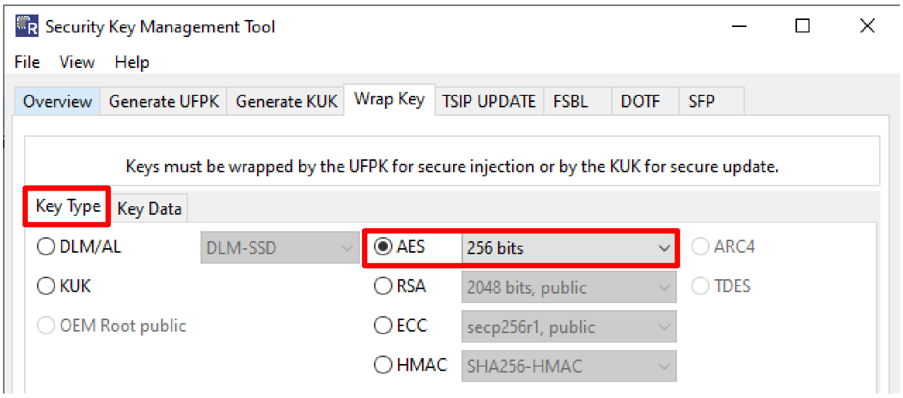


Figure 67. Choose AES 256 bits as the Key Type

Navigate to the **Key Data** page and input the **Raw** key data as shown below based on the NIST vector shown in Figure 66. The key data is duplicated here to easily copy and paste to the GUI interface.

KEY = 8000000000000000000000000000000000000000000000000000000000000000

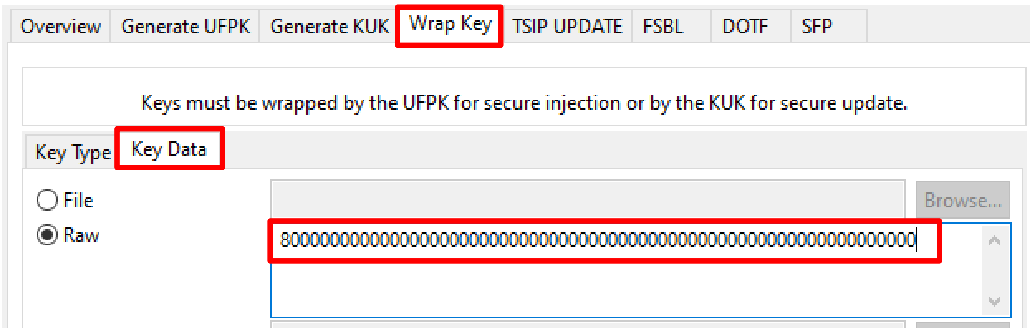


Figure 68. Set up the Key Data

Under the **Wrapping Key** section, click the corresponding **Browse** buttons to select the **UFPK** and **W-UFPK** key pair that you generated for RA6M4 created in section 4.2 and 4.4. For the **IV**, select **Generate random value**. In the **Output** option, select **RFP;** then click the **Browse** button, choose the output folder, and name the output file.

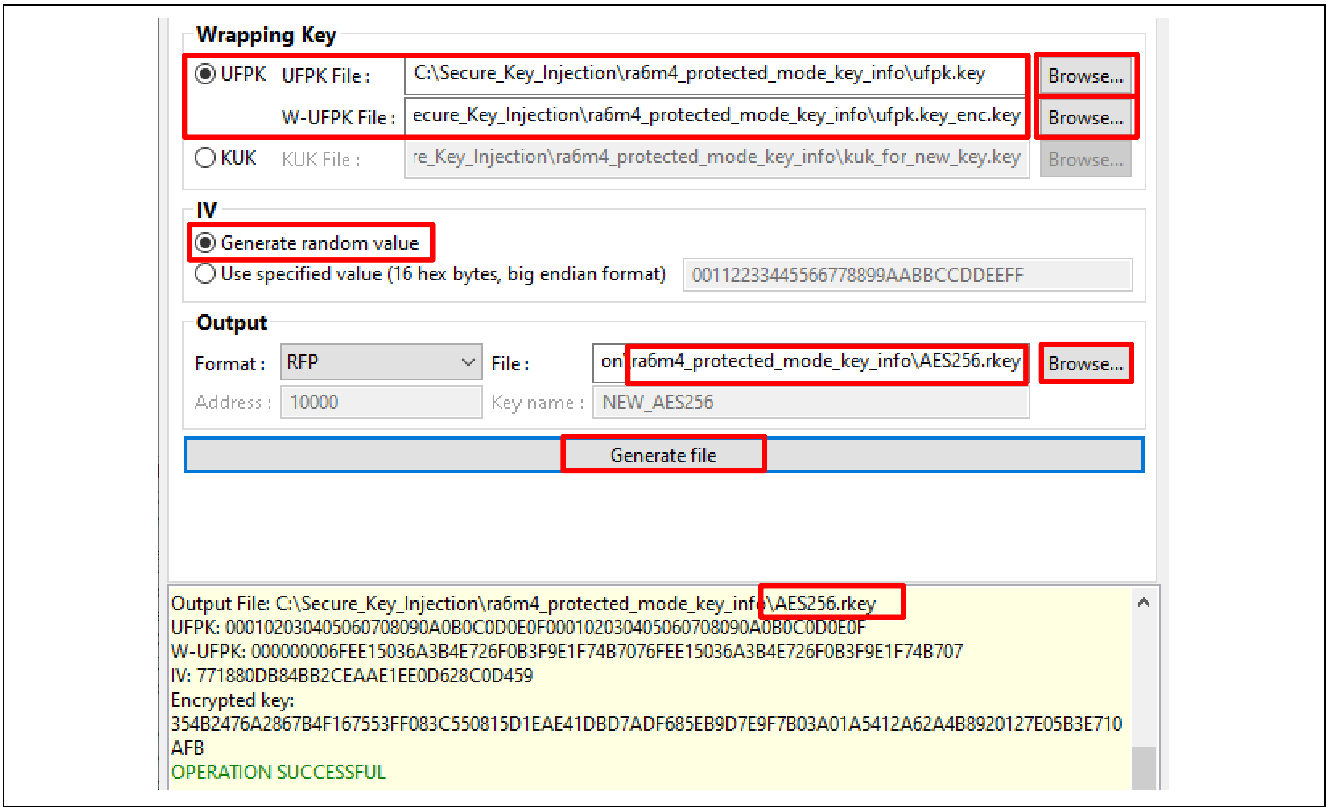


Figure 69. Generate the AES 256 RFP Injection Key File

Now click **Generate File**. The `AES256.rkey` file will be generated.

The plaintext AES-256 key and UFPK are NOT contained in the `*.rkey` file, enabling confidential transfer of the key injection file contents.

5.1.2 Wrap an Initial ECC Public Key with the UFPK

A set of NIST test vectors are used in this application project. The CAVP NIST test vectors can be downloaded from the following link. The ECDSA vectors are what we will use.

[Cryptographic Algorithm Validation Program | CSRC (nist.gov)]

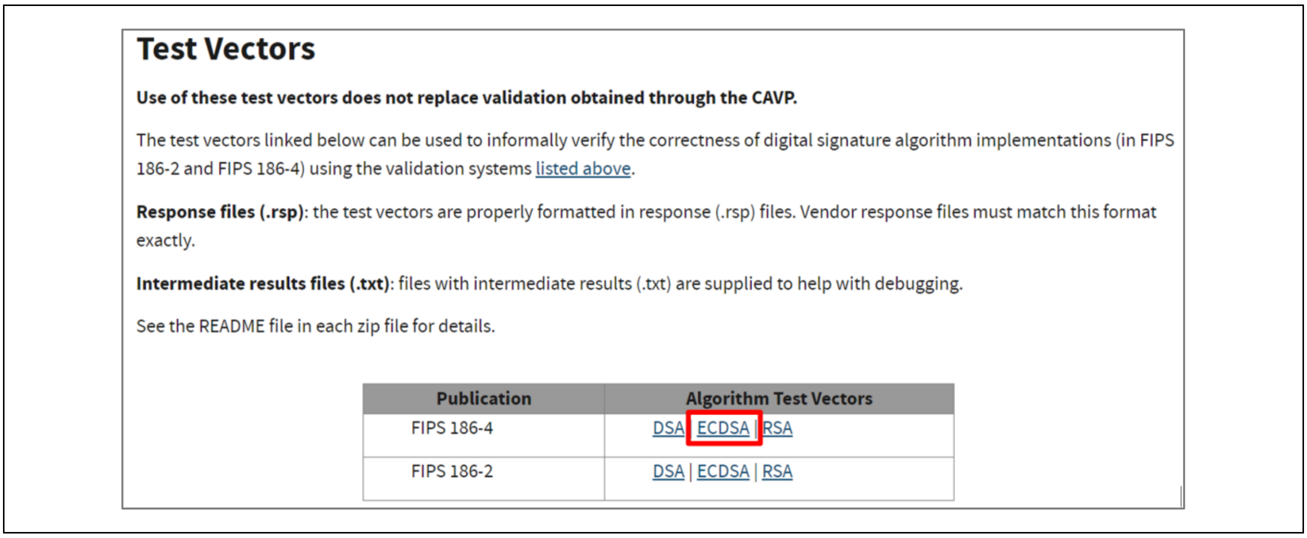


Figure 70. ECDSA Test Vectors

After downloading the zip file 186-4ecdsatestvectors.zip, unzip it. The following vectors can be found in the plaintext file SigGen.txt.

```
Msg =
5905238877c77421f73e43ee3da6f2d9e2ccad5fc942dcec0cbd25482935faaf416983fe165b1a045ee2bcd2
e6dca3bdf46c4310a7461f9a37960ca672d3feb5473e253605fb1ddfd28065b53cb5858a8ad28175bf9bd386
a5e471ea7a65c17cc934a9d791e91491eb3754d03799790fe2d308d16146d5c9b0d0debd97d79ce8
d = 519b423d715f8b581f4fa8ee59f4771a5b44c8130b4e3eacca54a56dda72b464
Qx = 1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83
Qy = ce4014c68811f9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9
k = 94a1bbb14b906a61a280f245f9e93c7f3b4a6247824f5d33b9670787642a68de
R = f3ac8061b514795b8843e3d6629527ed2afd6b1f6a555a7acabb5e6f79c8c2ac
S = 8bf77819ca05a6b2786c76262bf7371cef97b218e96f175a3ccdda2acc058903
```

Figure 71. NIST ECC secp256r1 Test Vector

Launch the SKMT GUI and select **RA Family, SCE9 Security Functions, and Protected Mode** on the **Overview** tab. On the **Wrap Key** tab, select the **Key Type** as **ECC** and **secp256r1, public,** as shown in Figure 72.

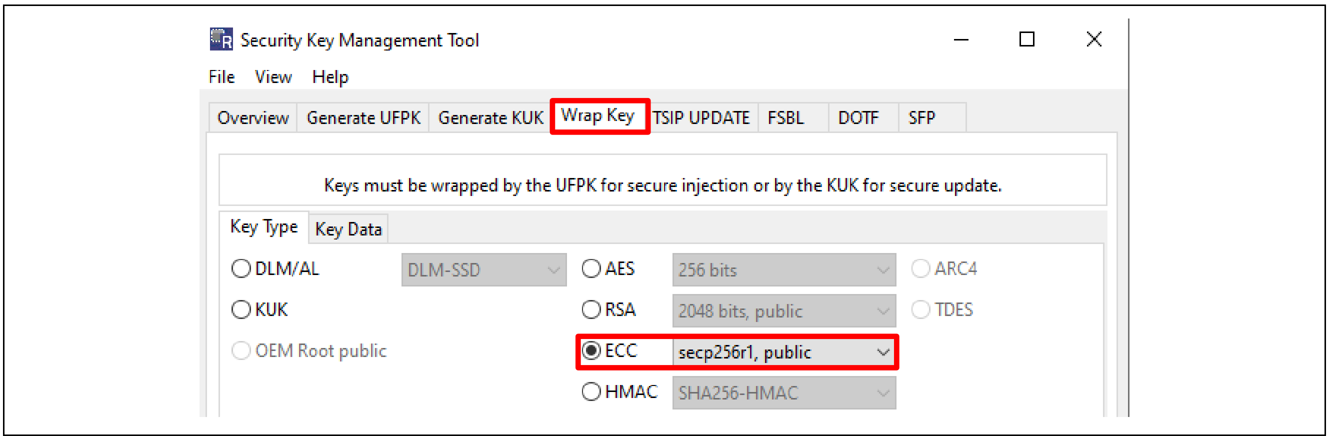


Figure 72. Choose secp256r1 Public Key

Next, configure the **Key Data**. Under the **Key Data** area, select **Raw** and provide the **Qx** and **Qy** as shown below. The key data is duplicated here to easily copy and paste to the GUI interface.

Qx = 1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83

Qy = ce4014c68811f9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9

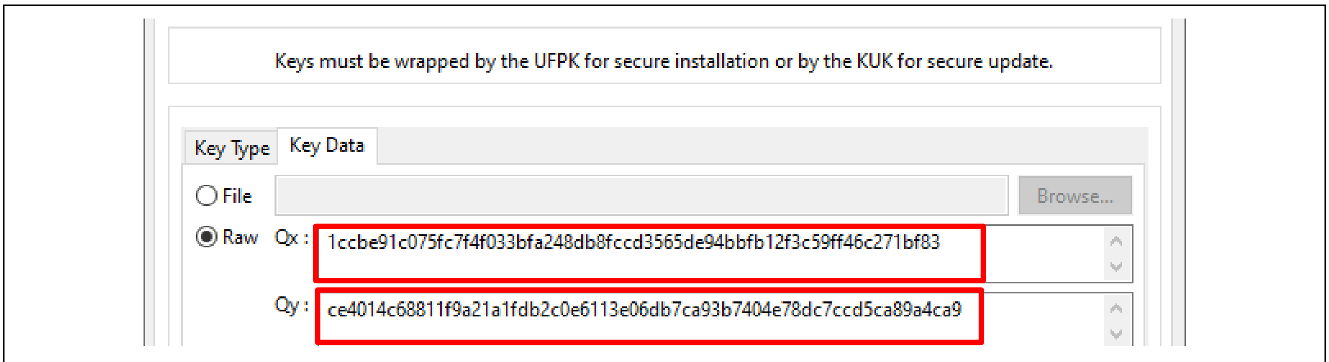


Figure 73. Provide the ECC Public Key data

Next, under the **Wrapping Key** section, click the corresponding **Browse** buttons to select the **UFPK** and **W-UFPK** key pair created in section 4.2 and 4.4. For the **IV**, select **Generate random value**. For the **Output** option, select **RFP**; then click the **Browse** button, choose the output folder, and name the output file.

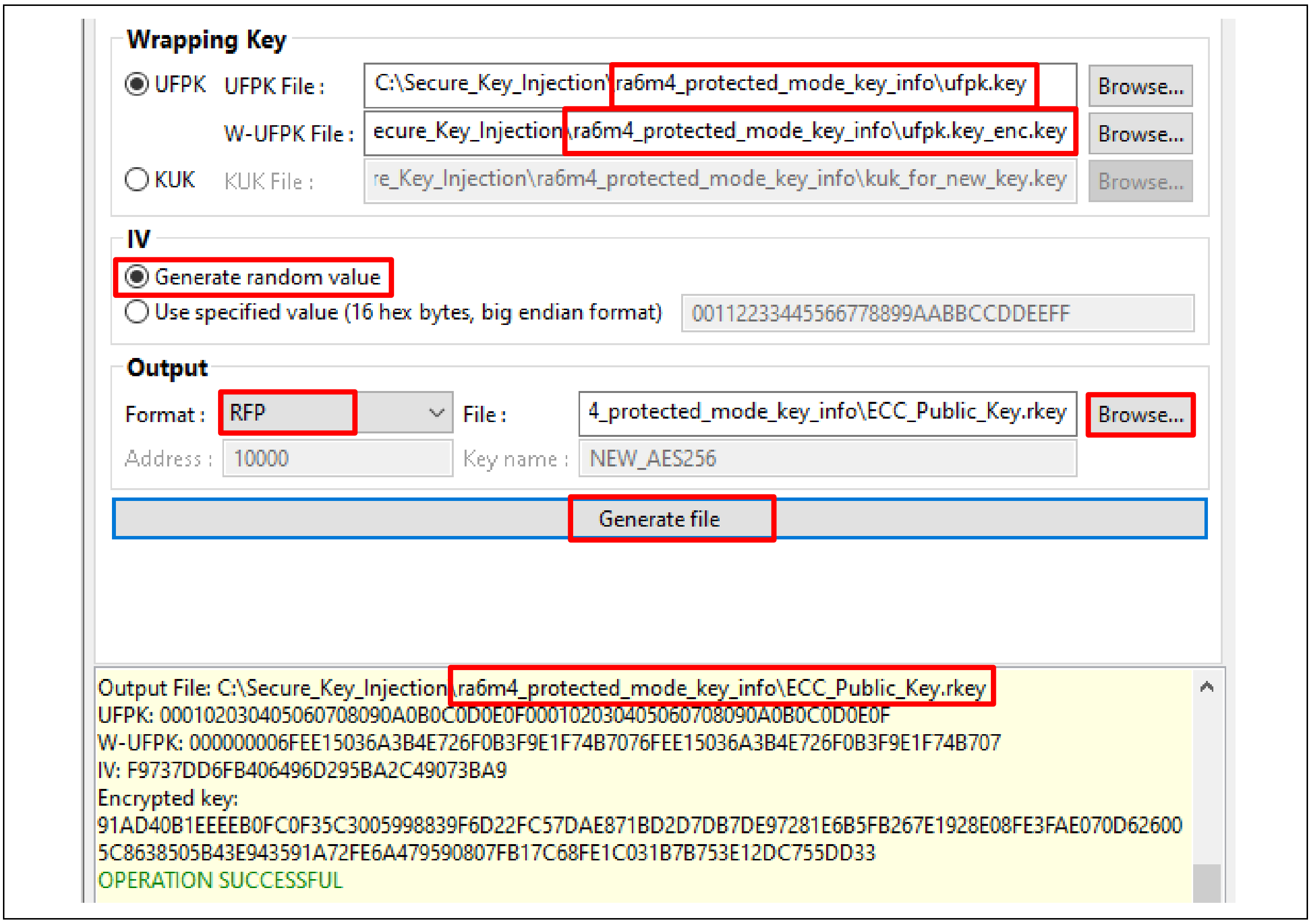


Figure 74. Generate the ECC Public Key RFP Injection Key File using GUI

The plaintext ECC key and UFPK are NOT contained in the `*.rkey` file, enabling confidential transfer of the key injection file contents.

5.1.3 Wrap a Key-Update Key with the UFPK

The SKMT can be used to generate a sample KUK. To generate the KUK key file, navigate to the **Generate KUK** tab and use: `000102030405060708090a0b0c0d0e0f000102030405060708090a0b0c0d0e0f`.

Click the **Browse** button to select the folder and file name for the generated key file, here specified as `kuk_for_new_key.key`. Next, click **Generate KUK key file**, and the `kuk_for_new_key.key` file will be generated in the selected folder.

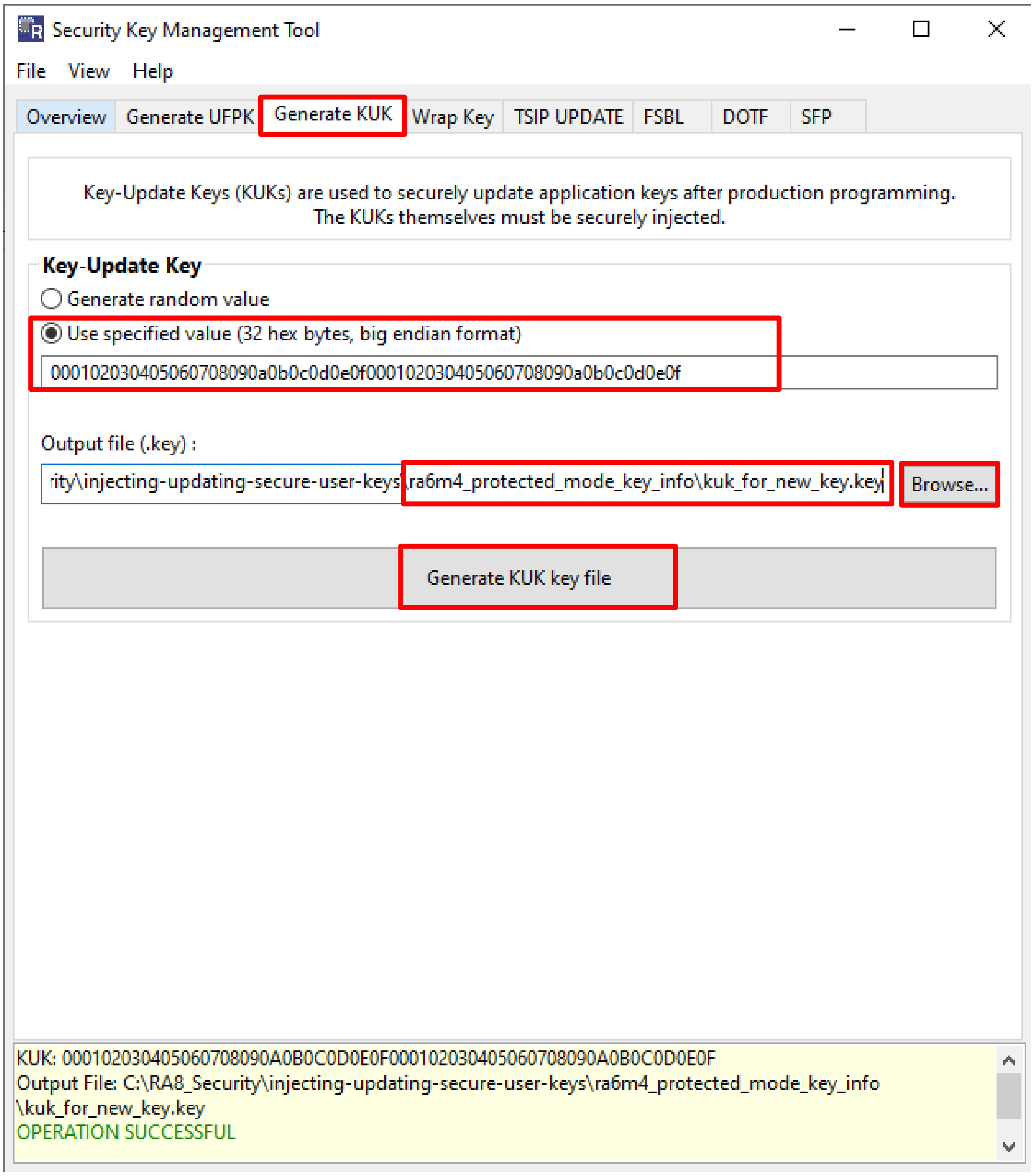


Figure 75. Generate the KUK File used to Encrypt the User Key for SCE9

Next, we will wrap the KUK so it can be injected into the MCU. Navigate to the **Wrap Key** page and choose **KUK** from the **Key Type** area.

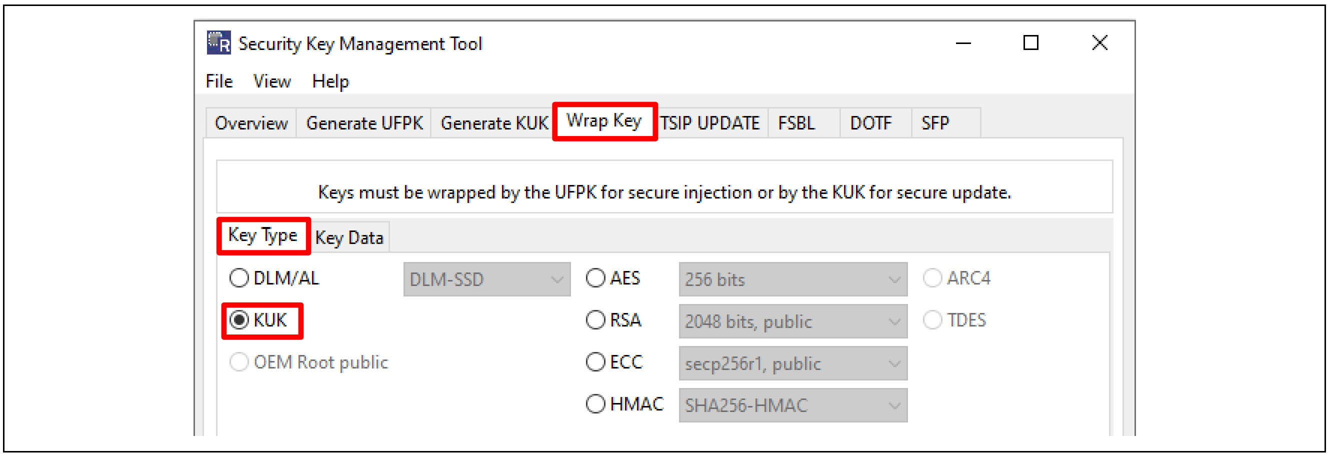


Figure 76. Choose KUK to Wrap

Navigate to the **Key Data** page, select the **File** option, and browse to the `kuk_for_new_key.key` key file generated in Figure 75.

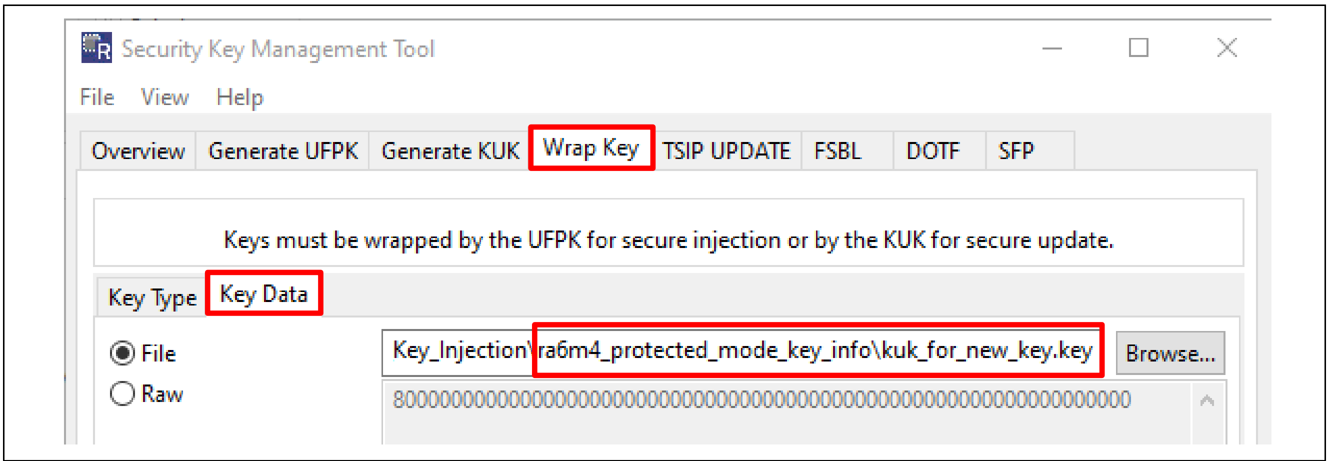


Figure 77. Provide the KUK .key File

Next, under the **Wrapping Key** section, click the corresponding **Browse** buttons to select the **UFPK** and **W-UFPK** key pair created in section 4.2 and 4.4. For the **IV**, select **Generate random value**. For the **Output** option, select **RFP**; then click the **Browse** button, choose the output folder, and name the output file.

Now click the **Generate File** button. The `KUK.rkey` file will be generated.

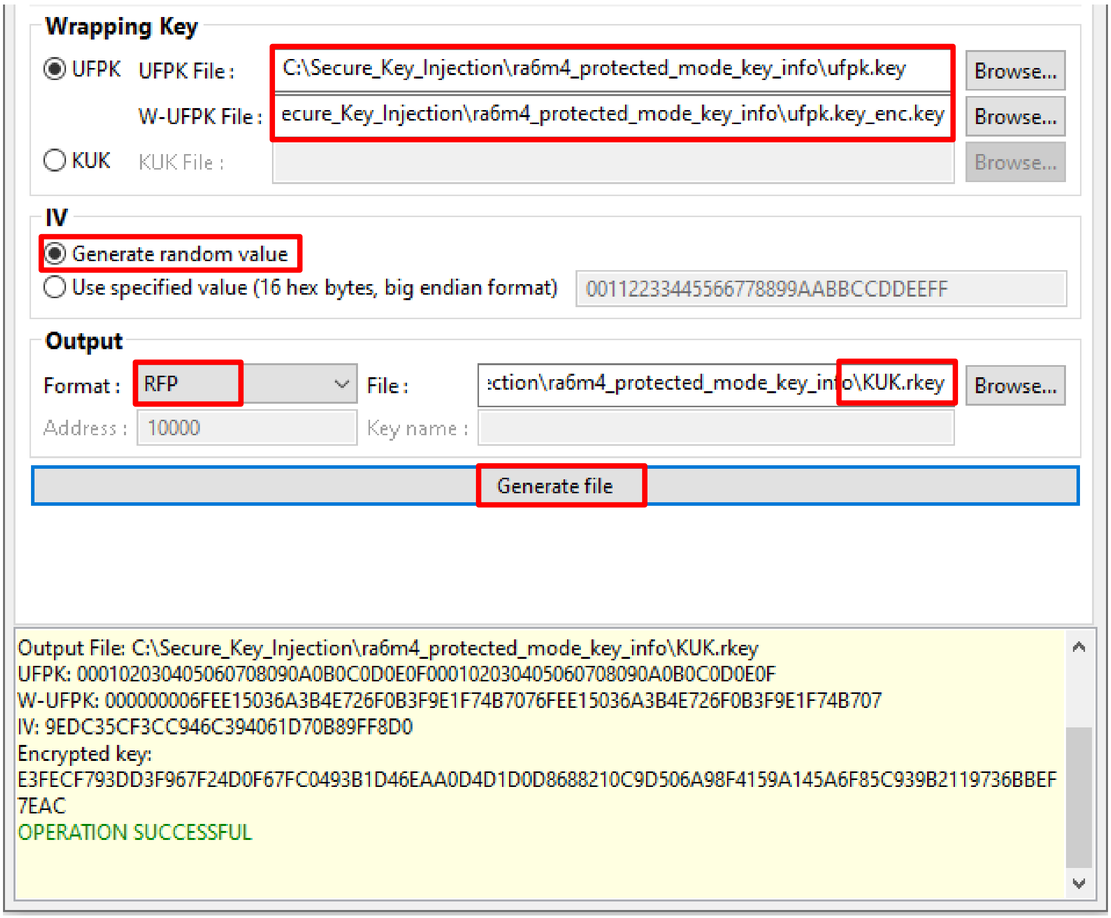


Figure 78. Generate the Key-Update Key Injection File using GUI for SCE9

The plaintext KUK and UFPK are NOT contained in the *.rkey file, enabling confidential transfer of the key injection file contents.

5.1.4 Wrap a New AES-256 User Key with the KUK

In the section, we will use the `kuk_for_new_key.key` generated in Figure 75 to wrap a new AES-256 key.

We will use a second NIST test vector to demonstrate secure key updates using the KUK.

```
KEY = c000000000000000000000000000000000000000000000000000000000000000
IV = 00000000000000000000000000000000
PLAINTEXT = 00000000000000000000000000000000
CIPHERTEXT = b29169cdcf2d83e838125a12ee6aa400
```

Figure 79. NIST Test Vector as New AES-256 Key Test Data

Navigate to the SKMT **Wrap Key** tab. In the **Key Type** area, select **AES-256** with **256 bits**.

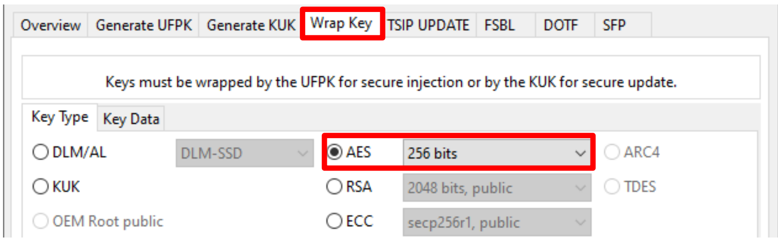


Figure 80. Choose AES 256bit New User Key

In the **Key Data** area, provide the key data from the NIST vector based on Figure 79. The key data is duplicated here to copy and paste into the GUI interface.

KEY = c000000000000000000000000000000000000000000000000000000000000000

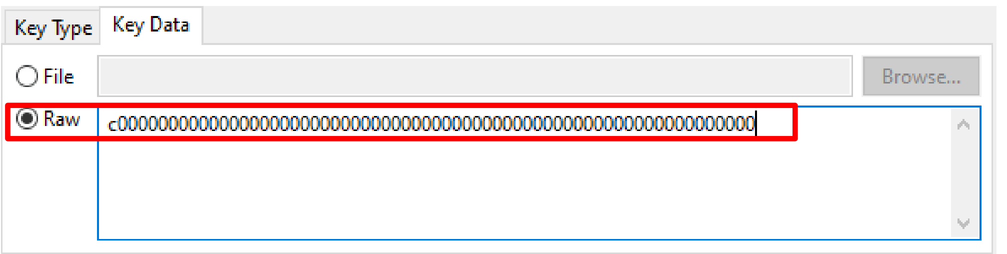


Figure 81. Provide the New AES 256-bit Key Data

In the **Wrapping Key** area, select **KUK** as the wrapping key and click **Browse** to locate the `kuk_for_new_key.key` file generated in Figure 75. For the **IV**, choose **Generate random value**. For the **Output** option, choose **C Source** and name the output file as `new_aes_key.c`. Name the **Key name** property as **NEW_AES256**. This name will be used in the source files for key-specific definitions.

Finally, click **Generate file**. Both the `new_aes_key.c` and the `new_aes_key.h` files will be generated.

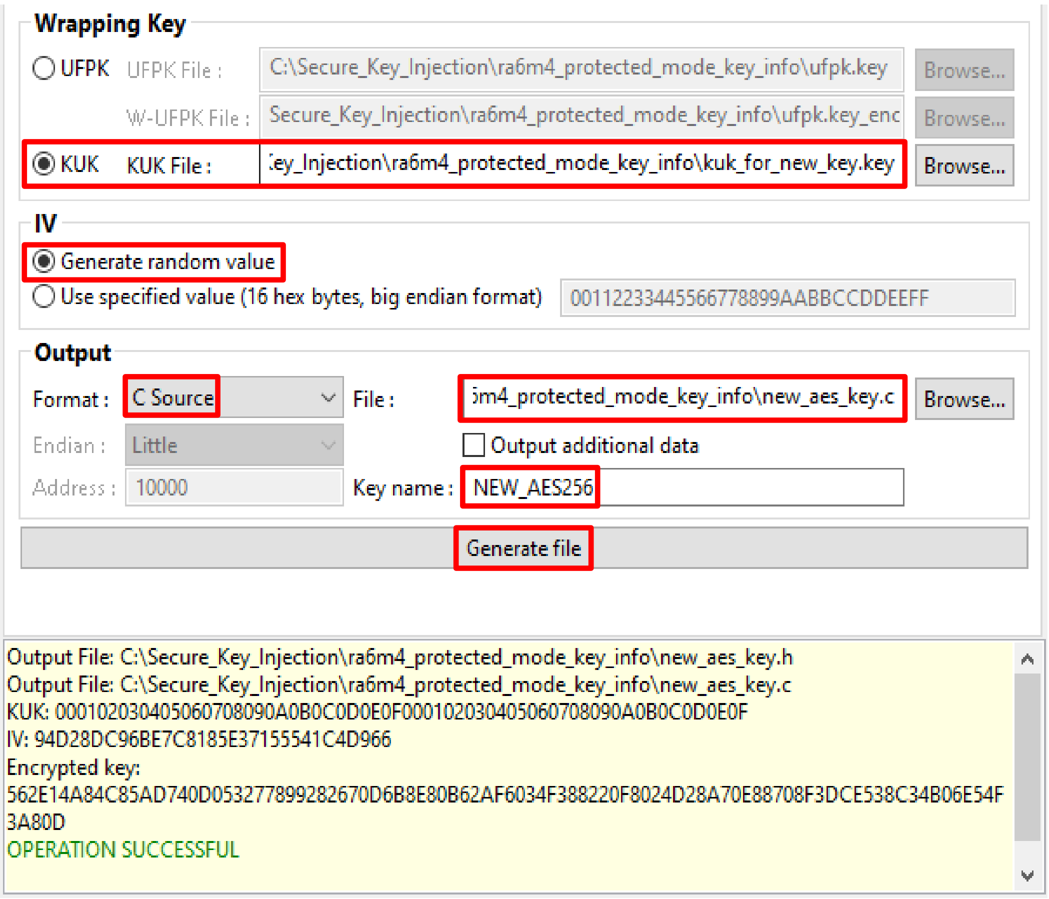


Figure 82. Generate KUK-Wrapped AES-256 Key

5.1.5 Wrap a New ECC Public Key with the KUK

In the section, we will use the `kuk_for_new_key.key` generated in Figure 75 to wrap a new ECC Public key.

To demonstrate updating the ECC public key, another NIST ECC secp256r1 test vector is used in this application project. This test vector can be found in SigGen.txt, downloaded based on Figure 70.

```
Msg =
c35e2f092553c55772926bdbe87c9796827d17024dbb9233a545366e2e5987dd344deb72df987144b8c6c43b
c41b654b94cc856e16b96d7a821c8ec039b503e3d86728c494a967d83011a0e090b5d54cd47f4e366c0912bc
808fbb2ea96efac88fb3ebec9342738e225f7c7c2b011ce375b56621a20642b4d36e060db4524af1
d = 0f56db78ca460b055c500064824bed999a25aaf48ebb519ac201537b85479813
Qx = e266ddfdc12668db30d4ca3e8f7749432c416044f2d2b8c10bf3d4012aeffa8a
Qy = bfa86404a2e9ffe67d47c587ef7a97a7f456b863b4d02cfc6928973ab5b1cb39
k = 6d3e71882c3b83b156bb14e0ab184aa9fb728068d3ae9fac421187ae0b2f34c6
R = 976d3a4e9d23326dc0baa9fa560b7c4e53f42864f508483a6473b6a11079b2db
S = 1b766e9ceb71ba6c01dcd46e0af462cd4cfa652ae5017d4555b8eeefe36e1932
```

Figure 83. New Set of NIST ECC Test Vectors

Follow the procedure below to wrap the new ECC public key using the KUK file generated in Figure 75.

From the SKMT GUI, make sure **RA Family, SCE9 Security Functions, and Protected Mode are** selected from the **Overview** page. Next, navigate to the **Wrap Key** page. Select the **Key Type** as **secp256r1, public** as shown in Figure 72.

Under the **Key Data** area, select **Raw** and provide **Qx** and **Qy** as shown below. The key data is duplicated here so the user can copy and paste it to the GUI interface.

Qx = e266ddfdc12668db30d4ca3e8f7749432c416044f2d2b8c10bf3d4012aeffa8a

Qy = bfa86404a2e9ffe67d47c587ef7a97a7f456b863b4d02cfc6928973ab5b1cb39

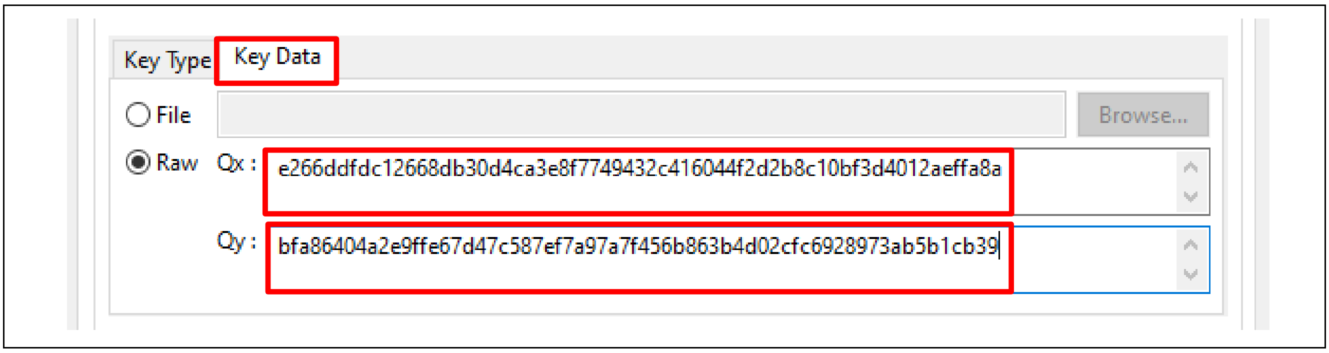


Figure 84. Provide the New ECC Public Key Data

Next, under the **Wrapping Key** section, click the corresponding **Browse** button to select the KUK generated in section 5.1.3. For the **IV**, select **Generate random value**. In the **Output** option, choose **C Source** and name the output as `new_ecc_public_key.c`. set the **Key name** to NEW_ECC_PUB.

Finally, click **Generate file**. Both the `new_ecc_public_key.c` and the `new_ecc_public_key.h` files will be generated.

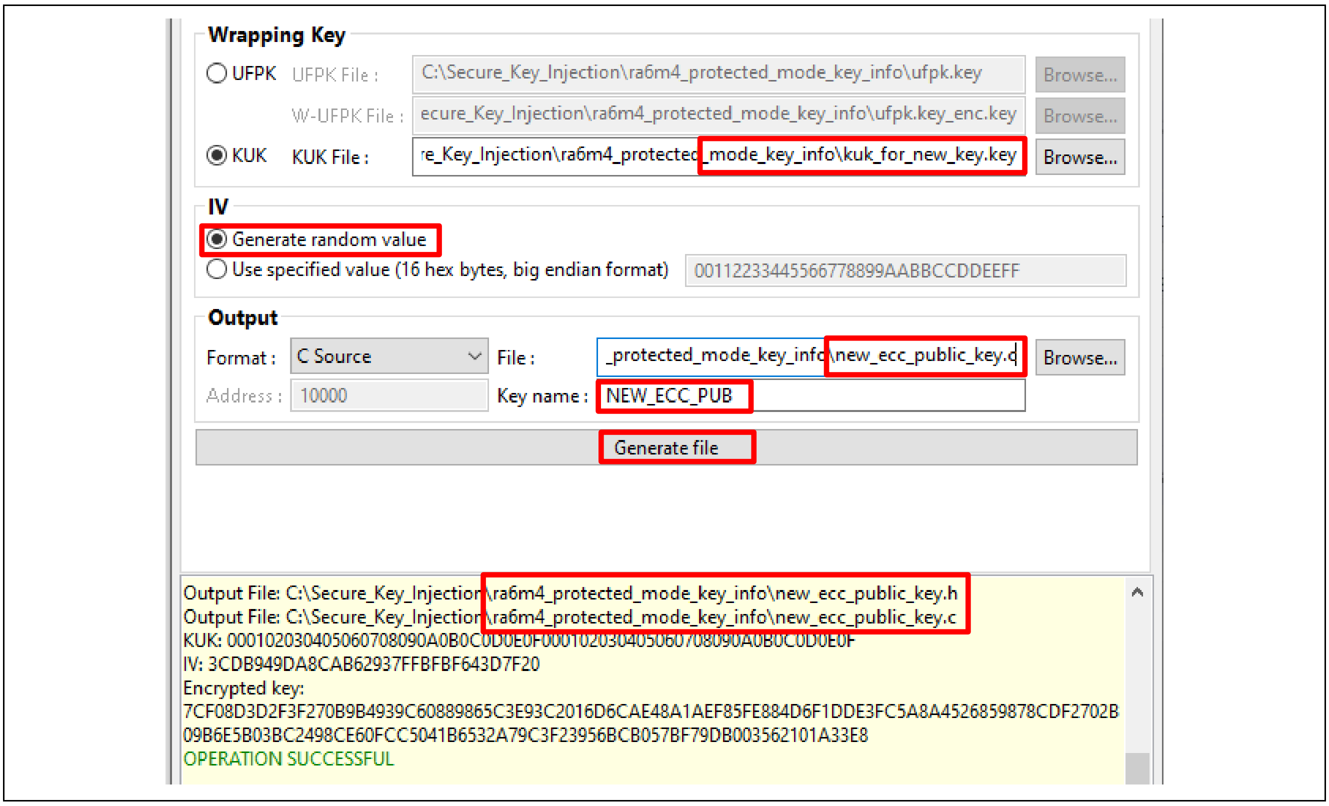


Figure 85. Generate KUK-Wrapped ECC Public Key

5.2 Wrap Keys with the UFPK and W-UFPK for RSIP-E50D Protected Mode using the SKMT GUI Interface

5.2.1 Wrap an Initial ECC Key Pair with the UFPK

The same NIST CAVP test vector is shown in Figure 71 is used for the demonstration.

To wrap a ECC public key, from the SKMT GUI, make sure **RA Family, RSIP-E50D Security Functions, and Protected Mode** is selected from the **Overview** page. Next, on the **Wrap Key** tab, select the **Key Type** as **ECC** and **secp256r1, public,** as shown in Figure 86.

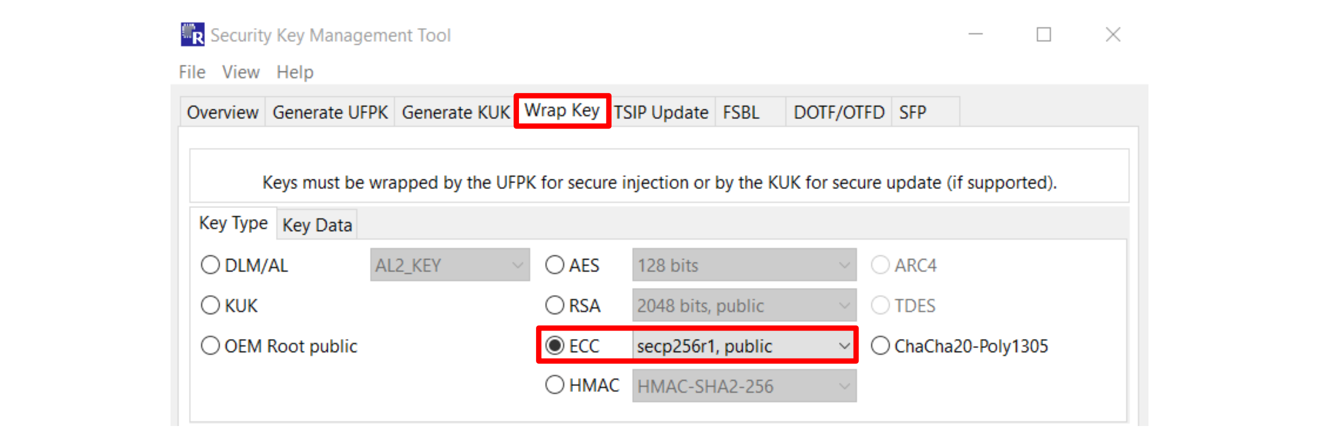


Figure 86. Choose secp256r1 Public Key

Next, configure the **Key Data**. Under the **Key Data** area, select **Raw** and provide the **Qx** and **Qy** as shown below. The key data is duplicated here to easily copy and paste to the GUI interface.

Qx = 1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83

Qy = ce4014c68811f9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9

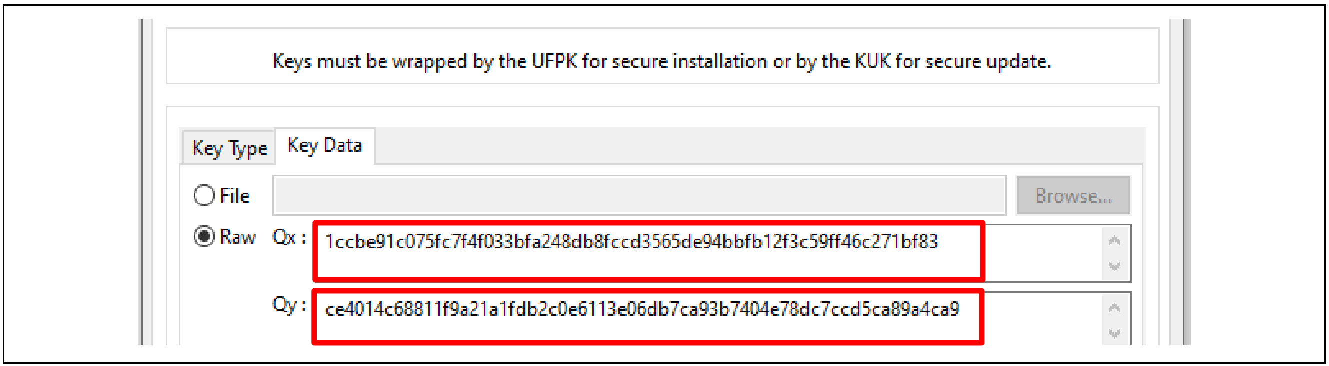


Figure 87. Provide the ECC Public Key data

Next, under the **Wrapping Key** section, click the corresponding **Browse** buttons to select the **UFPK** and **W-UFPK** key pair created in section 4.2 and 4.4. For the **IV**, select **Generate random value**. For the **Output** option, select **RFP**; then click the **Browse** button, choose the output folder, and name the output file.

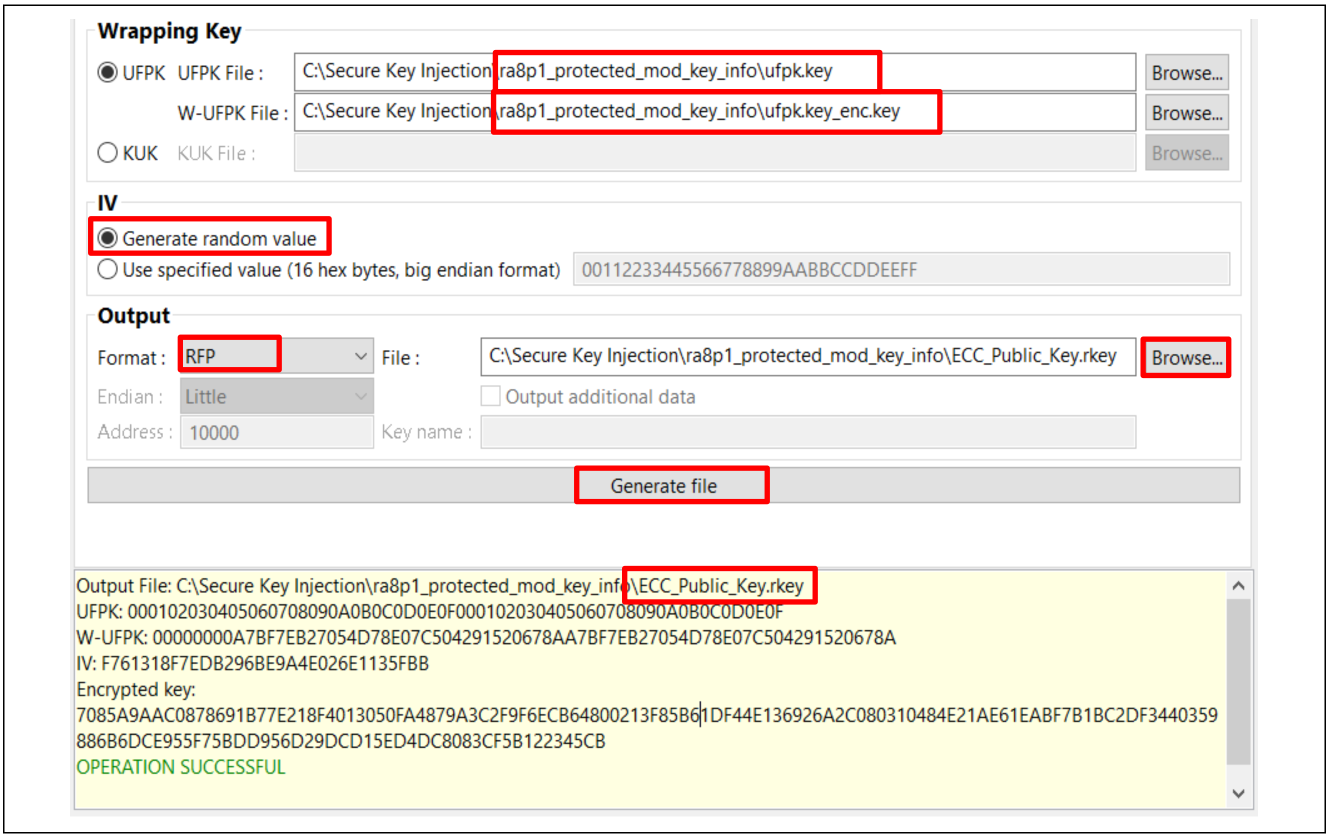


Figure 88. Generate the ECC Public Key RFP Injection Key File using GUI

The plaintext ECC key and UFPK are NOT contained in the `*.rkey` file, enabling confidential transfer of the key injection file contents

Follow the steps described above for generating a wrapped public key to wrap the corresponding private key with the UFPK. On the **Wrap Key** tab, select the **Key Type** as **ECC** and **secp256r1, private,** as shown in Figure 89.

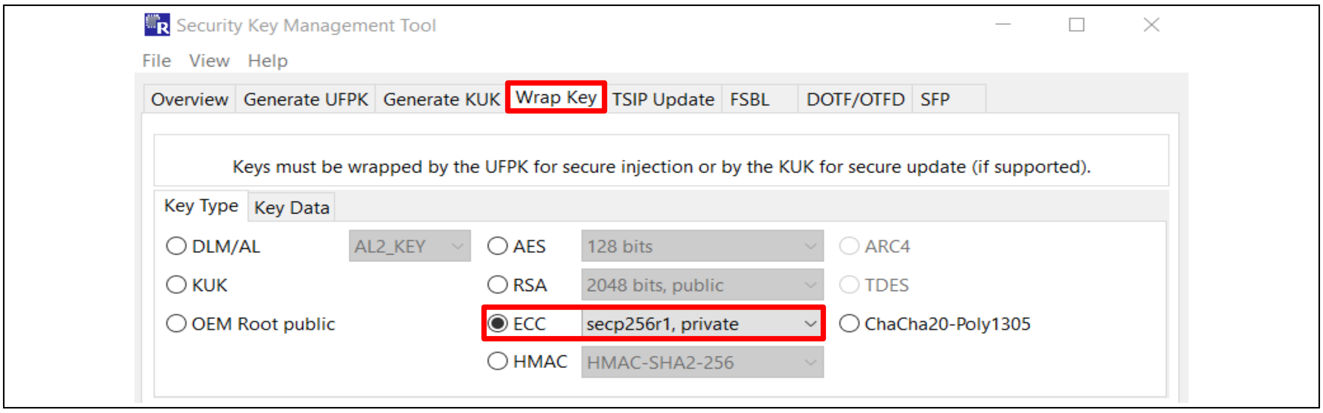


Figure 89. Choose secp256r1 Private Key

Next, configure the **Key Data**. Under the **Key Data** area, select **Raw** and provide the **d** as shown below. The key data is duplicated here to easily copy and paste to the GUI interface.

d = 519b423d715f8b581f4fa8ee59f4771a5b44c8130b4e3eacca54a56dda72b464

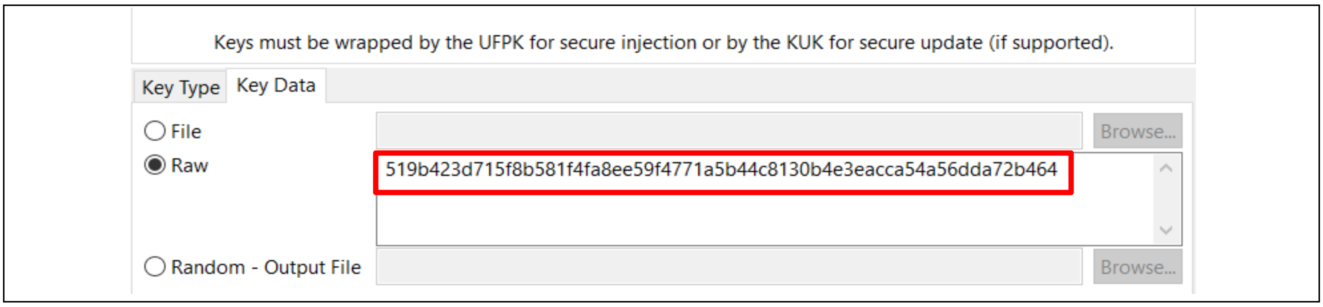


Figure 90. Provide the ECC Private Key data

Next, under the **Wrapping Key** section, click the corresponding **Browse** buttons to select the **UFPK** and **W-UFPK** key pair created in section 4.2 and 4.4. For the **IV**, select **Generate random value**. For the **Output** option, select **RFP**; then click the **Browse** button, choose the output folder, and name the output file.

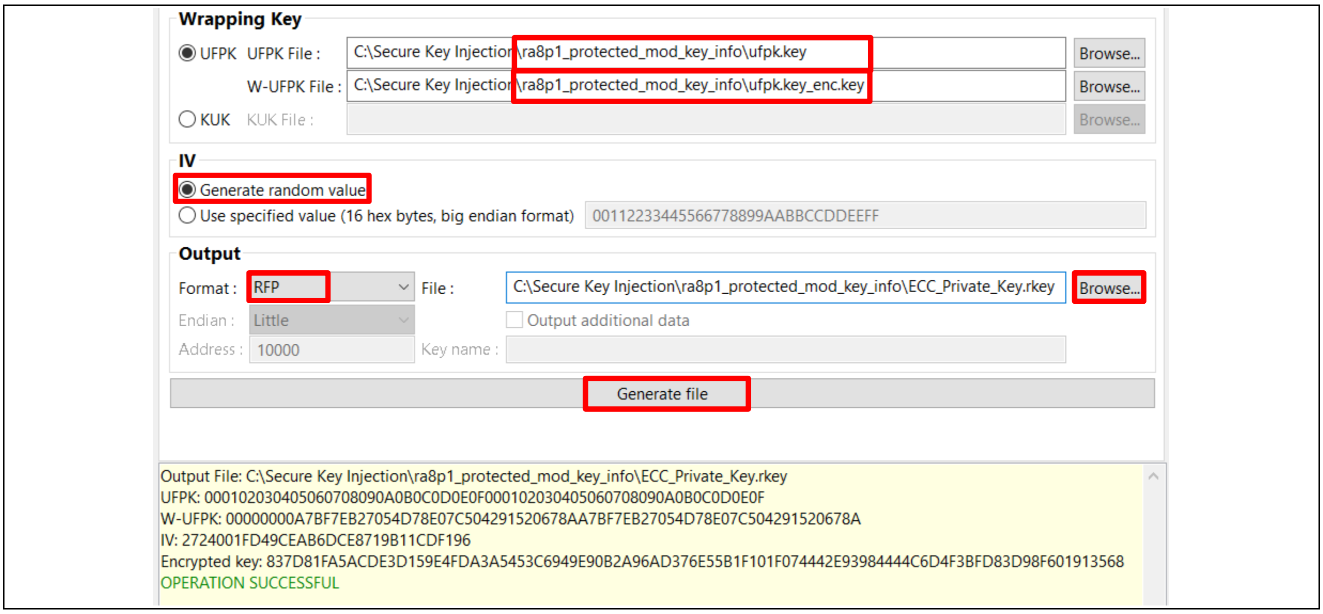


Figure 91. Generate the ECC Private Key RFP Injection Key File using GUI

The plaintext ECC key and UFPK are NOT contained in the `*.rkey` file, enabling confidential transfer of the key injection file contents.

5.2.2 Wrap a New ECC Key Pair with the KUK

Instruction to create a KUK and wrap it with the UFPK can be referred to section 5.1.3 with **RA Family, RSIP-E50D Security Functions, and Protected Mode** is selected from the **Overview** page of SKMT GUI. In this section, we will use the `kuk_for_new_key.key` generated in Figure 75 to wrap a new ECC key pair.

To demonstrate updating the ECC key pair, another NIST ECC secp256r1 test vector is used in this application project. This test vector is the same as the new NIST CAVP test vector shown in Figure 83.

From the SKMT GUI, select **RA Family, RSIP-E50D Security Functions, and Protected Mode** on the **Overview** page. Next, navigate to the **Wrap Key** page. Select the **Key Type** as **ECC** and **secp256r1, public** as shown in Figure 86.

Under the **Key Data** area, select **Raw** and provide **Qx** and **Qy** as shown below. The key data is duplicated here so the user can copy and paste it to the GUI interface.

Qx = e266ddfdc12668db30d4ca3e8f7749432c416044f2d2b8c10bf3d4012aeffa8a

Qy = bfa86404a2e9ffe67d47c587ef7a97a7f456b863b4d02cfc6928973ab5b1cb39

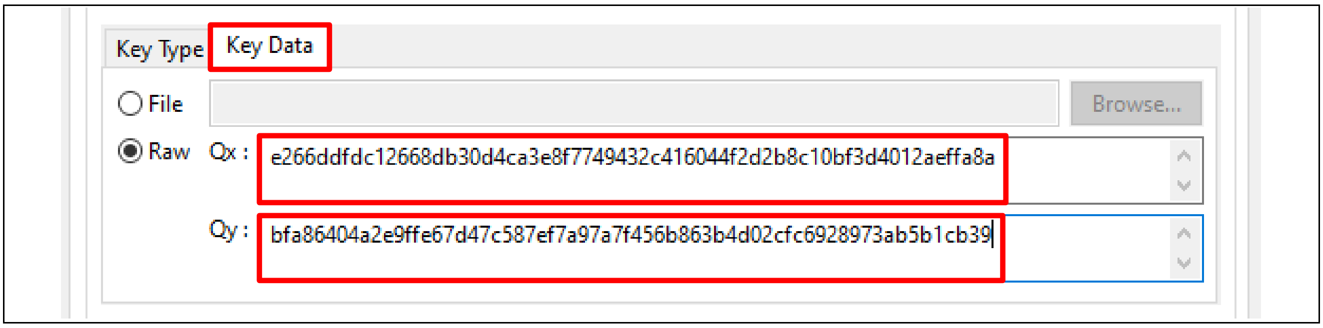


Figure 92. Provide the New ECC Public Key Data

Next, under the **Wrapping Key** section, click the corresponding **Browse** button to select the KUK generated in section 5.1.3. For the **IV**, select **Generate random value**. In the **Output** option, choose **C Source** and name the output as `new_ecc_public_key.c`. set the **Key name** to NEW_ECC_PUB.

Finally, click **Generate file**. Both the `new_ecc_public_key.c` and the `new_ecc_public_key.h` files will be generated

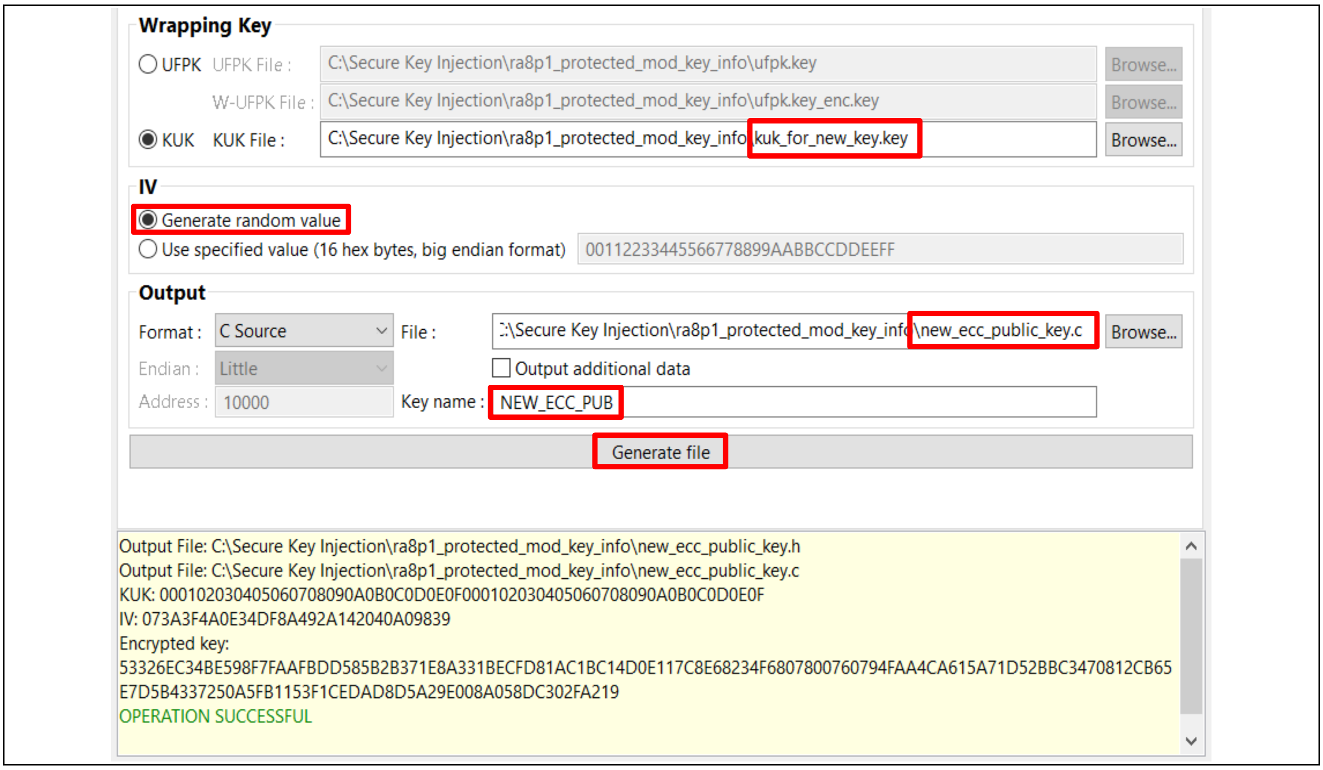


Figure 93. Generate KUK-Wrapped ECC Public Key

Follow the steps described above for generating a new wrapped public key to wrap the corresponding private key with KUK. On the **Wrap Key** tab, select the **Key Type** as **ECC** and **secp256r1, private,** as shown in Figure 89.

Next, configure the **Key Data**. Under the **Key Data** area, select **Raw** and provide the **d** as shown below. The key data is duplicated here to easily copy and paste to the GUI interface.

d = 0f56db78ca460b055c500064824bed999a25aaf48ebb519ac201537b85479813

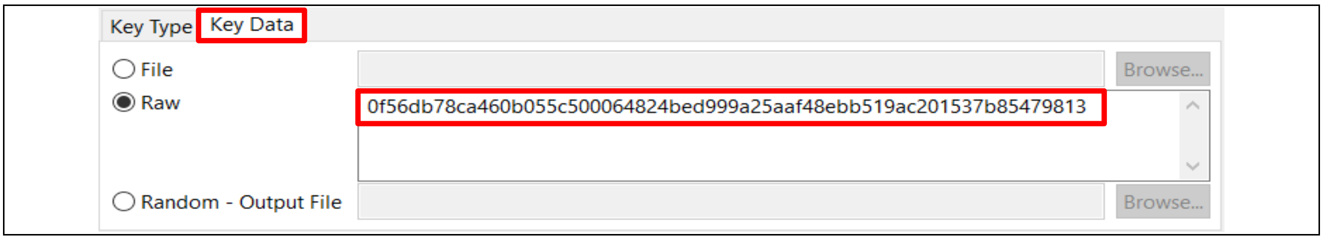


Figure 94. Provide the New ECC Private Key Data

Next, under the **Wrapping Key** section, click the corresponding **Browse** button to select the KUK generated in section 5.1.3. For the **IV**, select **Generate random value**. In the **Output** option, choose **C Source** and name the output as `new_ecc_private_key.c`. set the **Key name** to NEW_ECC_PRI.

Finally, click **Generate file**. Both the `new_ecc_private_key.c` and the `new_ecc_private_key.h` files will be generated.

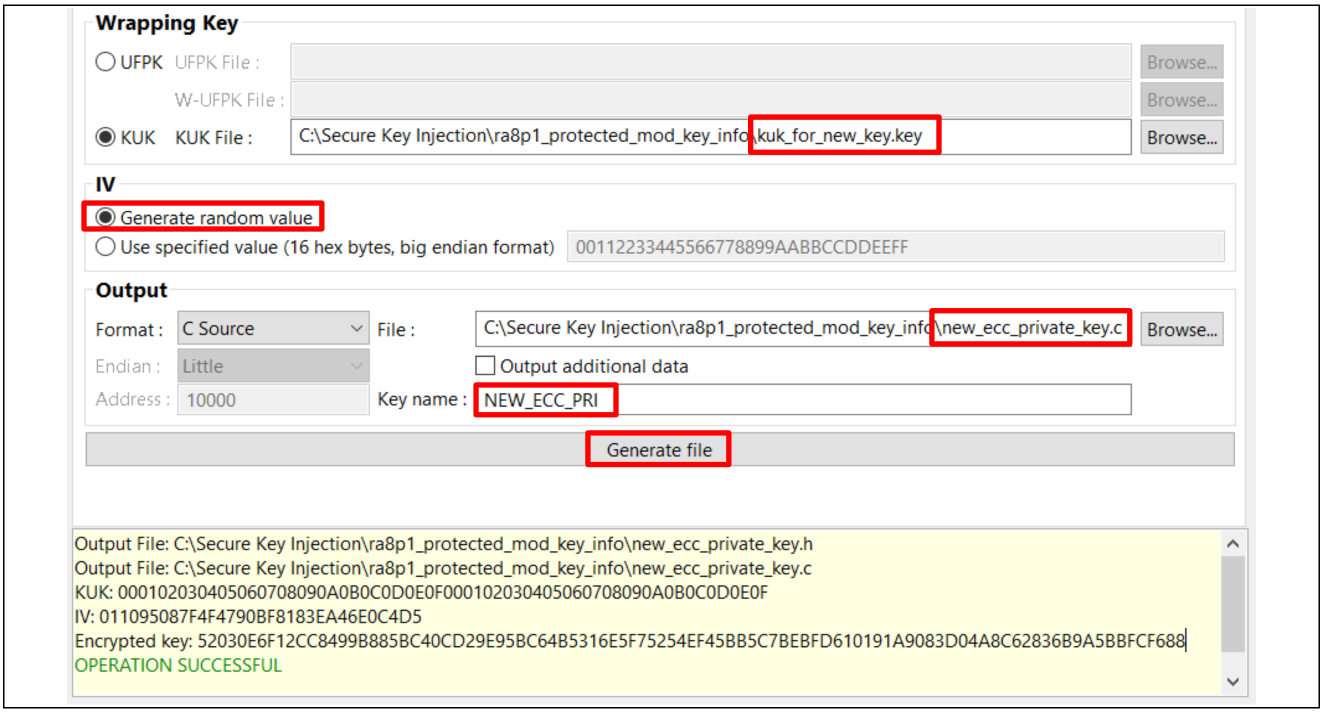


Figure 95. Generate KUK-Wrapped ECC Public Key

5.3 Wrap Keys with the UFPK and W-UFPK using the SKMT CLI Interface

This section describes how to perform the actions described above using the SKMT CLI interface. These examples use SCE9 Protected mode, but SCE7 and RSIP support is fundamentally the same.

The `/genkey` command of the Security Key Management Tool command line tool `skmt.exe` will be used to prepare keys for secure injection and update. These are the options for this command:

- `/keytype` – This input can take either ASCII or a one-byte hexadecimal input parameter indicating the key type.
- `/ufpk` – The User Factory Programming Key.
- `/wufpk` – The Renesas HRK-wrapped UFPK.
- `/kuk` – The Key-Update Key for secure key update.
- `/mcu` – The target MCU and security engine.
- `/output` – The output of the command.

Refer to the Security Key Management Tool user's manual for more information about these commands, including the valid values for each parameter.

This application project uses an AES-256 key and an ECC secp256r1 public key to illustrate the secure key injection and update processes.

For these examples, we will use the UFPK and W-UPFK created earlier.

5.3.1 Wrap an Initial AES-256 Key with the UFPK

In the Command Prompt window opened earlier (section 4.3), use the following command to create the AES-256 key injection file (`AES256_CLI.rkey`). Refer to the Security Key Management Tool user manual for more information on how to construct the command.

```
Skmt.exe /genkey /ufpk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ufpk.key" /wufpk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ufpk.key_enc.key"
/mcu "RA-SCE9" /keytype "AES-256" /key
"8000000000000000000000000000000000000000000000000000000000000000" /filetype
"rfp" /output
"C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\AES256_CLI.rkey"
```

Note that in this example:

- We are using `8000000000000000000000000000000000000000000000000000000000000000` from the NIST vector in Figure 66 as the AES-256 plaintext user key.
- We have specified the key type "`AES-256`".
- "`RA-SCE9`" is used for the `/mcu` option.
- We are using a randomly generated IV. The IV changes each time this command is executed.
- In this example, we have specified the complete file path for the key file `AES256_CLI.rkey`.

```
Output File: C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\AES256_CLI.rkey
UFPK: 000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F
W-UFPK: 000000006FEE15036A3B4E726F0B3F9E1F74B7076FEE15036A3B4E726F0B3F9E1F74B707
IV: 0B730F4F7194A9CB67E284A1B0D2A370
Encrypted key:
1D6612F7F276BFBBEBE05410151C43E74E0368D3FB0688FB7A5D2D35E2B286A9963C14F3FE16A4529AAC7E8B0650EB72
```

Figure 96. Create the AES-256 User Key Injection File

The generated key file `AES256_CLI.rkey` now contains the encrypted user key along with the W-UFPK. The plaintext AES-256 key and UFPK are NOT contained in the `*.rkey` file, enabling confidential transfer of the key injection file contents.

5.3.2 Wrap an Initial ECC Public Key with the UFPK

In this section, we will use the ECC key pair in Figure 71 as an example of preparing an ECC public key for secure key injection.

In the Command Prompt window opened earlier (section 4.3), use the following command to create the ECC public key injection file (`ECC_Public_Key_CLI.rkey`). Refer to the Security Key Management Tool user manual for more information on how to construct the command.

```
Skmt.exe /genkey /ufpk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ufpk.key" /wufpk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ufpk.key_enc.key"
/mcu "RA-SCE9" /keytype "secp256r1-public" /key
"1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83ce4014c68811f
9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9" /filetype "rfp" /output
"C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ECC_Public_Key_CLI.rkey
"
```

Note that in this example:

- `1ccbe91c075fc7f4f033bfa248db8fccd3565de94bbfb12f3c59ff46c271bf83 ce4014c68811f9a21a1fdb2c0e6113e06db7ca93b7404e78dc7ccd5ca89a4ca9` is the NIST ECC public key from Figure 71.
- We have specified the key type "secp256r1-public".
- "`RA-SCE9`" is used for the `/mcu` option.

- We are using a randomly generated IV. The IV is updated in each encryption instance.
- The command option `/output` defines the locations and name of the output file.

```
Output File: C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ECC_Public_Key_CLI.rkey
UFPK: 000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F
W-UFPK: 1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEFE12345678
IV: 0273B7277508F33491F2BA569B092535
Encrypted key:
1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234
567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF
```

Figure 97. Create the ECC Public Key Injection File Using CLI

5.3.3 Create and Wrap a Key-Update Key with the UFPK

We can use the SKMT to create a key file for a KUK. This is done with the following command:

```
skmt.exe /genkuk /kuk
"000102030405060708090A0B0C0D0E0F000102030405060708090a0b0c0d0e0f" /output
"C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\kuk_for_new_key_cli.key
"
```

Note that in this example:

- We have specified the complete file path for the key file.
- We need to use the same Key-Update Key as used in section 5.1.3.

```
KUK: 000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F
Output File:
C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\kuk_for_new_key_cli.key
```

Figure 98. Create the KUK Key File

The generated key file `kuk_for_new_key_cli.key` now contains the KUK. Retain this key file to use for wrapping new user keys for secure key updates.

To enable secure key updates, we must first securely inject the KUK. Use the SKMT to wrap the KUK with the UFPK and create a key injection file for use with RFP with the following command:

```
skmt.exe /genkey /ufpk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ufpk.key" /wufpk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\ufpk.key_enc.key"
/mcu "RA-SCE9" /keytype "key-update-key" /key
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\kuk_for_new_key_cl
i.key" /filetype "rfp" /output
"C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\KUK_CLI.rkey"
```

Note that in this example:

- We are using the KUK key file created above.
- We have specified the key type "`key-update-key`".
- We are using a randomly generated IV. The IV changes each time this command is executed.
- In this example, we have specified a complete file path for the key file (`KUK_CLI.rkey`).

```
Output File: C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\KUK_CLI.rkey
UFPK: 000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F
W-UFPK: 1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEFE12345678
IV: 1234567890ABCDEF1234567890ABCDEF
Encrypted key:
1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF1234567890ABCDEF
```

Figure 99. Create the Key-Update Key Injection File Using CLI

The generated key file `KUK_CLI.rkey` now contains the wrapped KUK along with the W-UFPK. The plaintext KUK and UFPK are NOT contained in the `*.rkey` file, enabling confidential transfer of the key injection file contents.

5.3.4 Wrap a New AES-256 Key with the KUK

The user can use the following command to wrap the new AES key defined in Figure 79 using the KUK. This is done with the following command.

```
C:\Renesas\SecurityKeyMangementTool\cli>skmt.exe /genkey /kuk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\kuk_for_new_key_cli.
key" /mcu "RA-SCE9" /keytype "AES-256" /key
"c000000000000000000000000000000000000000000000000000000000000000" /filetype
"csource" /keyname "NEW_AES256" /output
"C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\new_aes_key_cli.c"
```

Note that in this example:

- We are using `c000000000000000000000000000000000000000000000000000000000000000` as the new AES-256 plaintext key.
- We are using a randomly generated IV. The IV changes each time this command is executed.
- We use the `/keyname` to create an identifiable key structure name that is unique in the software project. This resolves confusions when more than one set of new user keys are to be generated. If this option is not provided, a key structure name of `encrypted_user_key_data` is generated for the key structure.
- The generated `new_aes_key_cli.c` and `new_aes_key_cli.h` files include the output information in a data structure. The user can directly include these two files in the application project. This is demonstrated in the example project included.

```
Output File: C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\new_aes_key_cli.h
Output File: C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\new_aes_key_cli.c
KUK: 000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F
IV: 3C8841F6E6AE05B7625098EC70C542C1
Encrypted key:
03FE218ABCD0AD2F5A5634833ABD7F4D6F4CF8BF2CAC737CE1BE56C28DF0ADAD52536EED8DF405031230F935B087ECA0
```

Figure 100. Encrypt the New User Key with the KUK

5.3.5 Wrap a New ECC Public Key With the KUK

Use the following command to wrap the new ECC public key shown in Figure 83.

```
skmt.exe /genkey /kuk
file="C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\kuk_for_new_key.ke
y" /mcu "RA-SCE9" /keytype "secp256r1-public" /key
"e266ddfdc12668db30d4ca3e8f7749432c416044f2d2b8c10bf3d4012aeffa8abfa86404a2e9f
fe67d47c587ef7a97a7f456b863b4d02cfc6928973ab5b1cb39" /filetype "csource"
/keyname "NEW_ECC_PUB" /output
"C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\new_ecc_public_key_cli.
c"
```

Note that in this example:

- `e266ddfdc12668db30d4ca3e8f7749432c416044f2d2b8c10bf3d4012aeffa8a bfa86404a2e9ffe67d47c587ef7a97a7f456b863b4d02cfc6928973ab5b1cb39` is the ECC public key from the NIST test vector shown in Figure 83.
- The key type "secp256r1-public" is one of the available options specified in the Security Key Management Tool user's manual.
- "`RA-SCE9`" is used for the `/mcu` option.
- We are using a randomly generated IV. The IV changes each time this command is executed.
- The command option `/output` defines the locations and name of the output file.

- We use the `/keyname` to create an identifiable key structure name that is unique in the software project. This resolves confusions when more than one set of new user keys are to be generated. If this option is not provided, a key structure named `encrypted_user_key_data` is generated for the key structure.
- The generated `new_ecc_public_key_cli.c` and `new_ecc_public_key_cli.h` files include the output information in a data structure. 0This is demonstrated in the example project included.

```
Output File: C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\new_ecc_public_key_cli.h
Output File: C:\Secure_Key_Injection\ra6m4_protected_mode_key_info\new_ecc_public_key_cli.c
KUK: 000102030405060708090A0B0C0D0E0F000102030405060708090A0B0C0D0E0F
IV: 36E763D5A82924B4888732D50C93B602
Encrypted key:
9B0A7F8C91C038704A4F2C758EAC3DDD1372B4DC6AA4F22667D7D0E41218A1DEDBB8337E557B59B91100225BC8BBE2807221
4FF3C729D953AEFA9E997C3989967C831DC6501E9528715ADA30FA0D0402
```

Figure 101. Encrypt the New ECC Public Key with the KUK

5.4 Secure Key Injection via MCU Boot Interface

Follow this section to inject the AES-256 key, the ECC public key, the ECC private key and the Key-Update Key (KUK) that were prepared in section **Error! Reference source not found.**, section 5.2 or section 5.3. This capability is supported by RA Family MCUs that incorporate the SCE9 (Protected Mode), RSIP (Protected Mode) or SCE5_B security engine.

5.4.1 Setting up the Hardware

Set up the EK-RA6M4 evaluation board as follows.

- Connect the jumper setting to J16 to put the device in boot mode. Refer to the EK-RA6M4 User's Manual for details.
- Connect the EK-RA6M4 J10 connector to the development PC using a USB micro-B cable to provide power and a debug connection using the onboard debugger.

Erase the entire MCU flash and ensure that the MCU is in the SSD Device Lifecycle State. This can be done using the Renesas Flash Programmer, as shown here.

1. Unzip `rfp_resources_ra6m4.zip`
2. Launch the Renesas Flash Programmer GUI executable.
3. Select **File → Open Project** and select `ra6m4_secure_key_inject.rpj`.
4. Select **Target Device → Initialize Device**.

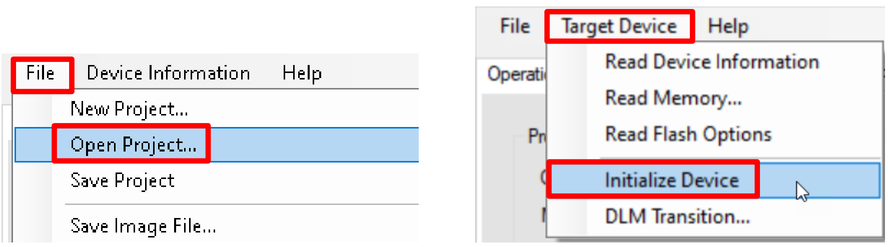


Figure 102. Open RFP Project and Initialize the Device

Upon successful initialization, the following message will be printed.

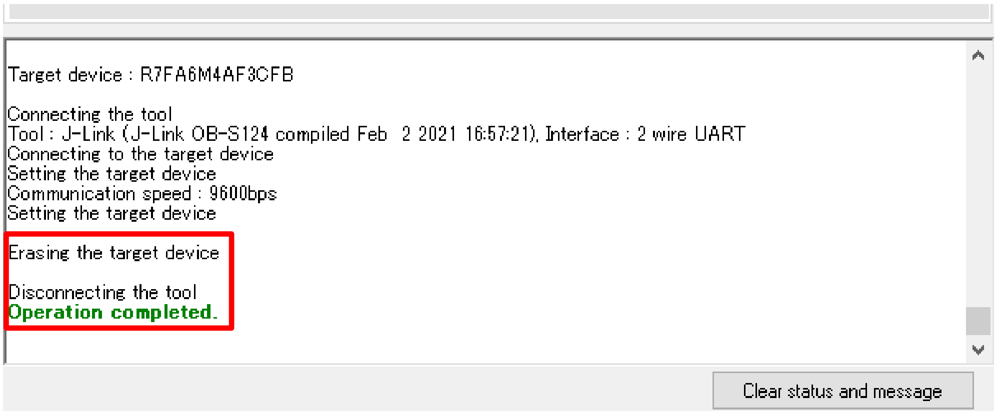


Figure 103. RA6M4 Initialization

Unless there are permanently locked flash blocks, the entire flash will be erased, and the RA6M4 will be set to SSD state through the above steps.

In case of using EK-RA8P1 evaluation board, the setup is as follows:

- Set jumper J16 on pins 2-3 to put the device in JTAG boot mode. Refer to the EK-RA8P1 User's Manual for details.
- Set jumper J9 on pins 2-3 and jumper J8 on pins 1-2 for normal operation.
- Connect J10 from EK-RA8P1 to the development PC to provide power and debugging capability using a type A to type C USB cable.

Erase the entire MCU MRAM and ensure that the MCU is in the OEM Device Lifecycle State. This can be done using the Renesas Flash Programmer, as shown here.

5. Unzip `rfp_resources_ra8p1.zip`
6. Launch the Renesas Flash Programmer GUI executable.
7. Select **File → Open Project** and select `ra8p1_secure_key_inject.rpj`.
8. Select **Target Device → Initialize Device** as shown in Figure 102.

Upon successful initialization, the following message will be printed, similar to Figure 103.

5.4.2 Inject the Initial User Key and Key-Update Key

After initializing the evaluation board, power-cycle the board and follow the steps below to inject the AES-256 key, the ECC public key, and the Key-Update Key if using EK-RA6M4 or inject the ECC key pair and the Key-Update Key if using EK-RA8P1. This section uses the set of injection keys generated from the GUI interface.

To simplify duplicating this example, the `.rkey` files that match the example project are included in the `rfp_resources_<mcu_name>.zip` file. If the user intends to use the NIST vectors included in this application project for verification purposes, they can use the included `.rkey` files for system verification. The screen captures included in this section use these files for demonstration purposes. If different keys are used, then the corresponding `.rkey` files must be updated to match those keys.

The following steps explain how to inject keys in the EK-RA6M4 example project.

Under the **Operation** tab, click **Add/Remove Files**. Next, click **Add Files**, and then add the `.rkey` file containing the AES256 key, which for this example is `\rfp_resources_ra6m4\user_keys\AES256.rkey` (Figure 69). Set the **Address** property to a data flash or code flash address applicable to your specific application. In this example, the AES key will be injected into the first block of Data Flash at 0x08000000.

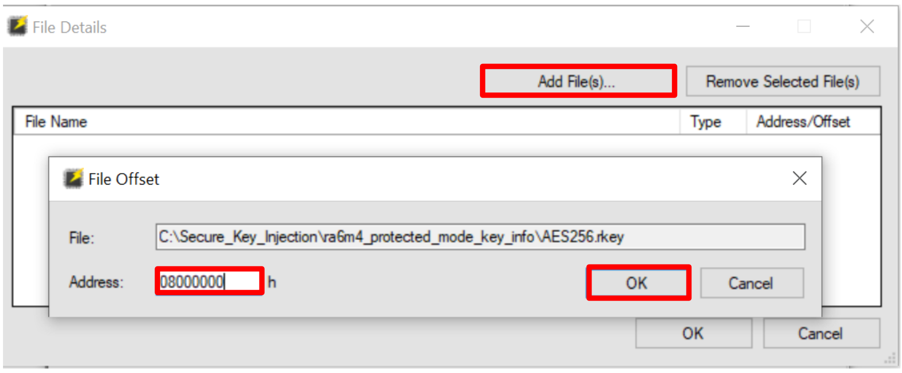


Figure 104. Add the AES256.rkey to RFP Configuration

Click **OK**, the AES256.rkey file will be configured to the corresponding load address.

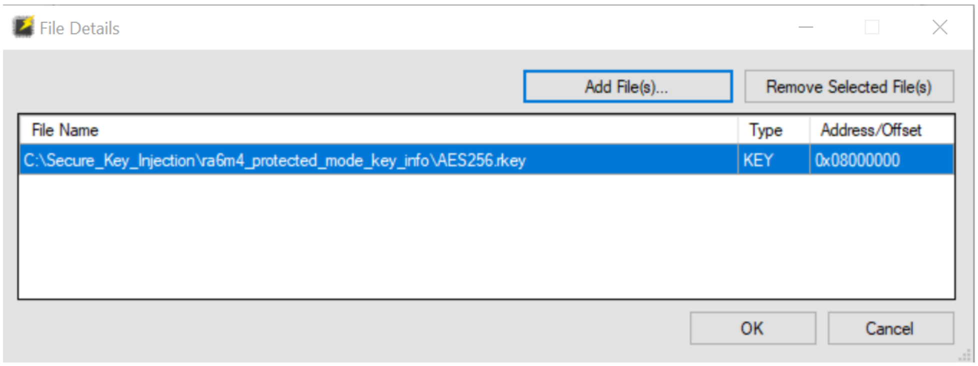


Figure 105. AES256.rkey is added to the RFP Configuration

Click **Add Files** again and add `ECC_Public_Key.rkey`. Browse to the `ECC_Public_Key.rkey` (Figure 74). Set the **Address** property to a data flash or code flash address applicable to your specific application. In this example, the ECC public key will be injected into the third block of Data Flash at 0x08000080.

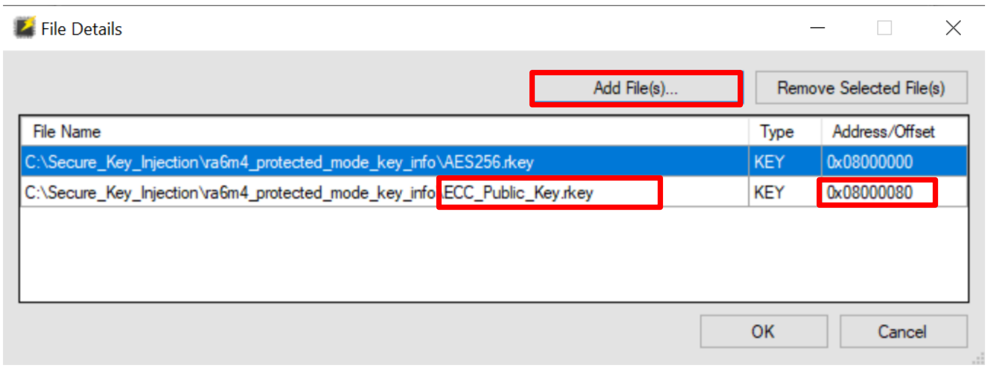


Figure 106. Configure the ECC Public Key Selection and Injection Address

Click **Add Files** again and add `KUK.rkey`. Browse to the `KUK.rkey` (Figure 78). Set the **Address** property to a data flash or code flash address applicable to your specific application. In this example, the Key-Update Key will be injected into the code flash at 0x00040000.

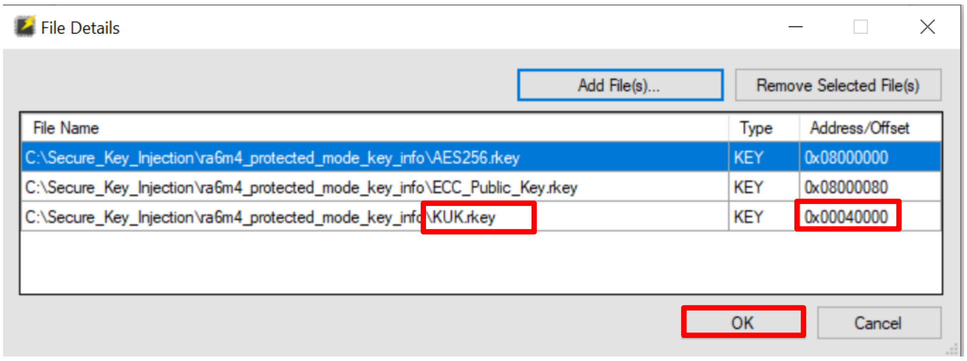


Figure 107. Configure the Key-Update Key Selection and Injection Address

Click **OK** and navigate to the **Operation Settings**. Note that **Erase**, **Program**, **Verify**, and **Erase Before Program** are selected.

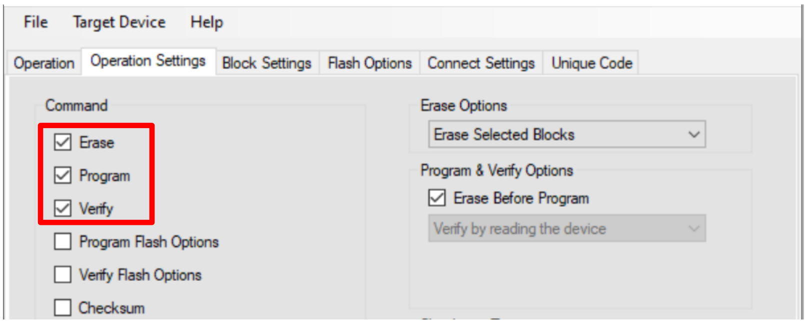


Figure 108. Select to Perform Flash Erase, Program, and Verify

- Browse to the **Block Settings** tab and note that the entire flash region is selected for Erase.

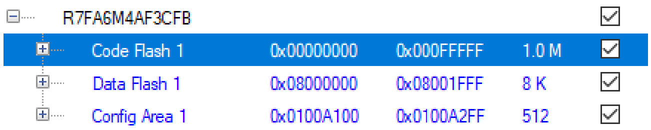


Figure 109. Entire Flash Region is Selected for Erase

- Browse the **Operation** tab. Click **Start** to inject the AES-256, the ECC public key, and the Key-Update Key. The injection should succeed with a similar output message as shown below at the selected flash addresses.

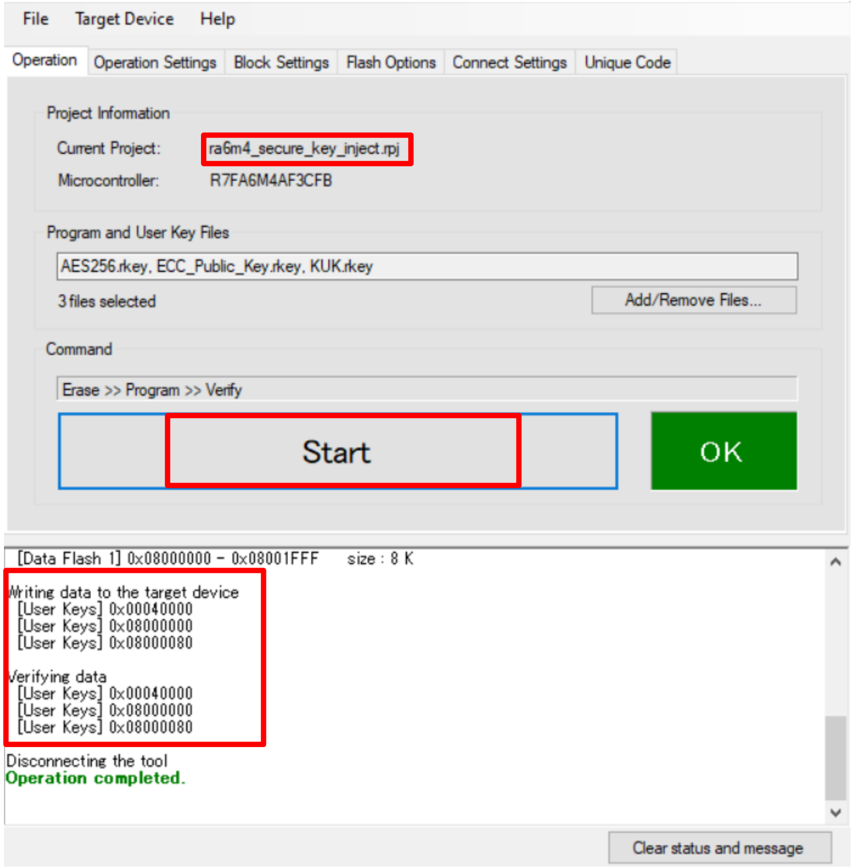


Figure 110. Secure Keys Successfully Injected

In this example code, no application is programmed since we are interested only in the key injection. In a production flow, it is possible to program the application and user keys together. This operation can also be performed using the command line function of RFP.

With the RA8P1 example project, the ECC key pair and the Key-Update Key will be injected following the same steps described above but using different addresses corresponding to the MRAM on the RA8P1 as shown in Figure 111.

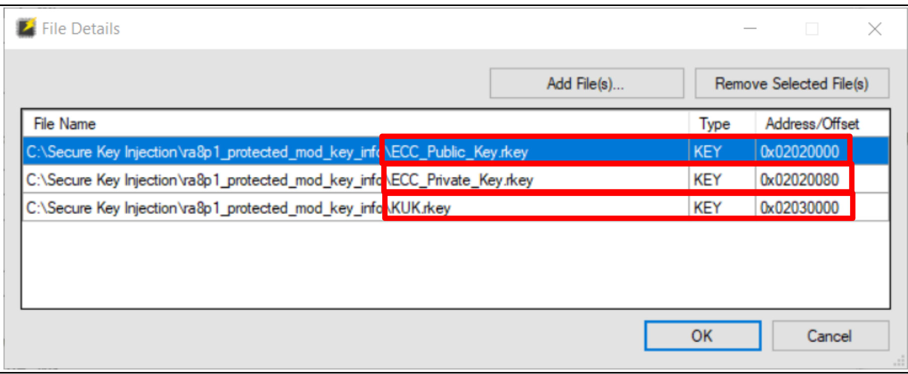


Figure 111. Configure the Keys Selection and Injection Address with RA8P1 project

In this example:

- The ECC public key will be injected into MRAM at 0x02020000.
- The ECC private key will be injected into MRAM at 0x02020080.

- The Key-Update Key will be injected into the MRAM at 0x02030000.

6. Secure Key Injection Preparation for RSIP and SCE7 Compatibility Mode

This section shows how to generate the `.c` and `.h` files, which can be used in an application project that uses the FSP APIs to inject keys into the PSA Crypto APIs using the security engine in Compatibility Mode. This key injection method must be used for both user keys and Key-Update Keys.

6.1 Wrap an AES-128 User Key Using the UFPK for RSIP-E51A Compatibility Mode

A NIST CAVP test vector is used for the demonstration.

https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program/Block-Ciphers

```
KEY = e0000000000000000000000000000000
IV = 00000000000000000000000000000000
PLAINTEXT = 00000000000000000000000000000000
CIPHERTEXT = 72a1da770f5d7ac4c9ef94d822affd97
```

Figure 112. NIST AES-128 Test Vector

Using the SKMT GUI interface, on the **Overview** tab, select **RA Family, RSIP-E51A Compatibility Mode**.

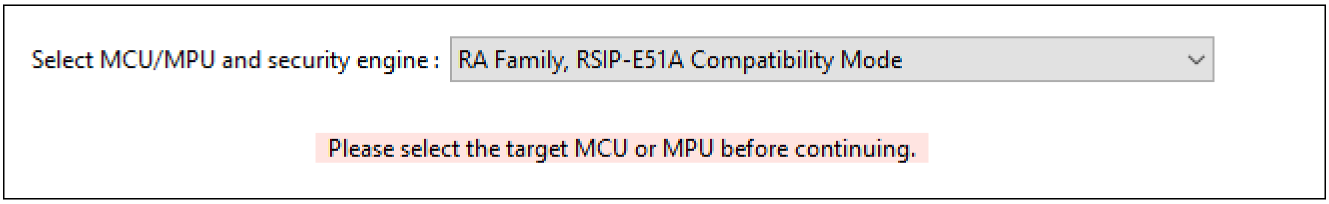


Figure 113. Choose RA Family, RSIP-E51A Compatibility Mode

On the **Wrap Key** tab, in the **Key Type** area, choose **AES** and **128 bits**.

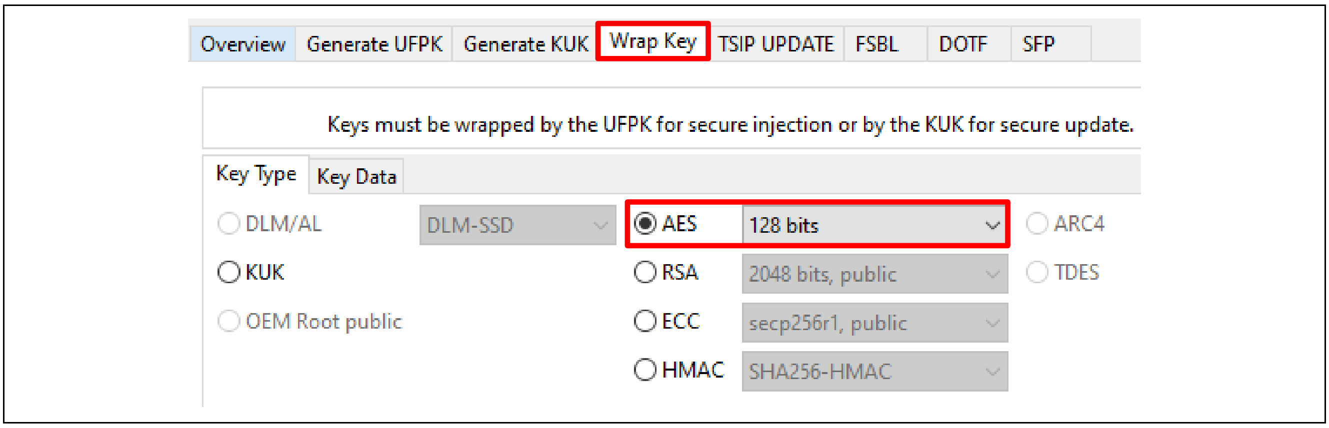


Figure 114. Choose AES-128 bits as the Key Type

Select the **Key Data** tab and input the **Raw** Key Data as shown below based on the NIST vector as shown in Figure 112.

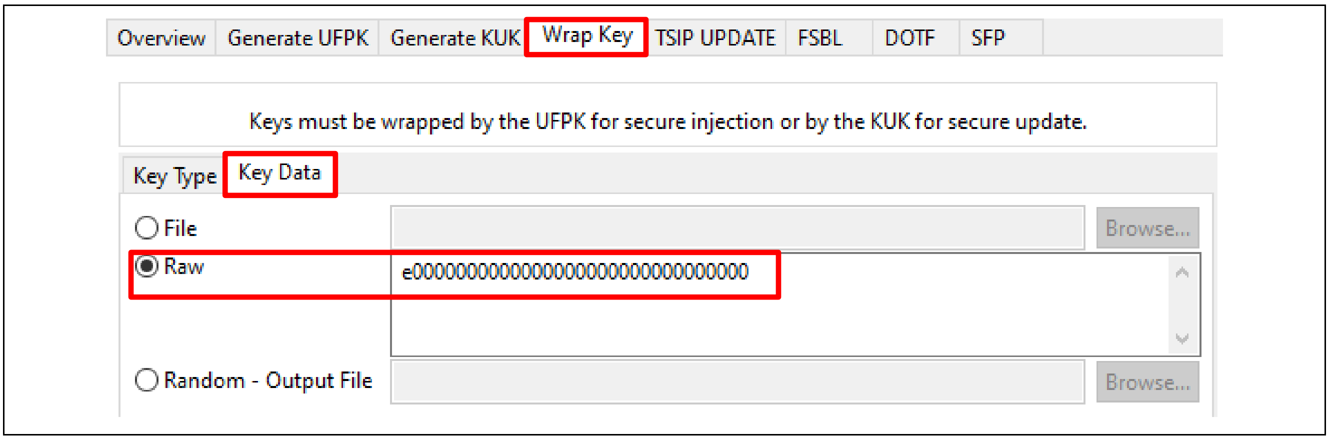


Figure 115. Set up the Initial AES-128 Key Data

Under the **Wrapping Key** section, click the corresponding **Browse** buttons to select the **UFPK** and **W-UFPK** key pair. Choose the **Generate random value** option for the **IV** data. For the **Output** option, select **C Source**; then click the **Browse** button, choose the output folder and file name, and name the key. This name will be reflected in the definitions generated for the C source files.

Now click the **Generate File** button. The source files to inject the AES key will be generated.

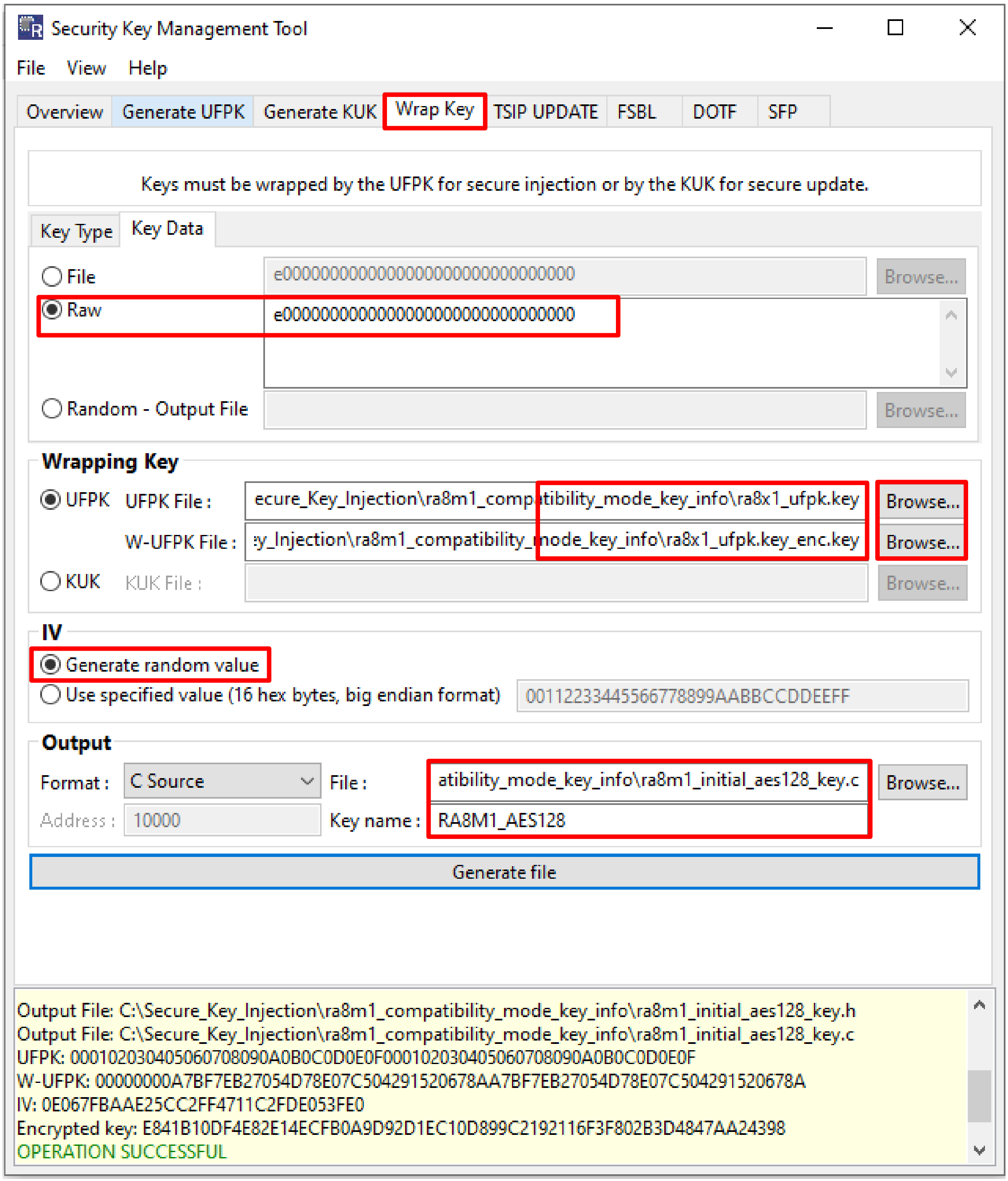


Figure 116. Generate the Initial AES-128 Encrypted Key File

Note that the generated `ra8m1_initial_aes_128.c` and `ra8m1_initial_aes_128.h` are used in the RA8M1 secure key injection example project.

6.2 Wrap an AES-128 User Key Using the UFPK for SCE7

The same NIST CAVP test vector is shown in Figure 112 is used for the demonstration.

Using the SKMT GUI interface, on the **Overview** tab, select **RA Family, SCE7**. On the **Wrap Key** tab, in the **Key Type** area, choose **AES** and **128 bits**.

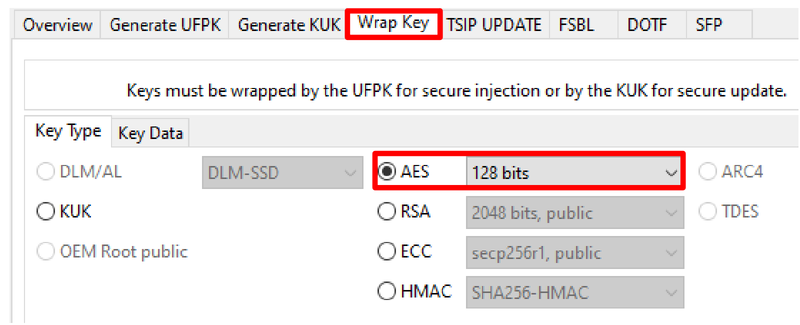


Figure 117. Choose AES-128 bits as the Key Type

Select the **Key Data** tab and input the **Raw** Key Data as shown below based on the NIST vector as shown in Figure 116.

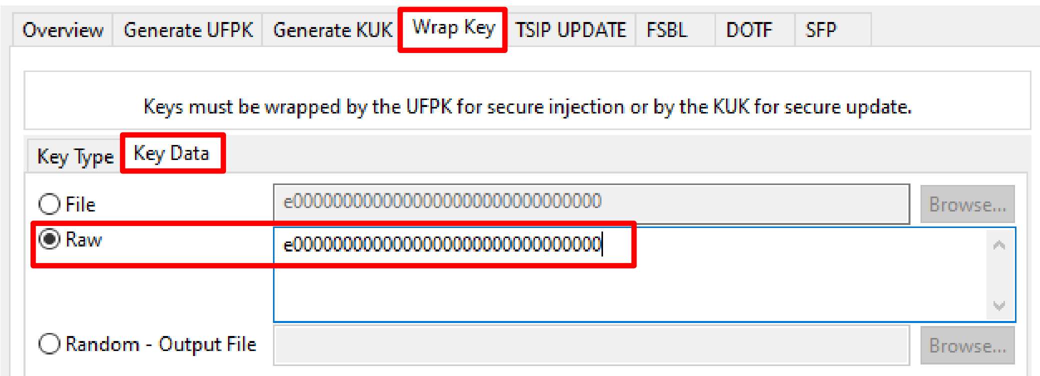


Figure 118. Set up the AES-128 Key Data

Under the **Wrapping Key** section, click the corresponding **Browse** buttons to select the **UFPK** and **W-UFPK** key pair for RA6M3. Choose the **Generate random value** option for the **IV** data. For the **Output** option, select **C Source**; then click the **Browse** button, choose the output folder and file name, and name the key. This name will be reflected in the definitions generated for the C source files.

Now click the **Generate File** button. The source files to inject the AES key will be generated.

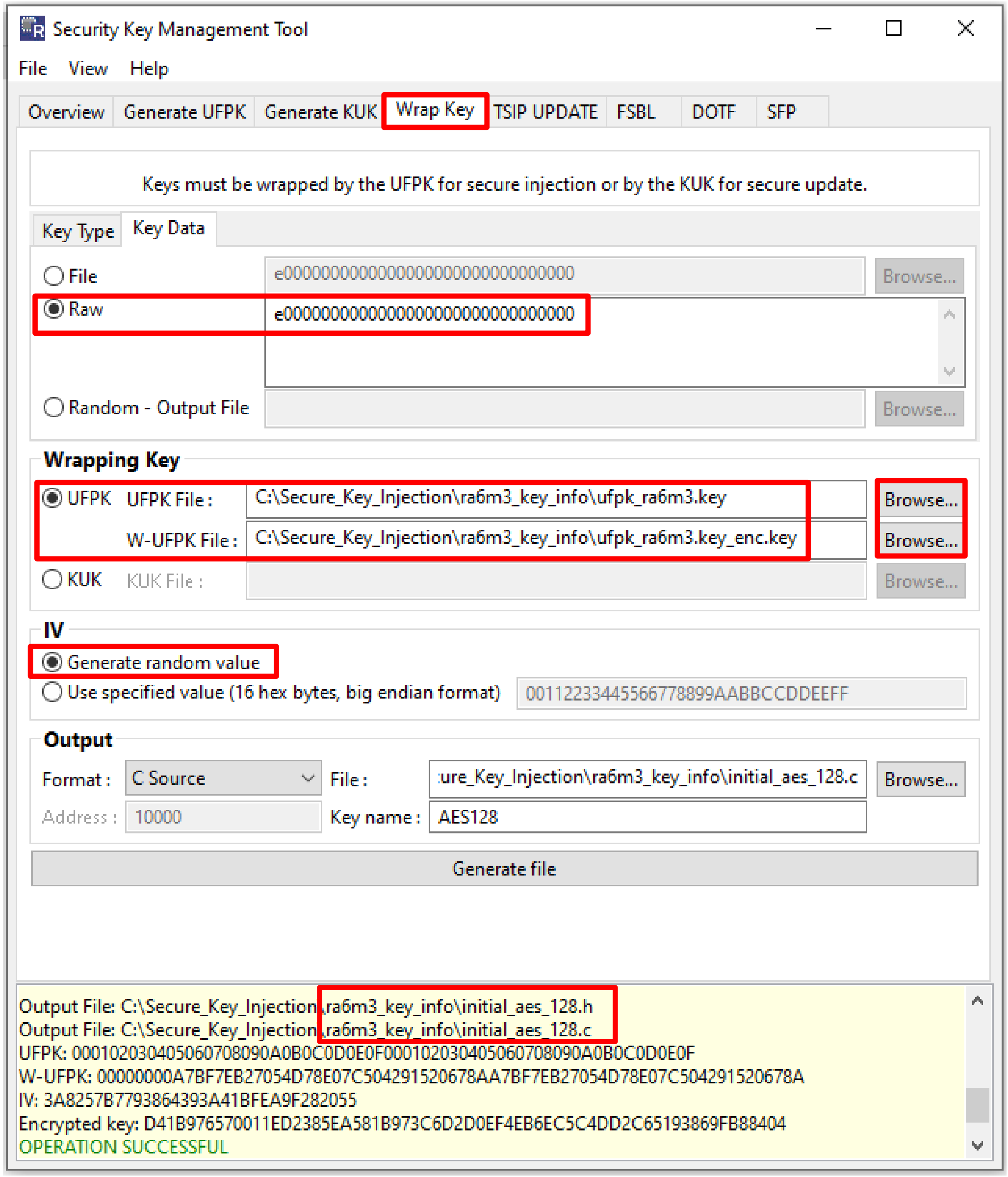


Figure 119. Generate the AES-128 Encrypted Key File

7. Example Project for RA6M4 (SCE9 Protected Mode)

To exercise the example projects as is, users can follow the steps below:

- Inject the included example RFP injection keys (`AES256.rkey`, `KUK.rkey`, and `ECC_Public_Key.rkey` which are included in `rfp_resource_ra6m4.zip`) by following section 5.4.2.
- A set of new user keys (AES256 as well as ECC Public Key) generated using the example KUK is already provisioned in the example projects. The user can then directly proceed to exercise the example project.

Note: Please do not use the example keys for production support.

To use the example projects with customized keys, the user can follow the steps below:

- To test customized RFP injection keys and new user update keys (generated by the following section 5.1 or 5.3 rather than using the ones included in `rfp_resources_ra6m4.zip`), the user needs to follow section 1.1 to inject the keys to the MCU. User also needs to generate customized new user key files (`new_aes_key.c/.h` and `new_ecc_public_key.c/.h`) with the same key name to replace the corresponding files used in the example project. Once the example projects are updated, the user can proceed to run the example projects to verify the operations.
- To test the new user key update procedure only, the user can use the included RFP `KUK.rkey` file to generate new source files to replace the corresponding files in the example project. Once the example projects are updated, the user can then proceed to the verification of the operations.

7.1 Example Project Overview

This pair of TrustZone-based secure and non-secure example projects provides the following functions:

Secure project (secure_key_inject_update_ra6m4_s):

- Uses the injected AES-256 key to perform cryptographic operation using AES256-CBC.
- The injected Key-Update Key (KUK) is used to inject the new AES-256 key and store this new AES-256 key for data flash.
- Uses the new AES-256 to perform cryptographic operation using AES256-CBC.
- Uses the injected ECC public key to verify the NIST test signature shown in Figure 71.
- The injected Key-Update Key (KUK) is used to inject the newly wrapped ECC public key and store this new ECC public key for data flash.
- Uses the new ECC public key to verify the NIST test signature shown in Figure 83.

Non-secure project (secure_key_inject_update_ra6m4_ns):

- Establishes an RTT Viewer interface to allow users to select the intended Secure Crypto Engine and flash operation.
- Calls the non-secure callable APIs provided from the secure project based on user selection from the RTT Viewer interface.
- Prints the user operation results on the RTT Viewer.

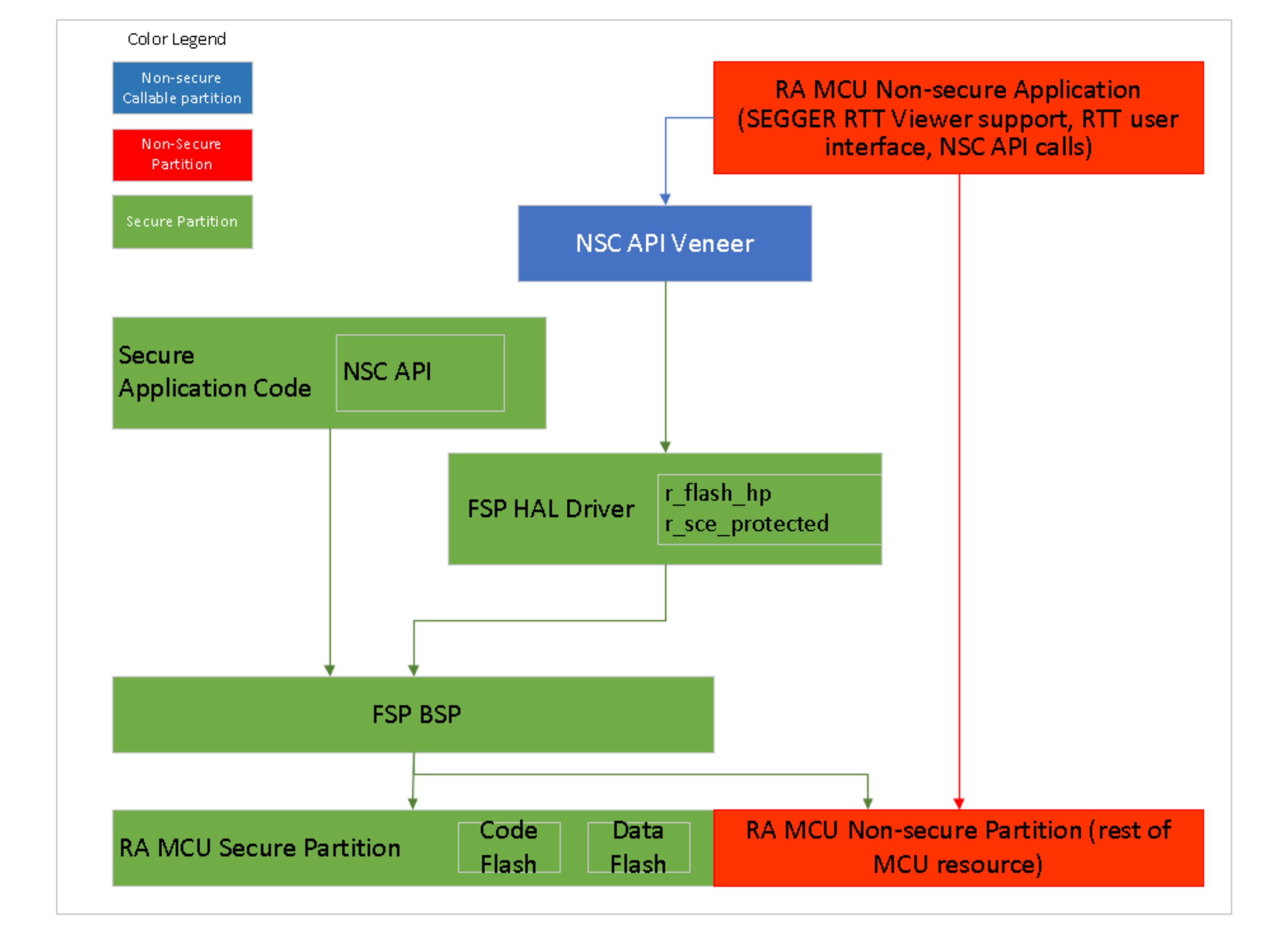


Figure 120. Software Block Diagram

The FSP modules used in this pair of example projects are:

- `r_sce_protected`: This module is used in the secure region and provides services to the non-secure region via non-secure callable APIs
- `r_flash_hp`: This module is used in the secure region and provides services to the non-secure region via non-secure callable APIs

For more information on designing applications with TrustZone® support, refer to the application project *Renesas RA Family MCU Security Design with TrustZone – IP Protection.*

7.2 Using the RFP Injected Keys

7.2.1 Formatting the Injected Keys

The keys that are injected into the MCU flash using RFP cannot be used directly by the FSP Crypto APIs. A minor formatting change is required.

7.2.1.1 Formatting the Injected AES Key

The following code snippet reads the AES-256 key from Flash. The destination buffer can then be used for cryptographic operations. Replace the macro `DIRECT_AES_KEY_ADDRESS` with the actual injection address.

```
static sce_aes_wrapped_key_t injected_key;
injected_key.type = SCE_KEY_INDEX_TYPE_AES256;
memcpy(injected_key.value, (uint32_t *)DIRECT_AES_KEY_ADDRESS,
    HW_SCE_AES256_KEY_INDEX_WORD_SIZE*4);
```

7.2.1.2 Formatting the Injected ECC Public Key

The following code snippet reads the ECC public key from Flash. The destination buffer can then be used for cryptographic operations. Replace the macro `DIRECT_ECC_PUB_KEY_ADDRESS` with the actual injection address.

```
static sce_ecc_public_wrapped_key_t ecc_public_key_injected;
ecc_public_key_injected.type = SCE_KEY_INDEX_TYPE_ECC_P256_PUBLIC;
wrapped_ecc_public_key_size = sizeof(ecc_public_key_injected.value);
memcpy((uint8_t *)(&(ecc_public_key_injected.value)), (uint8_t *)DIRECT_ECC_PUB_KEY_ADDRESS,
    wrapped_ecc_public_key_size);
```

7.2.1.3 Formatting the Injected KUK

The following code snippet reads the injected KUK from the flash. The destination buffer can then be used for secure key updates. Replace the macro `KUK_ADDRESS` with the actual injection address.

```
static sce_key_update_key_t kuk_key;
kuk_key.type = SCE_KEY_INDEX_TYPE_UPDATE_KEY_RING;
memcpy(kuk_key.value, (uint32_t *)(KUK_ADDRESS),HW_SCE_UPDATE_KEY_RING_INDEX_WORD_SIZE*4);
```

7.2.1.4 Formatting an Injected RSA Public Key

This application project does not include an example usage for RSA secure key injection and update, but the principles are identical. The following code snippet can be used to format an injected RSA public key. Replace the macro `RSA_2048_PUB_KEY_ADDRESS` with the actual injection address

```
static sce_rsa2048_public_wrapped_key_t injected_rsa_public_key;
injected_rsa_public_key.type = SCE_KEY_INDEX_TYPE_RSA2048_PUBLIC;
uint32_t wrapped_rsa_2048_public_key_size = sizeof(injected_rsa_public_key.value);
memcpy((uint8_t *)(&(injected_rsa_public_key.balur)), (uint32_t *)RSA_2048_PUB_KEY_ADDRESS,
    wrapped_rsa_2048_public_key_size);
```

7.2.2 Verifying the Injected Key and the Updated Key

To verify the AES injection, provide the plaintext message and the expected cipher text for the injected AES key and the updated AES key to the software project. For example, based on the NIST vectors presented in Figure 66 and Figure 79, use the plaintext data below in `aes_crypto_operations.c`:

```
#define BLOCK    16
/* NIST vector plaintext message  used for both directly injected AES key and updated AES key*/
static uint8_t plain_text[BLOCK] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

/* NIST vector initialization vector for the directly injected AES key and the AES key update*/
static uint8_t iv[BLOCK] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
/* NIST cipher to match directly injected AES key*/
static uint8_t cipher_expected[BLOCK] = {
   0xe3, 0x5a, 0x6d, 0xcb, 0x19, 0xb2, 0x01, 0xa0, 0x1e, 0xbc, 0xfa, 0x8a, 0xa2, 0x2b, 0x57, 0x59
};

/* NIST cipher to match new AES key */
static uint8_t cipher_expected_new[BLOCK] = {
  0xb2, 0x91, 0x69, 0xcd, 0xcf, 0x2d, 0x83, 0xe8, 0x38, 0x12, 0x5a, 0x12, 0xee, 0x6a, 0xa4, 0x00
};
```

To verify the ECC public key injection, the expected signature using the ECC private key, which matches the injected ECC public key (see Figure 71) is provided in the array `ECC_SECP256R1ExpectedSignature` in `ecc_crypto_operation.c`.

```
/* This is an externally generated NIST test signature using the private key */
uint8_t ECC_SECP256R1ExpectedSignature[] =
{
 0xf3, 0xac, 0x80, 0x61, 0xb5, 0x14, 0x79, 0x5b, 0x88, 0x43, 0xe3, 0xd6, 0x62, 0x95, 0x27, 0xed,
 0x2a, 0xfd, 0x6b, 0x1f, 0x6a, 0x55, 0x5a, 0x7a, 0xca, 0xbb, 0x5e, 0x6f, 0x79, 0xc8, 0xc2, 0xac,
 0x8b, 0xf7, 0x78, 0x19, 0xca, 0x05, 0xa6, 0xb2, 0x78, 0x6c, 0x76, 0x26, 0x2b, 0xf7, 0x37, 0x1c,
 0xef, 0x97, 0xb2, 0x18, 0xe9, 0x6f, 0x17, 0x5a, 0x3c, 0xcd, 0xda, 0x2a, 0xcc, 0x05, 0x89, 0x03
};
```

Figure 121. Provision the ECC_SECP256R1ExpectedSignaure Array

Similarly, the expected signature using the ECC private key which matches the updated ECC public key (see Figure 83) is provided in the array `ECC_SECP256R1ExpectedSignature_New` in `ecc_crypto_operation.c`.

```
/* This is an externally generated signature using the private key */
uint8_t ECC_SECP256R1ExpectedSignature_New[] =
{
 0x97, 0x6d, 0x3a, 0x4e, 0x9d, 0x23, 0x32, 0x6d, 0xc0, 0xba, 0xa9, 0xfa, 0x56, 0x0b, 0x7c, 0x4e,
 0x53, 0xf4, 0x28, 0x64, 0xf5, 0x08, 0x48, 0x3a, 0x64, 0x73, 0xb6, 0xa1, 0x10, 0x79, 0xb2, 0xdb,
 0x1b, 0x76, 0x6e, 0x9c, 0xeb, 0x71, 0xba, 0x6c, 0x01, 0xdc, 0xd4, 0x6e, 0x0a, 0xf4, 0x62, 0xcd,
 0x4c, 0xfa, 0x65, 0x2a, 0xe5, 0x01, 0x7d, 0x45, 0x55, 0xb8, 0xee, 0xef, 0xe3, 0x6e, 0x19, 0x32
};
```

Figure 122. Provision the ECC_SECP256R1ExpectedSignaure_New Array

There is no action needed from the user if the same sets of keys and plaintext messages are used. If new sets of keys and messages are used, the user needs to update the project with the new keys and messages.

7.3 FSP Crypto Module Support for User Key Update

This section introduces the FSP Crypto APIs for SCE Protected Mode that are used for secure user key updates. For a complete description of all FSP Crypto APIs, refer to the FSP User's Manual.

To use keys that have been injected via the secure key injection process using the MCUboot interfaces, the application must refer to those keys at the address where they were injected. If you inject keys at addresses other than those demonstrated above, be sure to change your application code to reflect those addresses. See instructions in section 7.4.

To perform a secure AES key update, use the following API to MCU-uniquely wrap a new AES key using a previously injected Key-Update Key:

```
fsp_err_t R_SCE_AES256_EncryptedKeyWrap (
  uint8_t *initial_vector,
  uint8_t *encrypted_key,
  sce_key_update_key_t *key_update_key,
  sce_aes_wrapped_key_t *wrapped_key )
```

The API parameters are:

- `[in] initial_vector`: Pointer to a buffer that holds the initialization vector that was used to wrap the new key. This must be the IV that was used during the key wrap process shown in section 5.1.4 or section. This value will be included in the generated `new_aes_key.c` and `new_aes_key.h`.
- `[in] encrypted_key`: Pointer to a buffer that holds the new key, wrapped by the KUK. In this example, it is the KUK-wrapped AES-256 key that was output during the key wrap process shown in section 5.1.4 or section 5.3.5. This value will be included in the generated `new_aes_key.c` and `new_aes_key.h`.
- `[in] key_update_key`: Pointer to the Key-Update Key that was previously injected on the MCU. This address must match the address used when injecting the KUK into section 5.4.2**.** The user needs to update the macro definition `KUK_ADDRESS` defined in `flash_storage.h` to match the injection address.
- `[in, out] wrapped_key`: This is the SRAM buffer to store the wrapped new user key. For security considerations, it is recommended to erase this buffer right after the wrapped key is saved to flash. In this application project, the newly generated wrapped key is stored in data flash and used in the example project.

To perform a secure ECC public key update, use the following API to MCU-uniquely wrap a new ECC public key using a previously injected Key-Update Key:

```
fsp_err_t R_SCE_ECC_secp256r1_EncryptedPublicKeyWrap (
  uint8_t * initial_vector,
  uint8_t *encrypted_key,
  sce_key_update_key_t *key_update_key,
  sce_ecc_public_wrapped_key_t *wrapped_key )
```

The API parameters are:

- `[in] initial_vector`: Pointer to a buffer that holds the initialization vector that was used to wrap the new key. This must be the IV that was used during the key wrap process shown in section 5.1.5 or section 5.3.5. This value will be included in the generated `new_ecc_public_key.c` and `new_ecc_public_key.h`.
- `[in] encrypted_key`: Pointer to a buffer that holds the new key, wrapped by the HUK. In this example, it is the KUK-wrapped ECC private key that was output during the key wrap process shown in section 5.1.5 or section 5.3.5. This value will be included in the generated `new_ecc_public_key.c` and `new_ecc_public_key.h`.
- `[in] key_update_key`: Pointer to the Key-Update Key that was previously injected on the MCU. This address must match the address used when injecting the KUK into section 5.4.2**.** The user needs to update the macro definition `KUK_ADDRESS` defined in `flash_storage.h` to match the injection address.
- `[in, out] wrapped_key`: This is the SRAM buffer to store the wrapped new user key. For security considerations, it is recommended to erase this buffer right after the wrapped key is saved to flash. In this application project, the newly generated wrapped key is stored in data flash and used in the example project.

7.3.1 Save the New Wrapped Key to Data Flash

Once a new key is wrapped, the user needs to use the flash driver `r_flash_hp` to manually store it in the data flash.

```
sce_aes_wrapped_key_t wrapped_new_user_key;
error = R_SCE_AES256_EncryptedKeyWrap (
    iv_encrypt_new_key, encrypted_new_key, &kuk_key, &wrapped_new_user_key );
```

Refer to function `store_new_aes_key_to_data_flash()` and function `store_new_ecc_pub_key_to_data_flash()` for the operations of storing the new wrapped keys to data flash.

7.4 Import and Compile the Example Project

Follow the steps below to exercise the example project. Note that there are sections of the code that must be updated using the secure key injection results generated above prior to compiling and running the project. Note that if the user has used the NIST vectors included in this application project for verification purposes, steps 4 to 5 can be skipped.

1. Launch e[2] studio and import `secure_key_inject_update_ra6m4.zip` file to a workspace.
2. At the bottom of `flash_storage.h`, find the macro definitions `DIRECT_AES_KEY_ADDRESS`, `DIRECT_ECC_PUB_KEY_ADDRESS`, and `KUK_ADDRESS` based on Figure 110.
3. Replace `new_aes_key.h and new_aes_key.c` with the new sets of files generated in section 5.1.4 or section 5.3.4 located in folder `\secure_key_inject_update_ra6m4_s\src\`.
4. Replace `new_ecc_public_key.c` and `new_ecc_public_key.h` generated in section 5.1.5 or section 5.3.5 located in folder `\secure_key_inject_update_ra6m4_s\src\`.
5. If different file names are used, update the `#include` definition in `aes_crypto_operations.c` on this line to reflect the new file name.

```
#include "crypto_operations.h"
#include "hal_data.h"
#include "r_sce.h"
#include "flash_storage.h"
#include "new_aes_key.h"
```

Figure 123. Include the Generated Header File for AES Operation

6. If different file names are used, update the `#include` definition in `ecc_crypto_operations.c` on this line to reflect the new file name.

```
#include <crypto_operations.h>
#include "hal_data.h"
#include "r_sce.h"
#include "flash_storage.h"
#include "new_ecc_public_key.h"
```

Figure 124. Include the Generated Header File for ECC Operation

7. Next, double-click `configuration.xml` from the secure project. Once the configurator is opened, click **Generate Project Content** and then compile the secure project.
8. Expand the non-secure project and double-click the `configuration.xml` file. Once the configurator is opened, click **Generate Project Content** and compile the non-secure project.

7.5 Running the Example Project

Prior to running the example project, the user is requested to remove Jumper J16 to put the MCU in Normal execution mode.

Once the source code compilation is successful, follow the steps below to exercise the example projects:

1. Choose to debug from the non-secure application. Right-click on `secure_key_inject_update_ra6m4_ns` and select **Debug As > Renesas GDB Hardware Debugging**.
2. Execution will halt at the secure project reset handler.

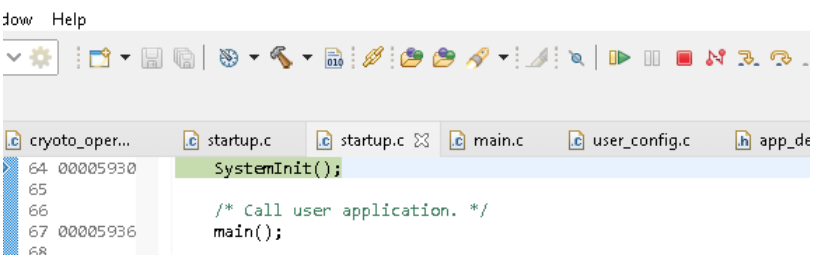


Figure 125. Running to the Secure Project Reset Handler

3. Click **Resume** twice to run the project.
4. Open the J-Link RTT Viewer with the settings shown below.

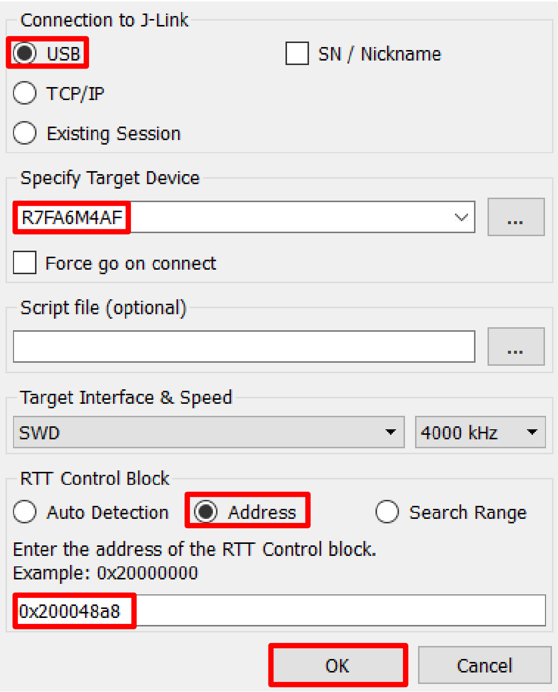


Figure 126. RTT Viewer Setting

Note: Block address for the variable in RAM called _SEGGER_RTT can be found in the .map file created in the build configuration (Debug) folder.

5. Click **OK**. The following menu should be printed.

```
00>  MENU to Select
00> Press 1 to exercise the cryptographic operation with directly injected AES key
00> Press 2 to create updated new AES key using FSP
00> Press 3 to exercise the cryptographic operation with newly updated AES key
00> Press 4 to exercise the cryptographic operation with directly injected ECC public key
00> Press 5 to create updated new ECC public key using FSP
00> Press 6 to exercise the cryptographic operation with newly updated ECC public key
```

Figure 127. Main RTT User Menu

A. Input **1** to confirm that the cipher text for the first AES key is successfully decrypted by the injected AES-256 key.

```
 < 1
00>  Cryptographic operation with directly injected AES key, which is injected via the serial interface.
00>
00> Result: Cryptographic operation is successful with directly injected AES key
```

Figure 128. Crypto Operation with Injected AES-256 Key

B. Input **2** to perform a key update to wrap the new AES-256 key and save the new key to data flash. Note that the SCE handles the wrapping of the new key internally without exposing the plaintext key. It is not possible to extract the plaintext key. The wrapped AES key in SRAM is deleted after storing it to the data flash. Note that if menu option '**1**' is rerun after menu item '**2**' is run, it will fail because the new AES key will not generate the same cipher text as the original key.

```
 < 2
00>  Update the new AES key encrypted with key update key and store the new wrapped key in data flash.
00>
00> Result: AES Key is updated and stored to Data Flash
```

Figure 129. Update the AES Key and Store to Data Flash

C. Input **3** to confirm that the cipher text for the second AES key is successfully decrypted by the updated AES-256 key.

```
 < 3
00>  Cryptographic operation with new wrapped AES key stored in data flash
00>
00>
00> Result: Cryptographic operation is successful with Updated AES Key
```

Figure 130. Crypto Operation with the New AES Key

D. Input **4** to confirm that the signature generated using the first ECC private key is successfully verified by the injected ECC public key.

```
 < 4
00>  Cryptographic operation with directly injected ECC public key, which is injected via the serial interface.
00>
00> Result: Cryptographic operation is successful with directly injected ECC public key
```

Figure 131. Crypto Operation with Injected ECC Public Key

E. Input **5** to perform a key update to wrap the new ECC public key and save the new key to data flash. Note that the SCE handles the wrapping of the new key internally without exposing the plaintext key. It is not possible to extract the plaintext key. The wrapped ECC public key in SRAM is deleted after storing to data flash. Note that if menu option '**4**' is rerun after menu item '**5**' is run, it will fail because the new ECC public key cannot verify a signature that was generated by the first key's private key.

```
 < 5
00>  Update the new ECC public key encrypted with key update key and store the new wrapped key in data flash.
00>
00> Result: ECC public Key is updated and stored to Data Flash
```

Figure 132. Update the ECC Public Key and Store to Data Flash

F. Input **6** to confirm that the signature generated using the second ECC private key is successfully verified by the updated ECC public key.

```
 < 6
00>  Cryptographic operation with new wrapped ECC public key stored in data flash
00>
00>
00> Result: Cryptographic operation is successful with Updated ECC public Key
```

Figure 133. Crypto Operation with the New ECC Public Key

Successful operations of the above menu items conclude the demonstration of the secure key injection and update in this application project.

8. Example Project for RA8P1 (RSIP Protected Mode)

This section introduces RSIP Protected Mode with an example of wrapped ECC key pair injection and update.

To exercise the example projects as is, users can follow the steps below:

- Inject the included example RFP injection keys (`KUK.rkey`, `ECC_Public_Key.rkey` and `ECC_Private_Key.rkey` which are included in `rfp_resource_ra8p1.zip`) by following section 5.4.2.
- A set of new user keys (ECC Public and Private Key) generated using the example KUK is already provisioned in the example projects. The user can then directly proceed to exercise the example project.

Note: Please do not use the example keys for production support.

To use the example projects with customized keys, the user can follow the steps below:

- To test customized RFP injection keys and new user update keys (generated by the following section 5.2 or 5.3 rather than using the ones included in `rfp_resources_ra8p1.zip`), the user needs to follow section 1.1 to inject the keys to the MCU. User also needs to generate customized new user key files (`new_ecc_public_key.c/.h` and `new_ecc_private_key.c/.h`) with the same key name to replace the corresponding files used in the example project. Once the example projects are updated, the user can proceed to run the example projects to verify the operations.
- To test the new user key update procedure only, the user can use the included RFP `KUK.rkey` file to generate new source files to replace the corresponding files in the example project. Once the example projects are updated, the user can then proceed to the verification of the operations.

8.1 Example Project Overview

This example project demonstrates the following functionalities of the protected mode of RSIP-E50D:

- Use the injected ECC private key to sign the NIST test message shown in Figure 71 then use the injected ECC public key to verify signature created by corresponding private key.
- The injected Key-Update Key (KUK) is used to inject the newly wrapped ECC public key and store this new ECC public key and new ECC private key for MRAM.
- Use the new ECC private key to sign the NIST test message shown in Figure 83 then use the new ECC public key to verify signature created by corresponding private key.

8.2 Using the RFP Injected Keys

8.2.1 Formatting the Injected Keys

The keys that are injected into the MCU flash using RFP cannot be used directly by the FSP Crypto APIs. A minor formatting change is required.

8.2.1.1 Formatting the Injected ECC Key Pair

The following code snippet stores information of the ECC public key and ECC private key including the key type and its value. The destination variables can then be used for cryptographic operations. Replace the macro `DIRECT_ECC_PUB_KEY_ADDRESS` and `DIRECT_ECC_PRI_KEY_ADDRESS` with the actual injection address.

```
static rsip_wrapped_key_t ecc_public_key_injected =
{
    .type = RSIP_KEY_TYPE_ECC_SECP256R1_PUBLIC, .p_value = (void *)DIRECT_ECC_PUB_KEY_ADDRESS
};

static rsip_wrapped_key_t ecc_private_key_injected =
{
    .type = RSIP_KEY_TYPE_ECC_SECP256R1_PRIVATE, .p_value = (void *)DIRECT_ECC_PRI_KEY_ADDRESS
};
```

8.2.1.2 Formatting the Injected KUK

The following code snippet stores information of injected KUK including the key type and its value. The destination varlable can then be used for secure key updates. Replace the macro `KUK_ADDRESS` with the actual injection address.

```
static rsip_wrapped_key_t kuk_key =
{
    .type = RSIP_KEY_TYPE_KUK, .p_value = (void *)KUK_ADDRESS
};
```

8.2.2 Verifying the Injected Key and the Updated Key

To verify the ECC key pair injection, a signature generated by using the ECC private key should be successfully verified using the injected ECC public key, as the key pair is derived from the same NIST test vector.

There is no action needed from the user if the same sets of keys are used. If new sets of keys are used, the user needs to update the project with the new keys.

8.3 FSP Crypto Module Support for User Key Update

This section introduces the FSP Crypto APIs for RSIP Protected Mode that are used for secure user key updates. For a complete description of all FSP Crypto APIs, refer to the FSP User's Manual.

To use keys that have been injected via the secure key injection process using the MCUboot interfaces, the application must refer to those keys at the address where they were injected. If you inject keys at addresses other than those demonstrated above, be sure to change your application code to reflect those addresses. See instructions in section 8.4.

To perform a secure ECC public key and ECC private key update, use the following API to MCU-uniquely wrap a new key using a previously injected Key-Update Key:

```
fsp_err_t R_RSIP_EncryptedKeyWrap (
   rsip_ctrl_t * const              p_ctrl,
   rsip_wrapped_key_t const * const p_key_update_key,
   void const * const               p_initial_vector,
   void const * const               p_encrypted_key
   rsip_wrapped_key_t * const       p_wrapped_key)
```

The API parameters are:

- `[in, out] p_ctrl`: Pointer to control block.
- `[in] p_key_update_key`: Pointer to the Key-Update Key that was previously injected on the MCU. This address must match the address used when injecting the KUK into section 1.1**.** The user needs to update the macro definition `KUK_ADDRESS` defined in `mram_storage.h` to match the injection address.
- `[in] p_initial_vector`: Pointer to a buffer that holds the initialization vector that was used to wrap the new key. This must be the IV that was used during the key wrap process shown in section 5.2.2. This value will be included in the generated `new_ecc_public_key.c` and `new_ecc_public_key.h`.
- `[in] p_encrypted_key`: Pointer to a buffer that holds the new key, wrapped by the HUK. In this example, it is the KUK-wrapped ECC private key that was output during the key wrap process shown in section 5.2.2. This value will be included in the generated `new_ecc_public_key.c` and `new_ecc_public_key.h`.
- `[in, out] wrapped_key`: This is the SRAM buffer to store the wrapped new user key. For security considerations, it is recommended to erase this buffer right after the wrapped key is saved to MRAM. In this application project, the newly generated wrapped key pair is stored in MRAM and used in the example project.

8.3.1 Save the New Wrapped Key to MRAM

Once a new key is wrapped, the user needs to use the mram driver `r_mram` to manually store it in the MRAM.

Refer to function `store_new_ecc_key_pair_to_mram()` for the operations of storing the new wrapped key pair to MRAM.

8.4 Import and Compile the Example Project

Follow the steps below to exercise the example project. Note that there are sections of the code that must be updated using the secure key injection results generated above prior to compiling and running the project. Note that if the user has used the NIST vectors included in this application project for verification purposes, steps 4 to 5 can be skipped.

1. Launch e[2] studio and import `secure_key_inject_update_ra8p1.zip` file to a workspace.
2. At the top of `mram_storage.h`, find the macro definitions `DIRECT_ECC_PUB_KEY_ADDRESS`, `DIRECT_ECC_PRI_KEY_ADDRESS` and `KUK_ADDRESS` based on Figure 111.
3. Replace `new_ecc_public_key.c` and `new_ecc_public_key.h` generated in section 5.2.2 or section 5.3 located in folder `\secure_key_inject_update_8p1\src\`.
4. If different file names are used, update the `#include` definition in `ecc_crypto_operations.c` on this line to reflect the new file name.

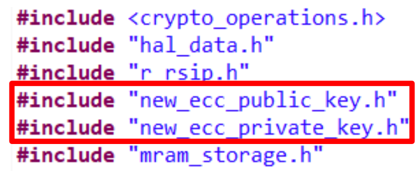


Figure 134. Include the Generated Header File for ECC Operation

5. Next, double-click `configuration.xml`. Once the configurator is opened, click **Generate Project Content** and then compile the project.

8.5 Running the Example Project

Once the source code compilation is successful, follow the steps below to exercise the example projects:

1. Choose to debug from the non-secure application. Right-click on `secure_key_inject_update_ra8p1` and select **Debug As > Renesas GDB Hardware Debugging**.
2. Execution will halt at the secure project reset handler.

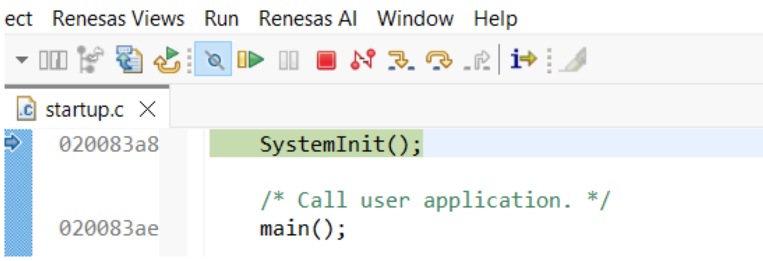


Figure 135. Running to the Secure Project Reset Handler

3. Click **Resume** twice to run the project.
4. Open the J-Link RTT Viewer with the settings shown below.

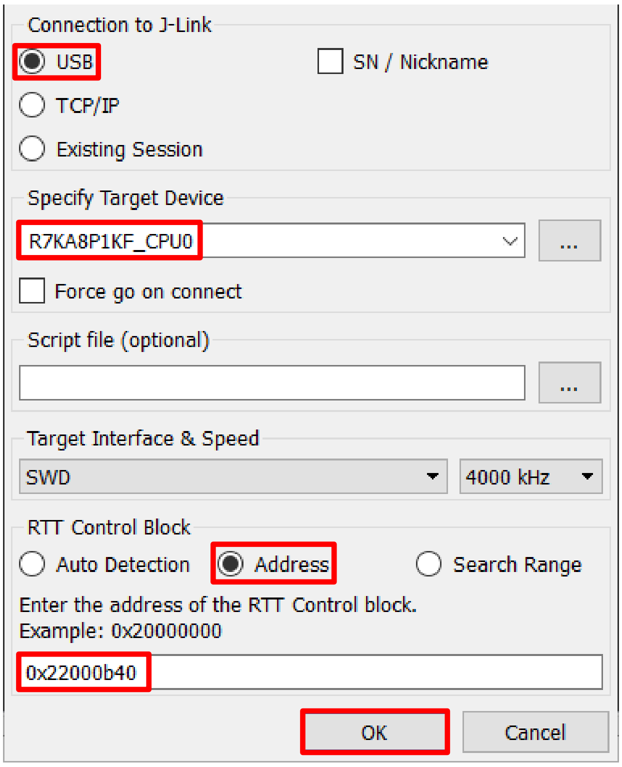


Figure 136. RTT Viewer Setting

Note: Block address for the variable in RAM called _SEGGER_RTT can be found in the .map file created in the build configuration (Debug) folder.

5. Click **OK**. The following menu should be printed.

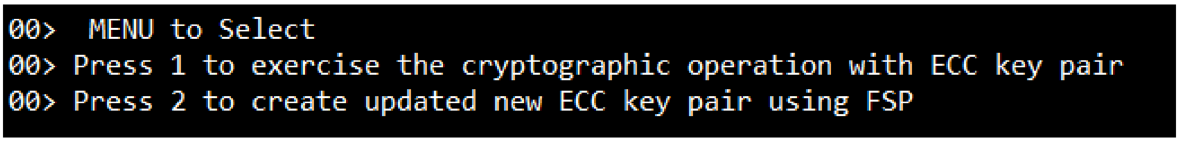


Figure 137. Main RTT User Menu

A. Input **1** to confirm that the signature generated using the initial injected ECC private key is successfully verified by the injected ECC public key.

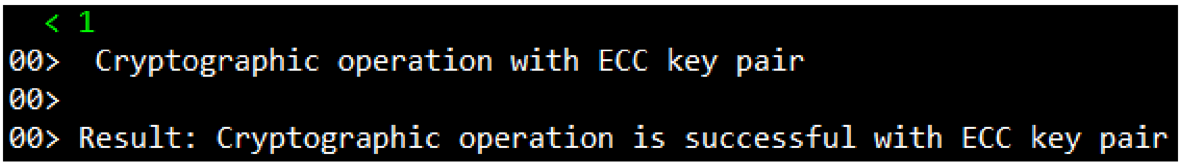


Figure 138. Crypto Operation with Injected ECC Public Key

B. Input **2** to perform a key update to wrap the new ECC key pair and save the new key to MRAM. Note that the RSIP handles the wrapping of the new key internally without exposing the plaintext key. It is not possible to extract the plaintext key. The wrapped ECC public key in SRAM is deleted after storing it into MRAM.

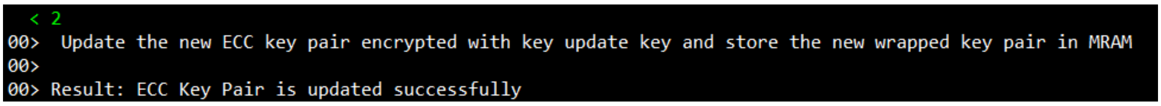


Figure 139. Update the ECC Key Pair and Store to MRAM

C. Input **1** to confirm that the signature generated using the updated injected ECC private key is successfully verified by the updated injected ECC public key.

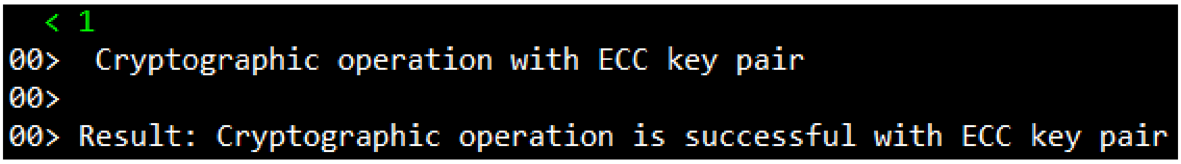


Figure 140. Crypto Operation with the New ECC Public Key

Successful operations of the above menu items conclude the demonstration of the secure key injection and update in this application project

9. Example Project for RA8M1 (RSIP Compatibility Mode)

This section introduces RSIP Compatibility Mode with an example of AES-128 user key injection and update.

9.1 Overview

This example project demonstrates the following functionalities of the compatibility mode of RSIP-E51A.

- AES-128 key injection using the files generated in section 6.1.
- Verifying the injected AES-128 key using PSA Crypto APIs and a NIST AES test vector.

9.2 Using the SKMT-Generated Files

The source files generated from Figure 116 are included in the example project. These files provide the UFPK-wrapped user key information used to demonstrate the functionality described in section .

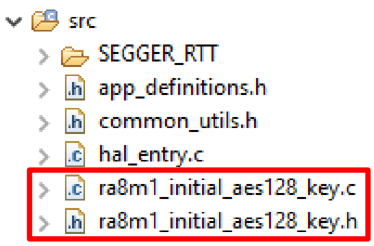


Figure 141. R8M1 Example Project Source Code

9.3 RSIP Compatibility Mode Key Injection APIs

This demonstration uses the APIs in the Key Injection module (`r_rsip_key_injection`) to perform key injection. Refer to the FSP User Manual for the complete list of key injection APIs and their parameters.

9.4 Import and Compile the Example Project

Note that if AES keys other than the NIST vectors are used, then those new source files need to replace the existing files in the example project prior to compiling and running the example project. If the NIST vectors included in this application project are being used for verification purposes, step 2 can be skipped.

1. Launch e^2 studio and import `secure_key_inject_ra8m1.zip` file to a workspace.
2. Replace `ra8m1_initial_aes128_key.h` and `ra8m1_initial_aes128_key.c` with the new set of files generated in Figure 119.

3. If different file names are used, update the `#include` definition in `hal_entry.c` on this line to reflect the new file name.

```
#include "hal_data.h"
#include "common_utils.h"
#include "app_definitions.h"
#include "hw_sce_ra_private.h"
#include "ra8m1_initial_aes128_key.h"
```

Figure 142. Include the Generated Header File for AES Operation

4. Next, double-click `configuration.xml`. Once the Configurator is opened, click **Generate Project Content** and then compile the secure project.

9.5 Running the Example Project

Follow the steps below to exercise the example projects:

1. Right-click on `secure_key_inject_ra8m1` and select **Debug As > Renesas GDB Hardware Debugging**.
2. Execution will halt at the reset handler.

```
          void Reset_Handler (void)
          {
              /* Initialize system using BSP. */
02006988      SystemInit();

              /* Call user application. */
0200698e      main();

02006992      while (1)
              {
                  /* Infinite Loop. */
              }
          }
```

Figure 143. Running to the Project Reset Handler

3. Click **Resume** twice to run the project.
4. Open the J-Link RTT Viewer with the settings shown below.

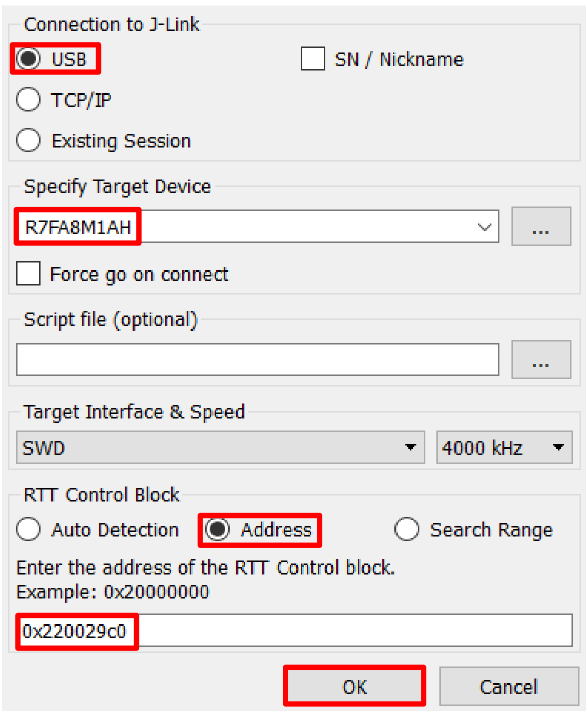


Figure 144. RTT Viewer Setting

Note: Block address for the variable in RAM called _SEGGER_RTT can be found in the .map file created in the build configuration (Debug) folder.

5. Click **OK**. The following execution result should be printed. Users can step into the code to understand the code execution flow.

```
00>
00> Result: Initial AES 128 Key Wrap is successful
00>
00>
00> Result: Cryptographic operation is successful with initial wrapped AES 128 key
00>
```

Figure 145. Execution Result - Secure Key Injection Example Project for RA8M1

10. Example Project for RA6M3 (SCE Compatibility Mode)

This section introduces SCE Compatibility Mode with an example of AES-128 user key injection and update.

10.1 Overview

This example project demonstrates the following functionalities of the compatibility mode of SCE7:

- AES-128 key injection using the files generated in section 6.2.
- Verifying the injected AES-128 key using PSA Crypto APIs and a NIST AES test vector.

10.2 Using the SKMT-Generated Files

The source files generated Section 6.2 from Figure 119 are included in the example project. These files provide the UFPK-wrapped AES key source files used to demonstrate the functionality described above.

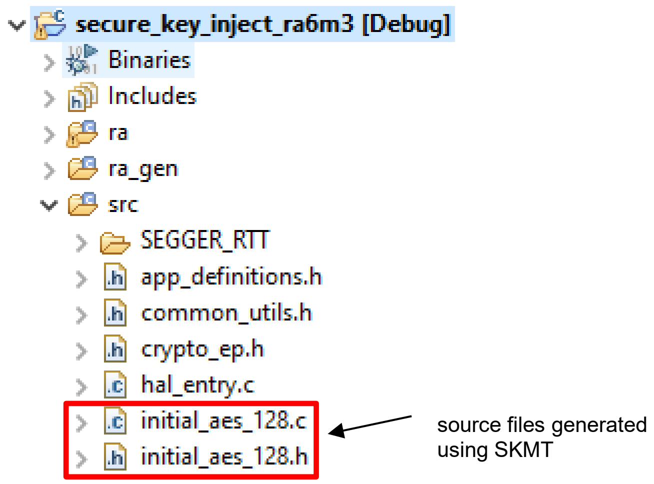


Figure 146. RA6M3 Example Project Source Code

10.3 SCE7 Compatibility Mode Key Injection APIs

This demonstration uses the APIs in the Key Injection module (`r_sce_key_injection`) to perform key injection. Refer to the FSP User Manual for the complete list of key injection APIs and their parameters.

10.4 Import and Compile the Example Project

Note that if AES keys other than the NIST vectors are used, then those new source files need to replace the existing files in the example project prior to compiling and running the example project. If the NIST vectors included in this application project are being used for verification purposes, steps 2 to 5 can be skipped.

1. Launch e^2 studio and import `secure_key_inject_ra6m3.zip` file to a workspace.
2. Replace `initial_aes_128.h` and `initial_aes_128.c` with the new set of files generated in Figure 119.

3. If different file names are used, update the `#include` definition in `hal_entry.c` on this line to reflect the new file name.

```
#include "hal_data.h"
#include "common_utils.h"
#include "crypto_ep.h"
#include "app_definitions.h"
#include "hw_sce_ra_private.h"
#include "initial_aes_128.h"
```

Figure 147. Include the Generated Header File for AES Operation

4. Next, double-click `configuration.xml`. Once the configurator is opened, click **Generate Project Content** and then compile the project.

10.5 Running the Example Project

Follow the steps below to exercise the example projects:

1. Right-click on `secure_key_injection_ra6m3` and select **Debug As > Renesas GDB Hardware Debugging**.
2. Execution will halt at the reset handler.

```
    SystemInit();

    /* Call user application. */
    main();

    while (1)
    {
        /* Infinite Loop. */
    }
}
```

Figure 148. Running to the Project Reset Handler

3. Click **Resume** twice to run the project.
4. Open the J-Link RTT Viewer with the settings shown below.

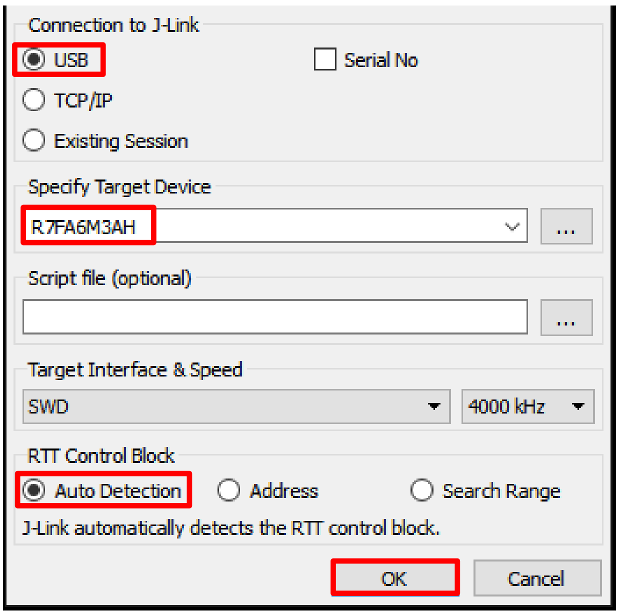


Figure 149. RTT Viewer Setting

Note: Block address for the variable in RAM called _SEGGER_RTT can be found in the .map file created in the build configuration (Debug) folder.

5. Click **OK**. The following execution result should be printed. Users can step into the code to understand the code execution flow.

```
00>
00> Result: AES 128 Key Wrap is successful
00>
00> Result: Cryptographic operation is successful with wrapped AES 128 key
```

Figure 150. Execution Result - Secure Key Injection for Example Project RA6M3

11. References

1. Renesas RA Family Device Lifecycle Management Key Injection Application Note (R11AN0469)
2. Renesas RA Family Secure Crypto Engine Operational Modes Application Note (R11AN0498)
3. Renesas RA Family MCU Security Design with TrustZone® – IP Protection (R11AN0467)
4. Renesas RA Family MCU Plaintext Key Injection (R11AN0473)

12. Website and Support

Visit the following URLs to learn about the RA family of microcontrollers, download tools and documentation, and get support.

EK-RA6M4 Resources	renesas.com/ra/ek-ra6m4
EK-RA8M1 Resources	renesas.com/ra/ek-ra8m1
EK-RA8P1 Resources	renesas.com/ra/ek-ra8p1
EK-RA6M3 Resources	renesas.com/ra/ek-ra6m3
RA Product Information	renesas.com/ra
Flexible Software Package (FSP)	renesas.com/ra/fsp
RA Product Support Forum	renesas.com/ra/forum
Renesas Support	renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	May.19.21	-	First release document
1.10	Jan.27.22	-	Update to use Security Key Management Tool CLI V1.0.0
1.20	Mar.25.22	-	Updated to add SKMT GUI support
1.30	Oct.25.22	-	Update to support SCE7 with FSP v4.0.0
2.00	Jan.03.24	-	Update to FSP v5.1.0
2.10	Oct.15.24	-	Update to FSP v5.5.0
2.20	Jul.16.25	-	Update to FSP v6.0.0
2.21	Oct.27.25	1	Add RSIP-E50D support on RA8x2

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.