GR5xx Firmware Encryption Application Note

Version: 3.0

Release Date: 2023-03-30

Shenzhen Goodix Technology Co., Ltd.

Trademarks and Permissions

GOODiX and other Goodix trademarks are trademarks of Shenzhen Goodix Technology Co., Ltd. All other trademarks and trade names mentioned in this document are the property of their respective holders.

Disclaimer

Information contained in this document is intended for your convenience only and is subject to change without prior notice. It is your responsibility to ensure its application complies with technical specifications.

Shenzhen Goodix Technology Co., Ltd. (hereafter referred to as “Goodix”) makes no representation or guarantee for this information, express or implied, oral or written, statutory or otherwise, including but not limited to representation or guarantee for its application, quality, performance, merchantability or fitness for a particular purpose. Goodix shall assume no responsibility for this information and relevant consequences arising out of the use of such information.

Without written consent of Goodix, it is prohibited to use Goodix products as critical components in any life support system. Under the protection of Goodix intellectual property rights, no license may be transferred implicitly or by any other means.

Shenzhen Goodix Technology Co., Ltd.

Headquarters: Floor 12-13, Phase B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 Zip Code: 518000

Website: www.goodix.com

Preface

Purpose

This document introduces the firmware encryption and decryption mechanisms, message verification, and digital signatures of the security modules of Bluetooth Low Energy (Bluetooth LE) GR5xx System-on-Chips (SoCs), to help developers understand and apply the security mode of GR5xx SoCs.

Audience

This document is intended for:

- GR5xx user
- GR5xx developer
- GR5xx tester
- Hobbyist developer
- Technical writer

Release Notes

This document is the second release of *GR5xx Firmware Encryption Application Note*, corresponding to Bluetooth LE GR5xx SoC series.

Revision History

Version	Date	Description
1.0	2023-01-10	Initial release
3.0	2023-03-30	Updated descriptions about GR5xx SoCs.

Contents

Preface I

1 Introduction 1

1.1 Security Solution 1

1.2 Features 2

2 Security Modules and Algorithms 4

2.1 Basics 4

2.1.1 Cryptosystem 4

2.1.2 Message Authentication 4

2.1.3 Digital Signature 4

2.2 Security Modules 4

2.3 Security Algorithms 5

2.3.1 ECIES 6

2.3.2 ECC 6

2.3.3 PRESENT-128 6

2.3.4 HMAC-SHA256 6

2.3.5 RSASSA-PSS 7

3 Message Verification in System Configuration Area 8

3.1 HMAC Verification Process 8

4 Firmware and Data Security 10

4.1 Firmware Encryption and Decryption 10

4.2 Data Encryption and Decryption 11

5 Digital Signature 13

5.1 Signing Process 13

5.2 Verification Process 14

6 Security Mode in Application 16

6.1 Firmware Programming and Encryption with GProgrammer 16

6.2 User-defined Encryption with GRPLT Lite Config Tool 16

6.3 SWD Interface 17

6.3.1 Register 17

6.3.2 APIs 18

7 FAQ 19

7.1 Why Do I Fail to Execute the Firmware Files? 19

7.2 Why Do Firmware Encryption and Signing Fail? 19

7.3 How to Manage Key Information and Encrypted Firmware? 19

1 Introduction

GR5xx System-on-Chips (SoCs) are Goodix high-security-performance Bluetooth Low Energy (Bluetooth LE) SoCs that precisely meet the growing high security demand from the rapidly evolving IoT technology. GR5xx SoCs effectively ensure the security of firmware, data, and system. User data and firmware in memories are protected against eavesdropping, thanks to the combination of security algorithms and security modules.

This document introduces firmware encryption for GR5xx SoCs by principle, implementation methods, and application.

- "Chapter 2 Security Modules and Algorithms" introduces the security modules and the security algorithms of GR5xx SoCs.
- "Chapter 3 Message Verification in System Configuration Area", "Chapter 4 Firmware and Data Security", and "Chapter 5 Digital Signature" introduce how to establish the security mode.
- "Chapter 6 Security Mode in Application" introduces the application of security mode with complementary tools.

1.1 Security Solution

GR5xx series adopts a complete security solution that integrates embedded encryption engine and complementary tools. Users can make security configurations on GR5xx SoCs with tools (such as GProgrammer). The embedded ROM will then launch secure boot to protect the SoC.

The security solution mainly includes message verification of System Configuration Area (SCA), encryption and decryption of firmware and data, and digital signature, which are detailed in "Chapter 3 Message Verification in System Configuration Area", "Chapter 4 Firmware and Data Security", and "Chapter 5 Digital Signature".

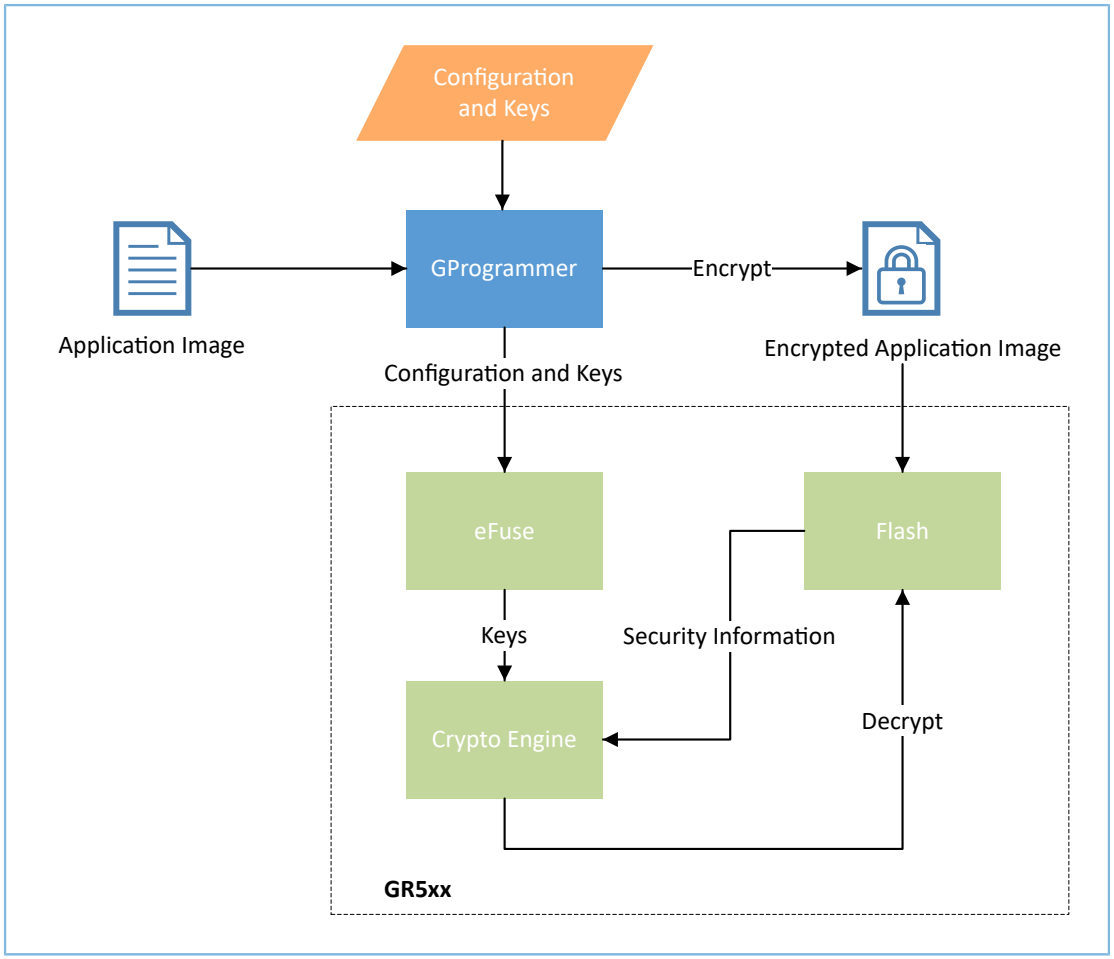

Figure 1-1 Security system block diagram

- Application Image is an executable binary file (often in HEX or BIN) generated from compilation.
- GProgrammer is a firmware programming tool that applies to GR5xx SoCs, including making security configurations on the SoCs.
- Encrypted Application Image is encrypted firmware (often in BIN) from tools (GProgrammer). Its security information can be used for chip verification.

1.2 Features

- Secure key storage

  Keys are stored in eFuse, which means the key information in eFuse cannot be directly accessed through MCU. In security mode, GR5xx SoCs load the key information to the KEYRAM module in encryption, and true random numbers are generated as masks in this process. The key information in the KEYRAM module cannot be directly

obtained by MCU. When key information is required by security modules, an independent hardware unit in the SoC exports the keys to the modules automatically.

- Preventing firmware from being eavesdropped

  GR5xx combines the efficiency of a symmetric-key cryptosystem with the convenience of a public-key cryptosystem. For decryption, elliptic curve cryptography (ECC) algorithm shall be used in combination with the private key stored in eFuse, to calculate the keys required for decrypting the PRESENT-128 module. Even if eavesdroppers obtain the encrypted firmware stored in Flash memories, they cannot use the firmware because they cannot obtain the key.

- Preventing malicious attacks

  To prevent malicious attacks, GR5xx provides Serial Wire Debug (SWD) locks and secure device firmware update (DFU) for firmware stored in Flash memories. At the stage of mass production, SWD can be disabled by modifying the configuration information stored in eFuse, to prevent the SoC firmware information from being read or modified. When SWD is disabled, users can also upgrade firmware, through DFU and the App Bootloader process, during which the encrypted firmware is verified, to prevent the firmware from being maliciously modified.

- One data key for one device

  Firmware keys and data keys are stored separately in different zones in eFuse. The same firmware can be programmed in SoCs with different data keys, so that one device owns its unique data key.

2 Security Modules and Algorithms

This chapter introduces the security modules and the security algorithms of GR5xx SoCs.

2.1 Basics

2.1.1 Cryptosystem

- Symmetric cryptosystem: also known as private-key encryption/decryption, in which the same key is used to cipher (by the sender) and decipher (by the receiver) data
- Public-key cryptosystem: also known as asymmetric encryption/decryption. It uses a pair of keys: a public key and a private key, with each used for encryption/decryption. For example, if the public key is used to encrypt a message, the private key will be used to decrypt the message.
- Hybrid cryptosystem: It combines the efficiency of a symmetric cryptosystem with the convenience of a public-key cryptosystem. Symmetric cryptosystem boosts the efficiency of encryption and decryption, and public-key cryptosystem helps distribute keys.

2.1.2 Message Authentication

- Digital digest: Turns messages at different lengths into short messages at a certain length, often used with one-way hash function.
- Message authentication code (MAC): a message authentication mechanism which not only examines whether a message is tampered with, but also checks whether a message comes from an expected communication object

2.1.3 Digital Signature

Digital signature: The strings, which can only be generated by the message sender and cannot be fabricated by other parties, can be used as valid credentials for the authenticity of a sent message. Compared with MAC, digital signature stands out for its non-repudiation.

2.2 Security Modules

The security modules for encryption in a GR5xx SoC include TRNG, PRESENT-128, eFuse, KEYRAM, PKC, HMAC, and XIP_DEC.

- **True Random Number Generator (TRNG) module**

  The TRNG module generates the random numbers used as masks in encryption and decryption. To ensure the quality of the random numbers and to correct the deviations in TRNG, linear-feedback shift registers (LFSRs) and Post-Process logics are added to the TRNG module.

- **PRESENT-128 module**

  PRESENT is a type of lightweight block cipher, featuring a compact size of algorithm (approximately 40% of the size of AES). It can be applied in scenarios that require low energy and high efficiency.

  The PRESENT-128 module enables encrypting or decrypting 128-bit data in one operation, supported by its two 64-bit PRESENT cores.

- **eFuse module**

  eFuse is a 512-byte one-time programmable (OTP) memory with random access interfaces, which stores security keys and chip calibration data.

- **KEYRAM module**

  KEYRAM is mainly applied for key derivation and storage after a chip is powered on. In the secure boot process, true random numbers are generated as masks in each boot, to prevent decryption by others through proof by exhaustion. Security modules (such as AES, HMAC, and XIP_DEC) can read keys under encryption through the KeyPort bus. The keys stored in the KEYRAM module cannot be accessed through CPU or debugging ports.

- **Public Key Cryptography (PKC) module**

  The PKC controller module focuses on basic modular arithmetic in public-key algorithms and 256-point elliptic curve cryptography (ECC) point multiplication, according to Federal Information Processing Standards (FIPS).

- **Hash Message Authentication Code (HMAC) module**

  The HMAC module authenticates and validates messages with HMAC algorithm in full compliance with FIPS Publication 198-1. The HMAC module supports SHA256 and HMAC-SHA256.

- **XIP_DEC module**

  The module is embedded with a PRESENT-128 submodule, so that firmware commands or data can be read for real-time decryption in execute in place (XIP) mode.

Note:

For more information about the XIP_DEC module, see the datasheet of the specific GR5xx SoC.

2.3 Security Algorithms

The security algorithms that enhance GR5xx SoCs are Elliptic Curve Integrated Encryption Scheme (ECIES) (ECC P-256 and PRESENT-128), HMAC-SHA256, and PKCS#1 V2.1 RSASSA-PSS.

Table 2-1 Security algorithms

Application Scenario	Security Algorithm	Key Size (Bit)
Message verification in SCA	HMAC-SHA256	256
Firmware encryption and decryption	ECIES (ECC P-256)	Private key: 256 Public key: 256 x 2 + (8)
	ECIES (PRESENT-128)	128
Data encryption and decryption	PRESENT-128	128

Application Scenario	Security Algorithm	Key Size (Bit)
Digital signature	PKCS#1 V2.1 RSASSA-PSS	Private key: 2048 Public key: 2048 + 32 + (2048 + 32)

2.3.1 ECIES

Integrated Encryption Scheme (IES) is a hybrid encryption scheme. One variant of IES is ECIES. It requires obtaining the shared secret through elliptic-curve Diffie-Hellman (ECDH), and then generating an independent symmetric key and a MAC key by using a key derivation function (KDF). The keys can be used for encryption/decryption and message authentication.

The ECIES hybrid encryption scheme is based on P-256 elliptic curve obtaining PRESENT-128 key. This security algorithm distributes keys and boosts efficiency in encryption/decryption.

2.3.2 ECC

ECC is an approach to public key cryptography based on the algebraic structure of elliptic curves over finite fields. Mathematically, this cryptography is based on the difficulty in computing the discrete logarithms in an Abelian group consisting of rational points on the elliptic curve. ECC stands out for allowing smaller keys compared to non-EC cryptography while providing equivalent or higher security in certain circumstances.

Table 2-2 Key size of RSA/DSA and ECC at equivalent security level (unit: bit)

RSA/DSA	512	768	1024	2048	21000
ECC	106	132	160	211	600

Compared with RSA, ECC excels in:

- Better security guarantee: Security performance of the 160-bit ECC equals that of 1024-bit RSA or 1024-bit DSA.
- Higher processing speed: ECC stands out in the speed of processing private keys, which is much faster than that of RSA or DSA.
- Lower bandwidth and less memory occupation: Compared with RSA and DSA, ECC excels in smaller key size and system parameters.

2.3.3 PRESENT-128

PRESENT-128 is a lightweight block cipher based on substitution–permutation network (SPN) structure. It operates on 64-bit blocks for 31 rounds; the key size is 128 bits. Compared with other lightweight block ciphers such as TEA, MCRYTON, HIGHT, SEA, and CGEN, PRESENT-128 features simpler hardware implementation and a concise way to implement round functions.

2.3.4 HMAC-SHA256

HMAC-SHA256 is a type of keyed hash algorithm that is constructed from the SHA-256 hash function. According to HMAC and SHA-256 algorithms, plaintext of HMAC-SHA256 shall be in 512-bit unit. Users can obtain 256-bit MACs with keys at any size (recommended: 256 bits or larger) by using HMAC-SHA256.

2.3.5 RSASSA-PSS

RSA digital signature algorithm (RSASSA) is a kind of RSA encryption/decryption algorithm; Probabilistic Signature Scheme (PSS) is a padding scheme for private key signature. In essence, RSASSA-PSS can be regarded as RSA encryption used together with a padding scheme.

- Padding schemes: When PKCS#1 V1.5 is used, the output of the same message/key is the same; in contrast, when PKCS#1 PSS is used, the output of the same message/key varies due to true random numbers are used. But the corresponding public key works for verification in both signature algorithms.
- Security: When PKCS#1 V1.5 is used, the public exponent (e) is often 65537. This is because if the public exponent is too short (for example, e = 3), the encryption scheme will become vulnerable to ciphertext attacks (ciphertext may still comply with padding specifications after being tampered with). However, when the padding scheme is PKCS#1 PSS, the public exponent being 3 is no longer an issue.

3 Message Verification in System Configuration Area

SCA is in the first two sectors (8 KB in total; address: 0x0020_0000 to 0x0020_2000; the address for GR551x is 0x0100_0000 to 0x0100_2000) of Flash memory. It stores flags and other system configuration parameters used during system boot.

This chapter introduces the process for verifying HMAC in the SCA of GR5xx SoCs.

Note:

For details about SCA, refer to the developer guide of the specific GR5xx SoC.

3.1 HMAC Verification Process

When SCA is updated in security mode through GProgrammer or other tools, HMAC in SCA will be updated synchronously. In secure boot, GR5xx SoC first verifies HMAC in SCA. The system will proceed with the secure boot process (decryption and signature verification) if verification succeeds; otherwise, the system will enter DFU mode.

An HMAC verification process is shown in Figure 3-1.

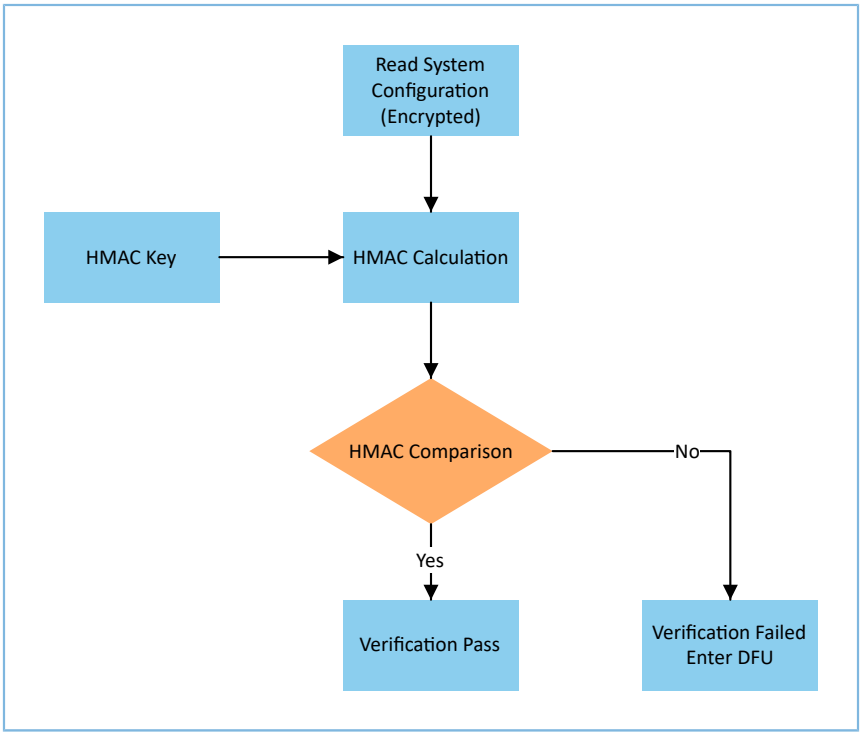

Figure 3-1 HMAC verification process

To perform HMAC verification,

1. Read SCA data (encrypted data).
2. Start HMAC calculation (HMAC-SHA256) on the SCA data (expect the HMAC data of the SCA) with HMAC keys in eFuse.

3. Compare the calculation result with the HMAC data of the SCA. If the two are the same, the verification succeeds.

4 Firmware and Data Security

4.1 Firmware Encryption and Decryption

GR5xx series adopts ECIES, a hybrid encryption scheme that ensures firmware security and efficiency by integrating symmetric encryption (PRESENT-128) and public key cipher (elliptic-curve cryptography, ECC), combining the strength of the two security mechanisms.

The encryption and decryption processes of hybrid cryptosystem are shown in the figure below.

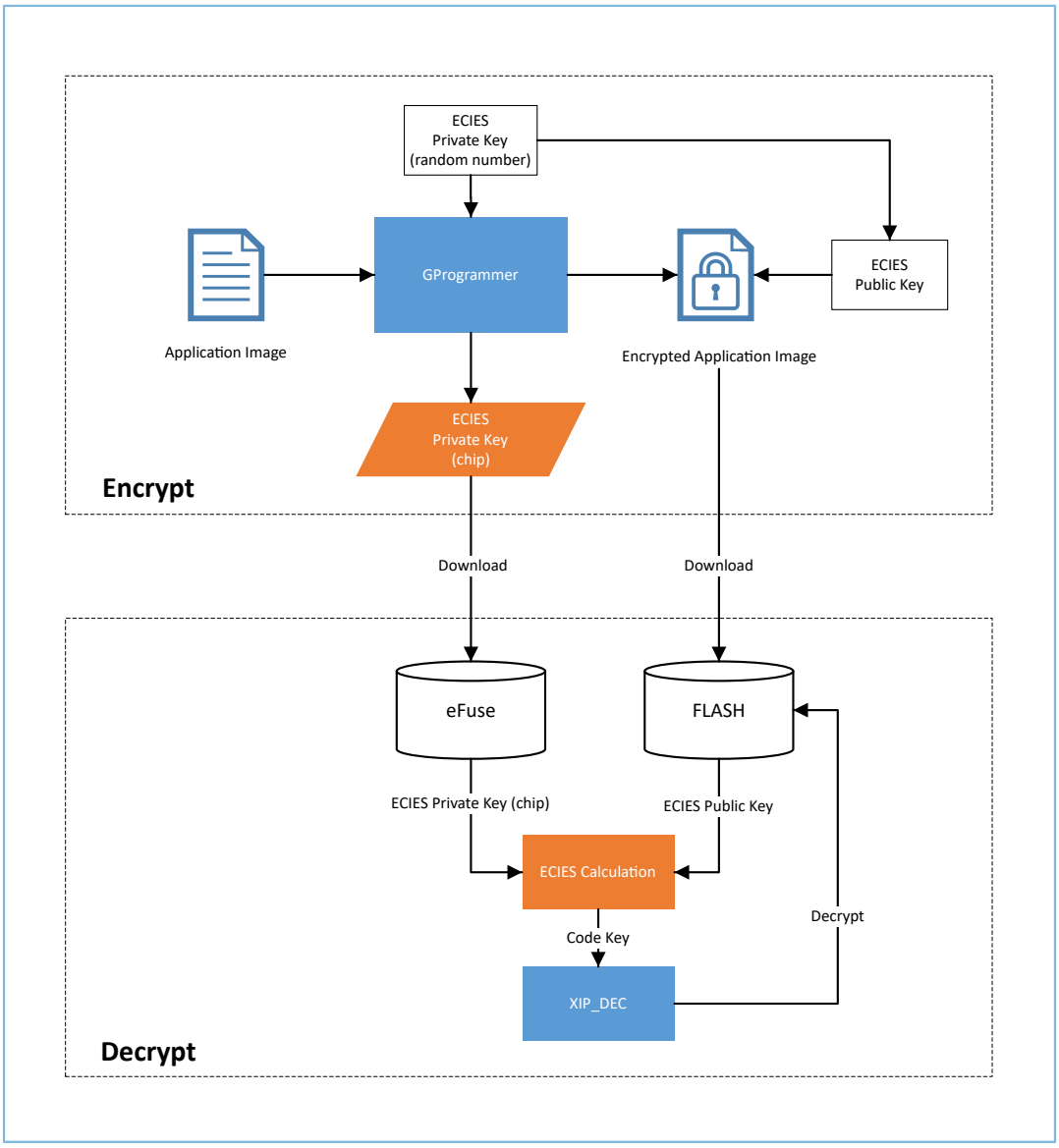

Figure 4-1 Encryption and decryption with hybrid cryptosystem

- Encryption with hybrid cryptosystem:

  1. Generate random private keys in ECIES: Generate random private keys used for ECIES encryption with a random number generator of GProgrammer or other tools.

2. Obtain the session key: Use the random keys generated in Step 1 for ECIES calculation, and obtain the session key (PRESENT-128 symmetric key) and random public keys in ECIES.
3. Generate ciphertext: Encrypt the session message with the session key in GProgrammer, and generate session message in ciphertext with the random public keys in ECIES.
4. Import private keys and ciphertext: Download the firmware key used for ECC decryption to eFuse with GProgrammer; download the session message in ciphertext to Flash.

- Decryption with hybrid cryptosystem:

1. Obtain private key in ECC: The firmware key used for ECC decryption is stored in eFuse. During system boot and initialization, the private key will be loaded to the KEYRAM module. The private key can hardly be obtained for undesired use thanks to the random number generator and eFuse.
2. Obtain the session key: Start ECIES calculation with random public key in ECIES and firmware key through the PKC module; obtain the symmetric firmware code key for PRESENT-128, and load the key to the KEYRAM module.
3. Decrypt the ciphertext: Load the firmware code key to the XIP_DEC module, so that the code on Flash can be decrypted automatically.

4.2 Data Encryption and Decryption

To ensure the security of user data stored in Flash memories (such as the user data and sensor sampling data that require secured storage), the security modules of GR5xx SoCs support lightweight symmetric encryption based on PRESENT-128.

The process for data encryption and decryption is shown in Figure 4-2.

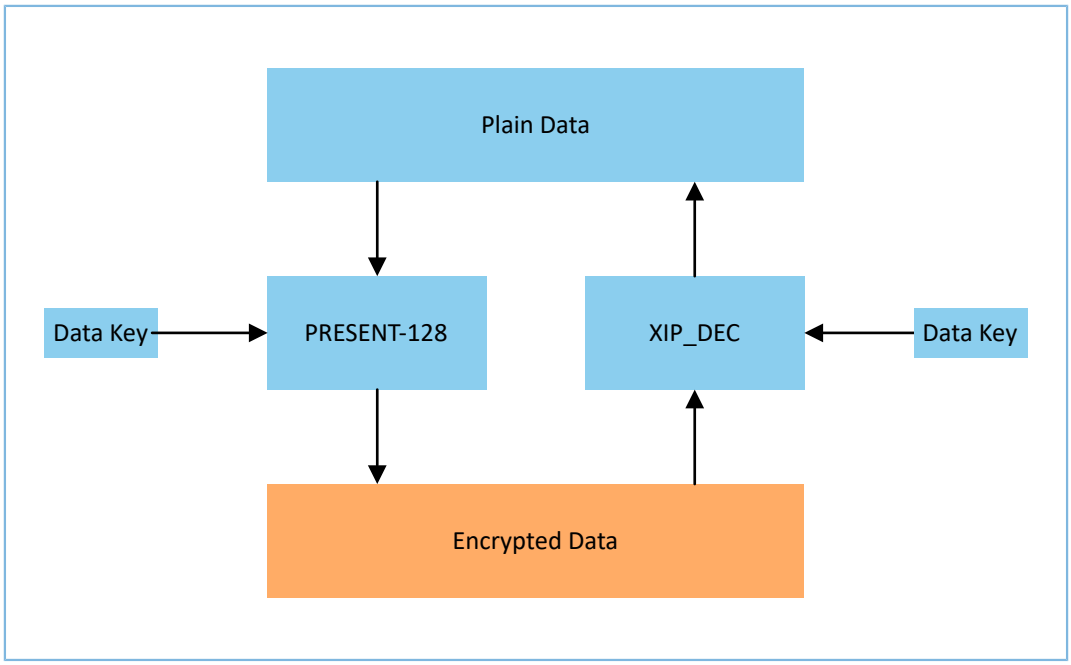

Figure 4-2 Data Encryption and Decryption

To encrypt/decrypt data:

1. The symmetric data keys for PRESENT-128 are stored in eFuse, and are loaded to the KEYRAM module during system startup and initialization. The symmetric private key can hardly be obtained for undesired use thanks to the random number generator and eFuse.
2. In data encryption, load the data key to the PRESENT-128 module, and the key will be encrypted and stored in Flash.
3. In data decryption, load the data key to the XIP_DEC module, so that the data on Flash can be decrypted automatically.

Note:

- Firmware keys and data keys are stored in eFuse separately. When encrypted firmware runs on more than one chip, the chips shall be programmed with the same firmware keys while differences between data keys are allowed, so that one data key is for one device/chip only.
- Flash APIs support encrypted read/write, and can automatically check whether the SoC is in security mode, based on which encrypt/decrypt the written/read data with data keys. Users can also disable encrypted read/write by calling the corresponding API, and write plaintext to/read plaintext from Flash.
- Encrypted read/write through Flash APIs with firmware keys is not supported. Firmware keys are only used to decrypt the code area in the Flash of the XIP_DEC module.
- To store data in Flash under encryption, users can also call the corresponding Non-volatile Data Storage (NVDS) API. For details about NVDS, see the developer guide of the specific GR5xx SoC.

---

5 Digital Signature

A digital signature is a kind of cryptography that ensures message integrity, authenticity, and non-repudiation.

The process of applying digital signatures in the security mode (see Figure 5-1) includes:

- Signing: Users can apply a digital signature to firmware, and download information related to digital signatures to eFuse and Flash memories with GProgrammer.
- Verifying: Bootloader obtains the relevant information from eFuse and Flash memories during booting, to verify the digital signature of firmware.

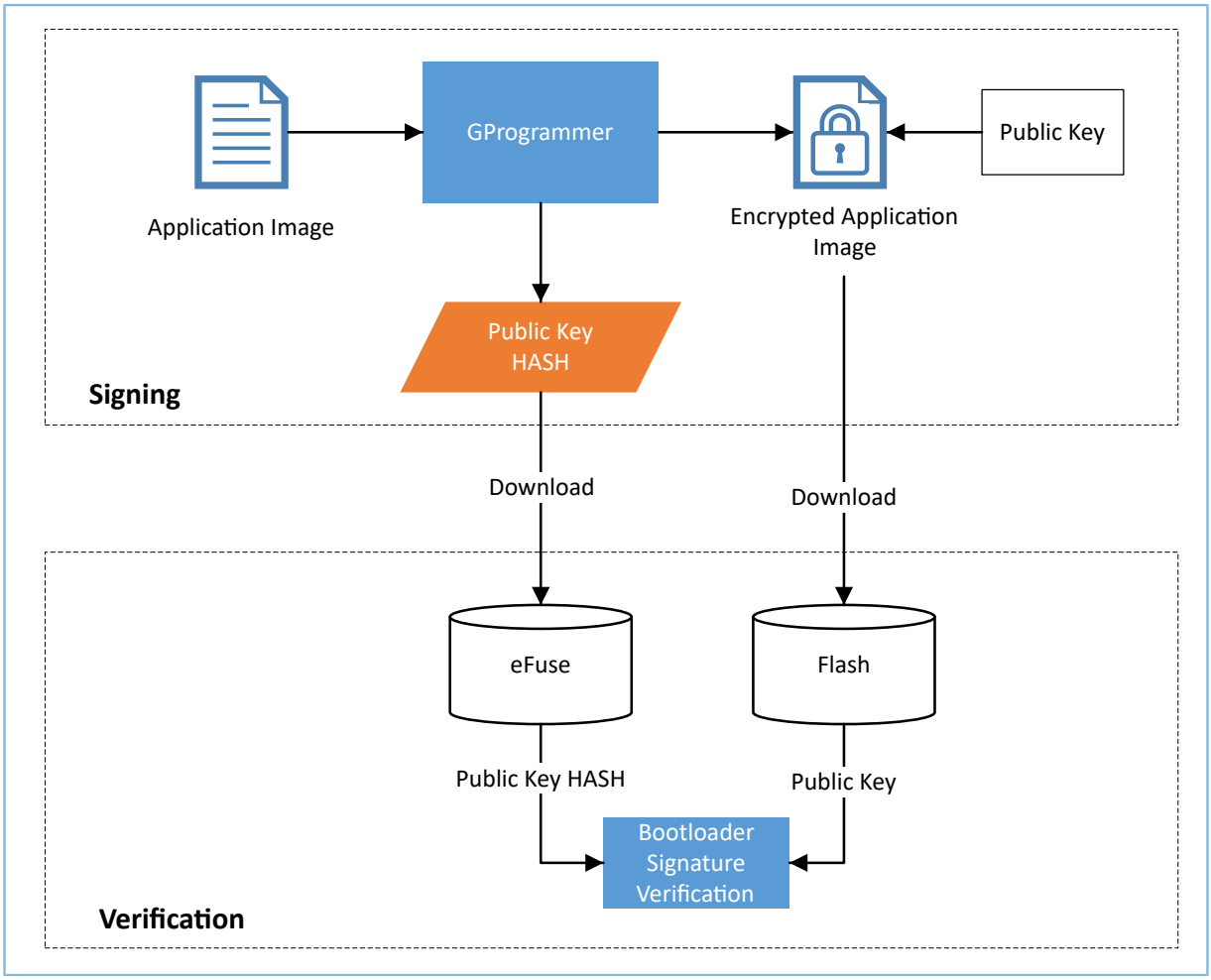

Figure 5-1 Digital signing process

5.1 Signing Process

The process for firmware signing is shown in the figure below.

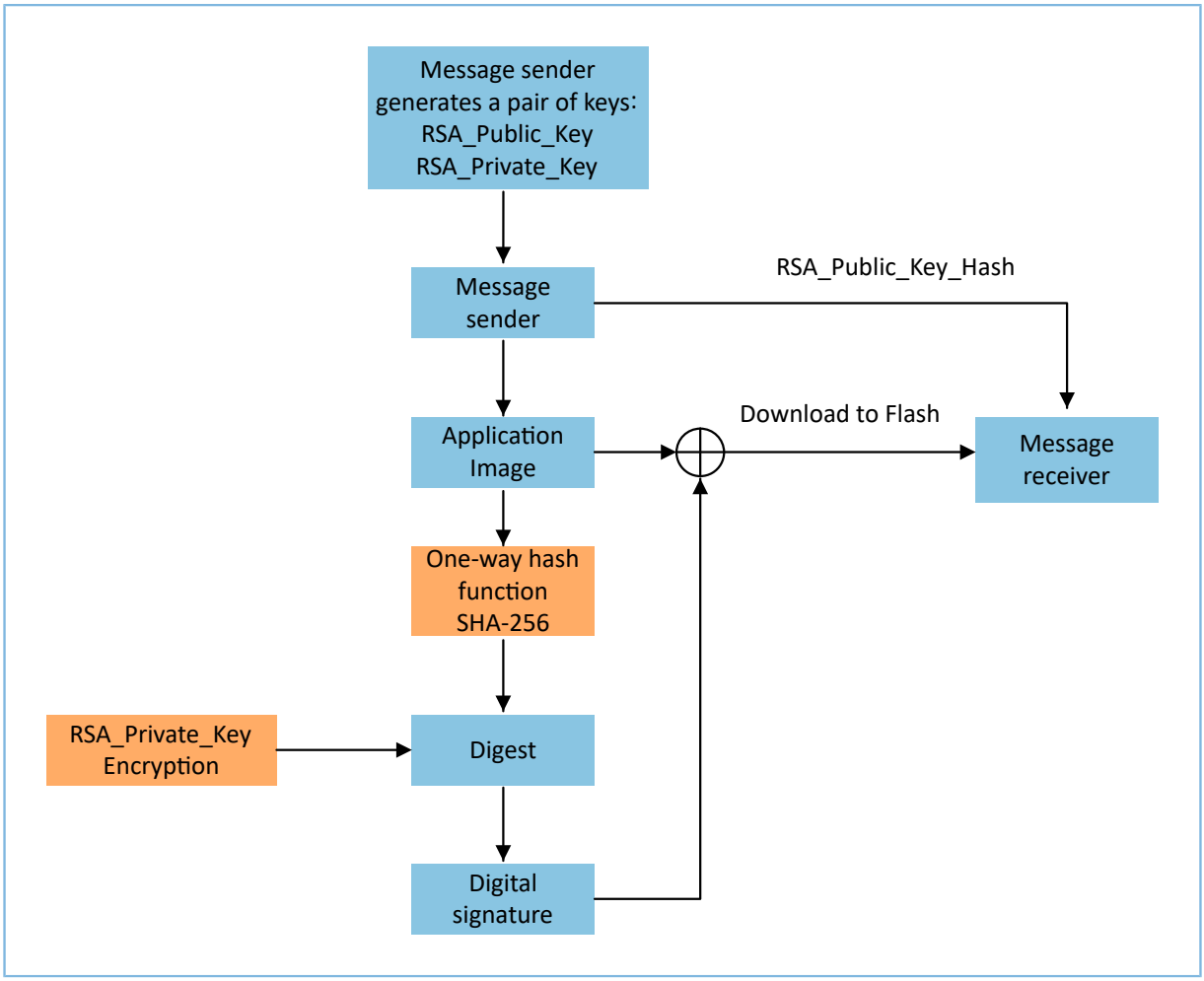

Figure 5-2 Firmware signing process

To sign the firmware:

1. The message sender (the user) creates a key pair (RSA_Public_Key and RSA_Private_Key) with GProgrammer. The key pair helps sign and verify signatures. The message sender creates signatures with the private key (RSA_Private_Key), and the message receiver (GR5xx SoC) verifies the signatures with the public key (RSA_Public_Key).
2. RSA_Public_Key is stored in Application Image and is passed to GR5xx SoCs; the hash value of the public key is stored in eFuse. The hash value generated based on RSA_Public_Key shall be consistent with the RSA_PUBLIC_KEY_HASH value stored in eFuse.
3. To generate a digital signature, generate firmware digests by using one-way hash functions, and encrypt the digest with RSA_Private_Key.

5.2 Verification Process

The process for firmware verification is shown in the figure below.

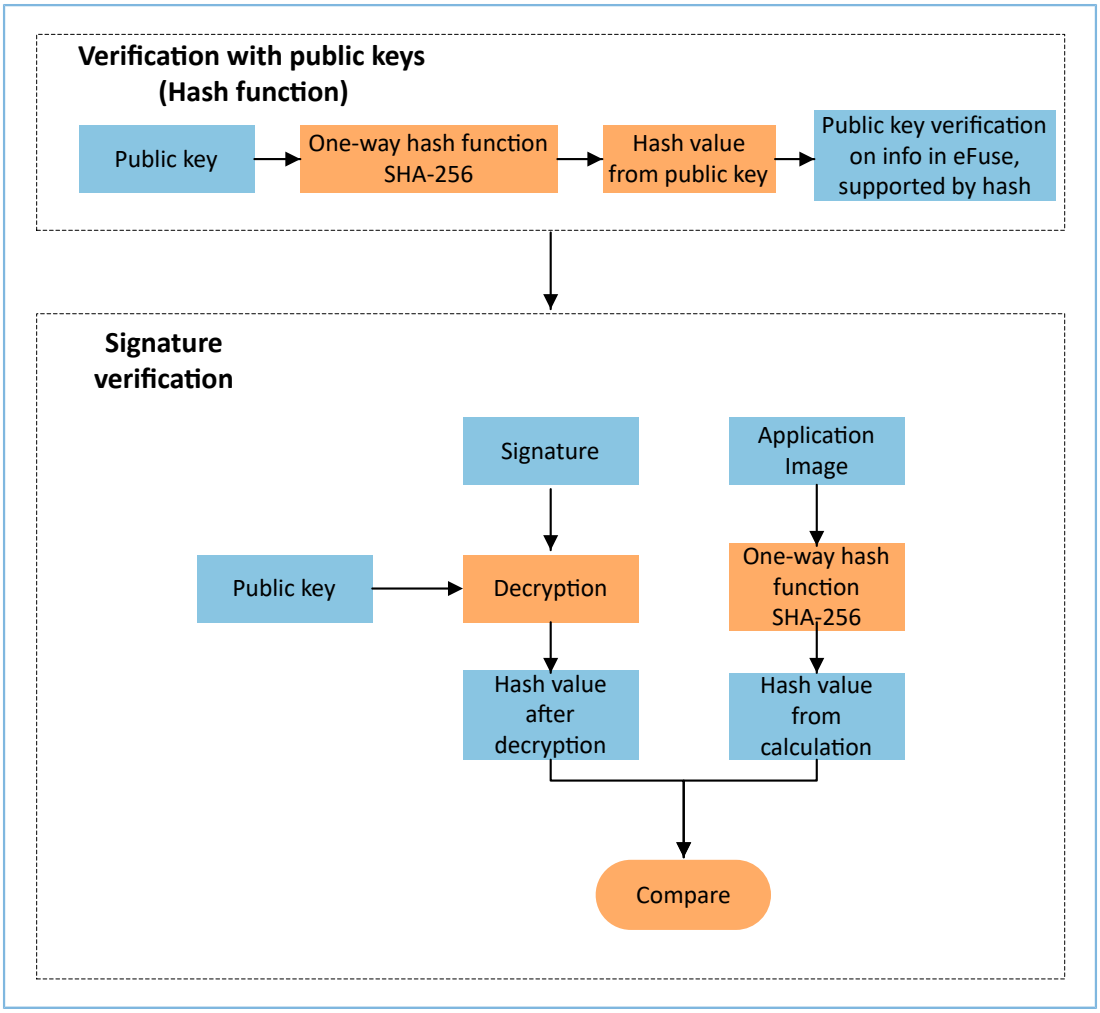

Figure 5-3 Firmware verification process

To verify the signature:

1. Public key verification: Users pass the RSA_Public_Key to Application Image through GProgrammer. After the Application Image is downloaded to Flash, verify RSA_Public_Key with RSA_PUBLIC_KEY_HASH in eFuse. eFuse is OTP. Therefore, in update or upgrade, the hash value generated by RSA_Public_Key in Application Image shall be consistent with RSA_PUBLIC_KEY_HASH stored in eFuse, or the verification fails.
2. Signature verification: GR5xx decrypts a firmware signature with RSA_Public_Key to obtain the decrypted hash value, and calculates the Application Image with a one-way hash function to obtain the hash value. Compare the hash value obtained from decryption and the value obtained from calculation by calling the one-way hash function. If the hash values are consistent, the signature passes verification.

6 Security Mode in Application

In security mode, GR5xx SoCs need eFuse to store information on product configuration and security mode control, and information on keys for encryption and signing. To prevent avoidable loss, it should be noted that eFuse is one-time programmable.

Security mode is often used at the stage of mass production. In production development, non-security mode is recommended.

This chapter introduces the application of security mode with complementary tools (GProgrammer and GRPLT Lite Config Tool).

6.1 Firmware Programming and Encryption with GProgrammer

To enter the encryption page on GProgrammer, click **Encrypt & Sign** on the toolbar on the left, as shown in Figure 6-1.

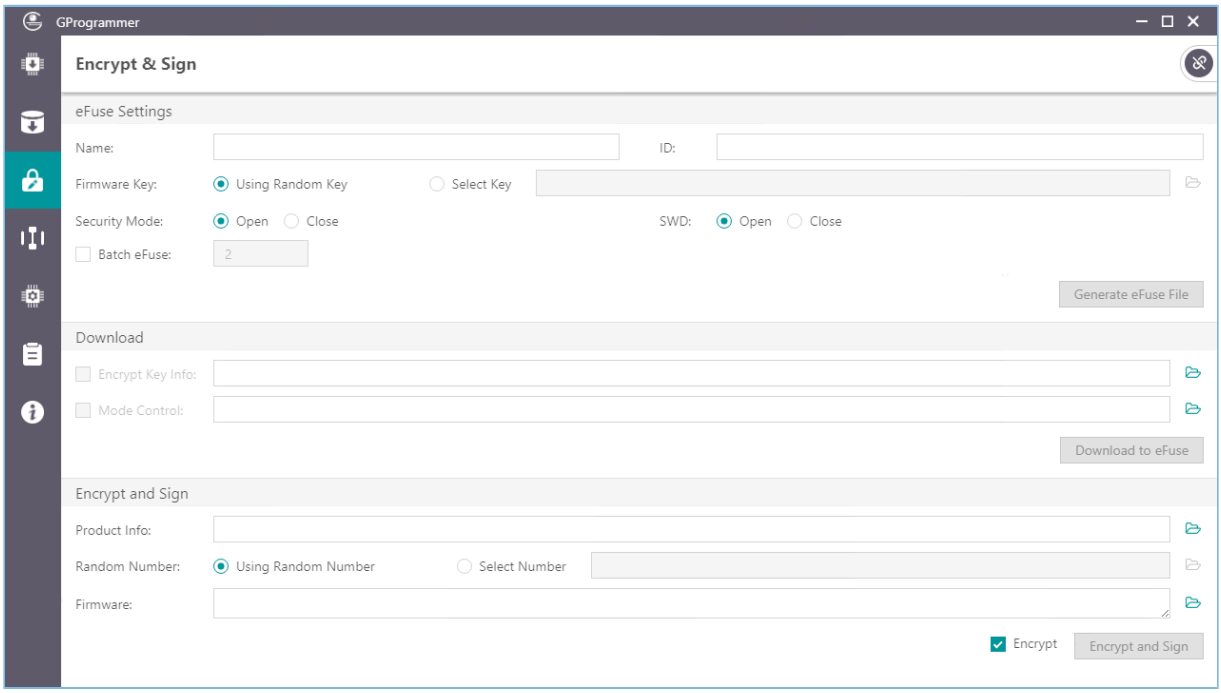

Figure 6-1 Encrypt & Sign page

For more information about eFuse configuration and download operations as well as firmware encryption and signing with GProgrammer, refer to *GProgrammer User Manual*.

6.2 User-defined Encryption with GRPLT Lite Config Tool

To encrypt firmware with tailored user demand, you can run *GRPLT Lite Config Tool.exe*; click **Optional Cfg** > **Encryption Algorithm** and select **Custom Encryption**.

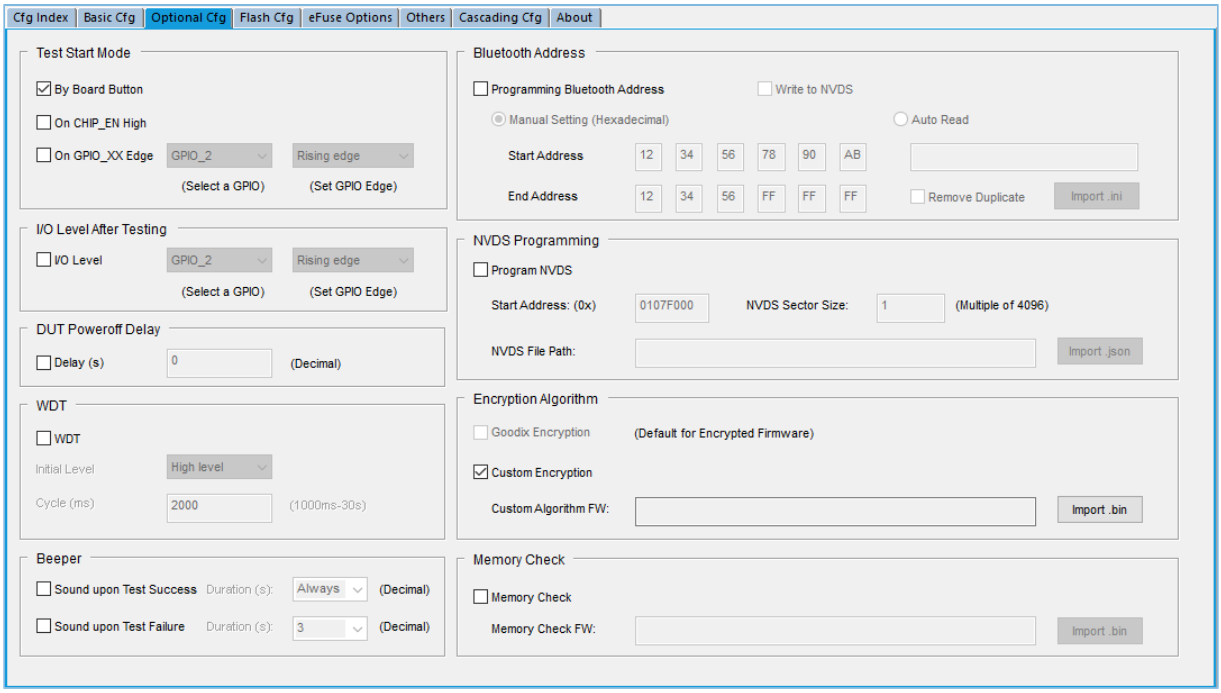

Figure 6-2 Customized firmware encryption pane

For more information about customized firmware encryption with GRPLT Lite Config Tool, see *GRPLT Lite Config Tool User Manual* and *GRPLT Lite Config Tool Customized Firmware Encryption Application Note*.

6.3 SWD Interface

GR5xx SoCs provide an SWD interface. Disabling the SWD interface can prevent unauthorized external access.

In security mode, when users download security mode control files in which SWD interface disabled to eFuse, the SWD interface will be disabled. Users can disable the SWD interface when generating the file to be downloaded to eFuse (*Mode_control.bin*) with GProgrammer. When SWD is disabled, users can also upgrade firmware through DFU. Alternatively, users can also enable the SWD interface with the corresponding register by programming the local application.

Note:

Because eFuse is one-time programmable, once the SWD interface is disabled by programming the security mode control file, the disablement is unrecoverable.

6.3.1 Register

Table 6-1 SWD control register

Bits	Field Name	RW	Reset	Description
SoC series: GR551x Register address: 0xA000C504				
17	SWD_ENABLE	RW	0x0	Enable SWD debugging.

Bits	Field Name	RW	Reset	Description
				Value: • 0x0: Disable • 0x1: Enable
Other SoC series Register address: 0x4000A004				
8	SWD_ENABLE	RW	0x0	Enable SWD debugging. **Value:** • 0x0: Disable • 0x1: Enable

6.3.2 APIs

Table 6-2 sys_swd_enable

Function Prototype	void sys_swd_enable(void)
Function Description	Enable SWD interface.
Input Parameter	None
Return Value	None
Remarks	

Table 6-3 sys_swd_disable

Function Prototype	void sys_swd_disable(void)
Function Description	Disable SWD interface.
Input Parameter	None
Return Value	None
Remarks	

7 FAQ

This chapter describes the possible problems, reasons, and solutions related to the security mode of GR5xx SoCs.

7.1 Why Do I Fail to Execute the Firmware Files?

- Description

  The BIN/HEX files compiled and generated with Keil MDK cannot be executed on encrypted chips.

- Analysis

  The BIN/HEX files compiled and generated with Keil MDK are not encrypted firmware files.

- Solution

  Convert the BIN/HEX files into encrypted firmware with the suffix "_encryptedandsign" or "_signed" through GProgrammer, and program the converted files to Flash with GProgrammer or by DFU.

Note:

The *product.json* file used for firmware encryption with GProgrammer must be consistent with the information stored in the eFuse of the encrypted chip.

7.2 Why Do Firmware Encryption and Signing Fail?

- Description

  During firmware encryption and signing with GProgrammer, the .hex file cannot be imported or **Encrypt and sign "xxx.hex" failed** pops up.

- Analysis

  GProgrammer earlier than V1.2.25 does not support importing .hex files.

  The firmware to be encrypted has been processed by *after_build.bat*.

- Solution

  Download and install GProgrammer V1.2.25 or later.

  Stop and disable *after_build.bat*; use the .hex or .bin file generated by the current project.

7.3 How to Manage Key Information and Encrypted Firmware?

- Description

  Could it be a security risk to provide key files and encrypted firmware for production lines?

- Analysis

  Both key files and encrypted firmware are programmed at the stage of mass production. Therefore, preventing file leakage is of great significance.

- Solution

  Customers shall make strict rules to ensure information security of production lines, and prevent leakage of key files.