GR551x Firmware Encryption Application Note

Version: 2.0

Release Date: 2022-02-20

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: 2F. & 13F., Tower B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 FAX: +86-755-33338099

Website: www.goodix.com

Preface

Purpose

This document introduces encryption and signing technologies, and process for digital signatures of the security module of GR551x System-on-Chips (SoCs), to help developers understand and apply the security mode of GR551x SoCs.

Audience

This document is intended for:

- GR551x user
- GR551x developer
- GR551x tester
- Hobbyist developer
- Technical writer

Release Notes

This document is the eighth release of *GR551x Firmware Encryption Application Note*, corresponding to GR551x SoC series.

Revision History

Version	Date	Description
1.0	2019-12-08	Initial release
1.3	2020-03-16	Updated the release time in the footers.
1.5	2020-05-30	Revised figure format in “Digital Signature”.
1.6	2020-06-30	Updated “Firmware Encryption and Signing with GProgrammer” based on the latest features of GProgrammer.
1.7	2021-02-26	Updated “Firmware Encryption and Signing with GProgrammer” based on the latest features of GProgrammer.
1.8	2021-06-24	Updated SoC model descriptions.
1.9	2021-08-09	Modified SoC model descriptions.
2.0	2022-02-20	Modified the "Firmware Encryption and Signing with GProgrammer" section, and updated the "FAQ" chapter.

Contents

1 Introduction

The security encryption module of GR551x System-on-Chips (SoCs) can encrypt user data and user application firmware in Flash memories, preventing the data and the application firmware from being stolen, and therefore protecting user products.

1.1 Security Fundamentals

This chapter introduces the fundamentals on encryption and signing, to help users better understand the technologies applied by GR551x SoCs for encryption, decryption, and digital signatures.

1.1.1 Encryption

- Symmetric-key encryption: also known as private-key encryption, in which only one key is involved to cipher (by the sender) and decipher (by the receiver) data
- Asymmetric-key encryption: also known as public-key cryptography. It involves a pair of keys: a public key and a private key, with each for encryption and decryption. If the public key is used to encrypt a message, the private key is used to decrypt the message.
- Message integrity: A message shall not be tampered with, which is normally checked with one-way hash tables.
- Message authentication code (MAC): an authentication mechanism which not only examines whether a message is tampered with, but also checks whether a message comes from an expected communication object
- Hybrid cryptosystem: It combines the efficiency of a symmetric-key cryptosystem with the convenience of a public-key cryptosystem. Symmetric-key cryptosystems boost the efficiency of encryption and decryption, and public-key cryptosystems help distribute keys.

1.1.2 Digital Signature

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

2 Encryption and Decryption

This chapter introduces encryption modules and decryption modules, algorithms for firmware encryption and decryption, and processes for data encryption and decryption for GR551x SoCs.

2.1 Encryption and Decryption Modules

The security encryption modules in a GR551x 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 for 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.

  PRESENT-128 is an IP core which enables encrypting or decrypting 128-bit data in one operation, supported by its two 64-bit PRESENT cores. The figure below shows the operating mechanism of PRESENT-128 in encryption and decryption.

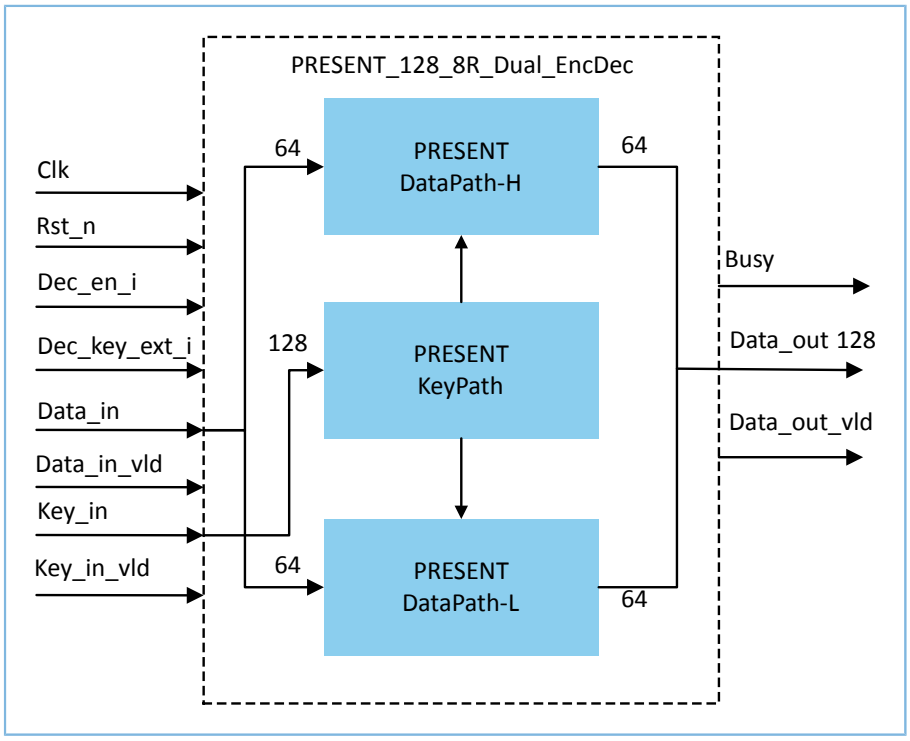

Figure 2-1 Operating mechanism of PRESENT-128 in encryption and decryption

- **eFuse Module**

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

- **KEYRAM Module**

  KEYRAM is mainly applied for key derivation after a chip is powered on. In the secure boot process, true random numbers are generated as masks in each cold boot, to prevent decryption by others through proof by exhaustion. The keyPort bus interface automatically enables encrypted reading of keys and import of the key values to other modules, including AES, HMAC, and XIP_DEC, so that keys are automatically imported to hardware.

- **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, enabled by the HMAC coprocessor. The Keyed-Hash Message Authentication Code (HMAC) applied is in full compliance with the description in FIPS Publication 198. The HMAC module supports more than one type of algorithms (including SHA256 and HMAC-SHA256). The key size is 256 bits.

- **XIP_DEC Module**

  In execute in place (XIP) mode, it is required to call the XIP_DEC module when reading firmware commands or data for real-time decryption. The PRESENT-128 submodule is embedded in the XIP_DEC module, which automatically decrypts encrypted firmware. The figure below is the functional block diagram of the XIP_DEC module.

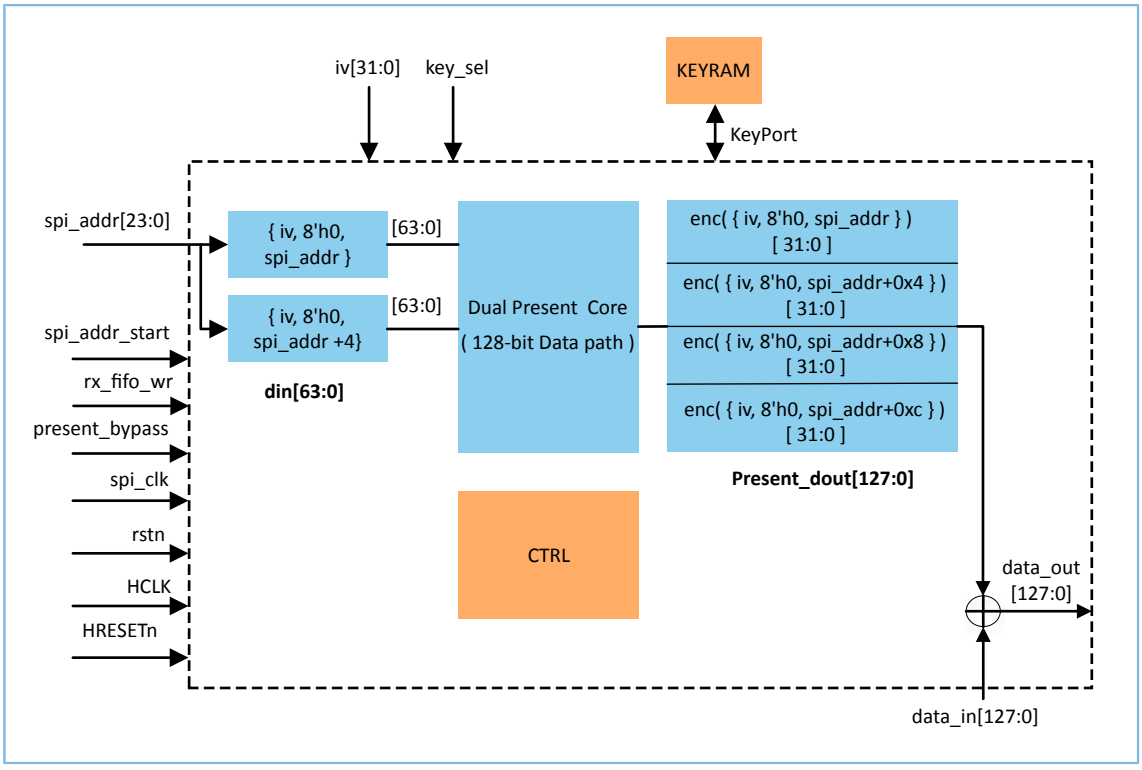

Figure 2-2 XIP_DEC module

Tip:

For more information about the XIP_DEC module, see *GR551x Datasheet*.

2.2 Processes for Firmware Encryption and Decryption

Symmetric encryption (PRESENT-128) and public-key encryption (ECC) algorithms are applied by GR551x SoCs, to ensure firmware security and efficiency.

GR551x encrypts plaintexts with symmetric encryption (PRESENT-128), and encrypts the keys used in symmetric encryption with ECC. The hybrid cryptosystem combines the efficiency of a symmetric-key cryptosystem with the convenience of a public-key cryptosystem in telecommunications.

2.2.1 PRESENT-128 Algorithm

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

The figure below is the functional block diagram of the PRESENT-128 module.

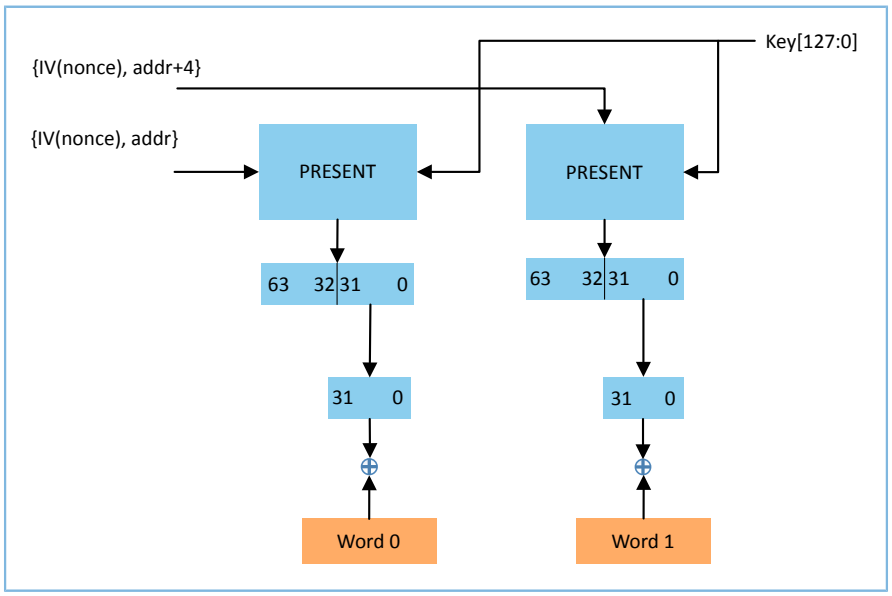

Figure 2-3 Block diagram of PRESENT-128 module

2.2.2 Elliptic Curve Cryptography (ECC) Algorithm

Different from other encryption approaches, which are based on the complicated factorization of large integers, ECC generates keys through the elliptic curve equation.

Compared with RSA, ECC excels in the following aspects:

- 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.

2.2.3 Hybrid Cryptosystem

Public keys are not applicable for long message encryption, because the processing speed of public-key encryption is several hundred times slower than that of symmetric encryption when the keys are of the same length and the same confidentiality level.

The encryption and decryption processes of hybrid cryptosystem are introduced by taking the encryption process as an example, as shown in the figure below.

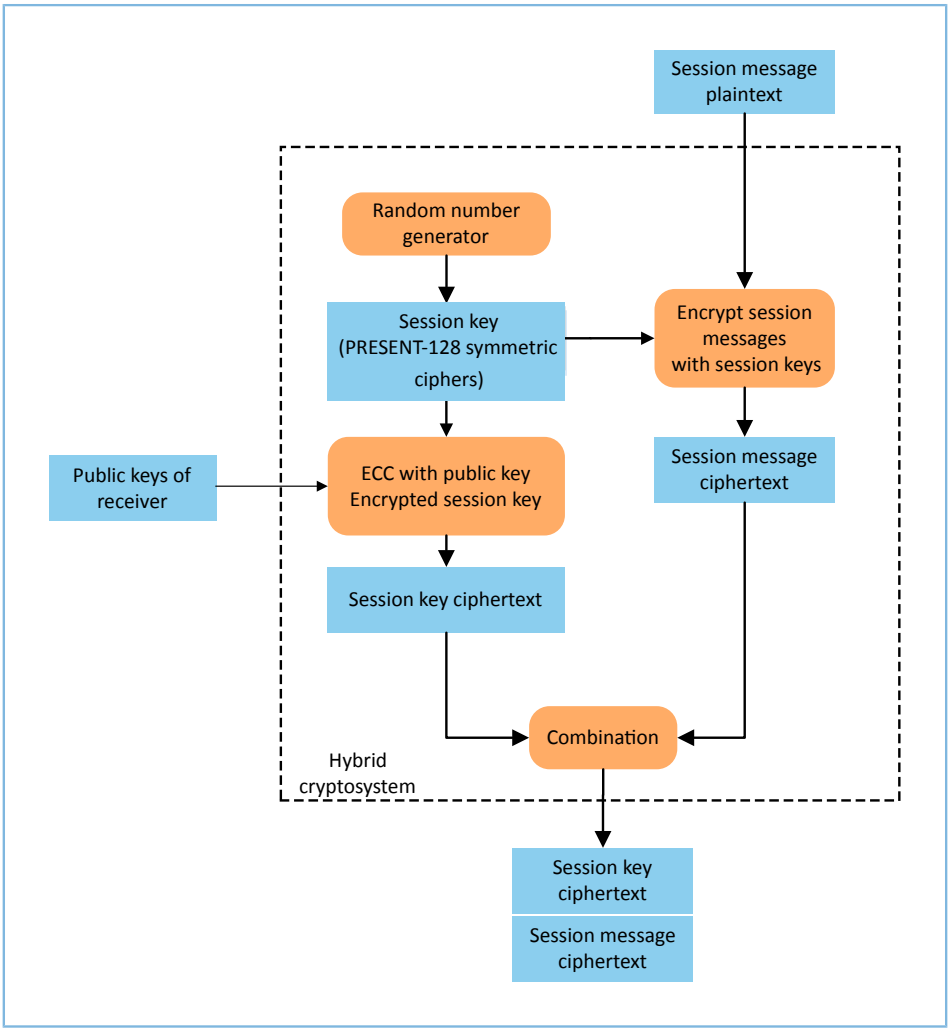

Figure 2-4 Encryption process of a hybrid cryptosystem

Process description:

1. Generate session keys: Generate the session keys (PRESENT-128 symmetric ciphers) for symmetric encryption with random number generators.
2. Encrypt session messages with session keys: Encrypt session messages with session keys, and generate ciphertexts for session messages.

3. Encrypt session messages with public keys: Encrypt session keys with ECC public keys obtained outside of the hybrid cryptosystem, to get the ciphertext of the session keys.
4. Combine ciphertext: Combine the ciphertext of session keys and the ciphertext of session messages.

2.3 Process for 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 module of GR551x SoCs supports lightweight symmetric encryption based on PRESENT-128.

The process for data encryption and decryption is shown in the figure below:

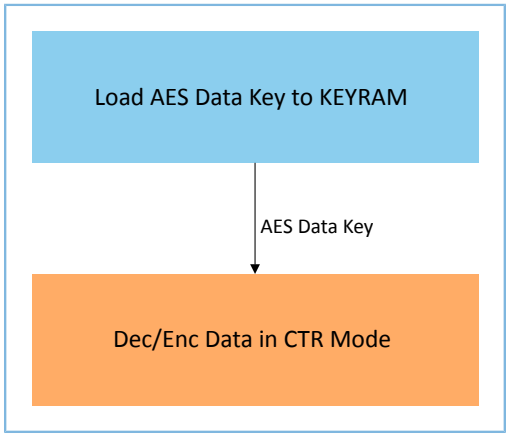

Figure 2-5 Process for data encryption and decryption

Process description:

1. The symmetric keys for AES algorithm are stored in eFuse, and are loaded to the KEYRAM module during initialization when a system boots. By combining the strength of random number generator and eFuse, GR551x SoCs ensure that symmetric keys cannot be obtained by irrelevant parties.
2. In AES algorithm, the encryption process is similar to the decryption process. Both the two processes adopt the block mode with CTR.

Note:

GR551x SoCs adopt asymmetric keys for firmware and symmetric keys for data, which are stored in eFuse separately. When encrypted firmware runs on more than one chip, the chips shall be programmed with the same firmware keys, giving room for differences in data keys, to enable one data key for one device/chip.

Non-volatile Data Storage (NVDS) interface can be directly called when using encrypted data in Flash. For details about NVDS, see *GR551x Developer Guide*.

---

3 Digital Signature

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

The process of applying digital signatures in the security mode of GR551x (see Figure 3-1) includes:

- Sign: Users can download information related to digital signatures to eFuse and Flash memories with GProgrammer, so as to apply digital signatures to firmware.
- Verify: GR551x bootloader obtain the relevant information from eFuse and Flash memories during booting, to verify the digital signatures of firmware.

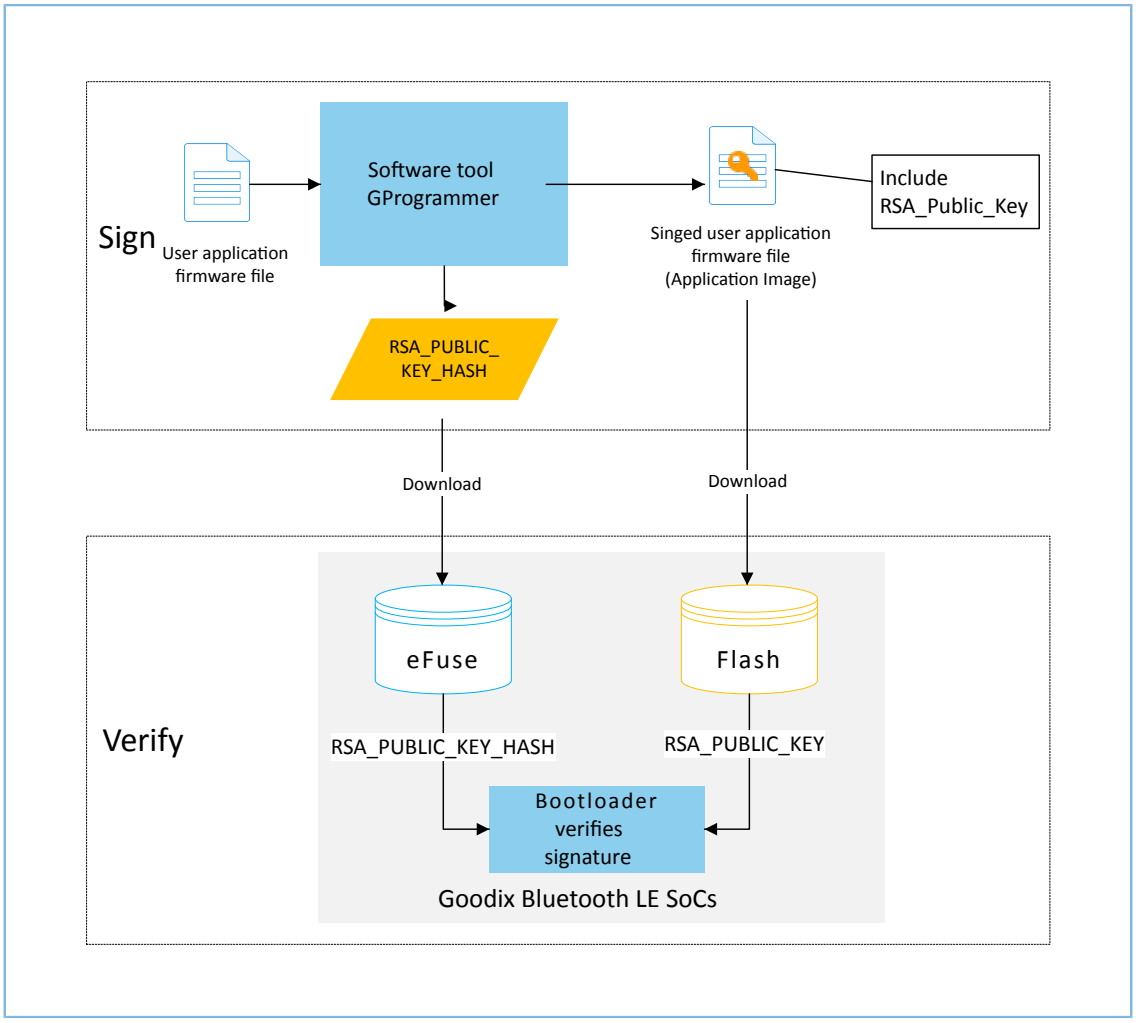

Figure 3-1 Process of applying digital signature to GR551x SoC

3.1 Signing Process

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

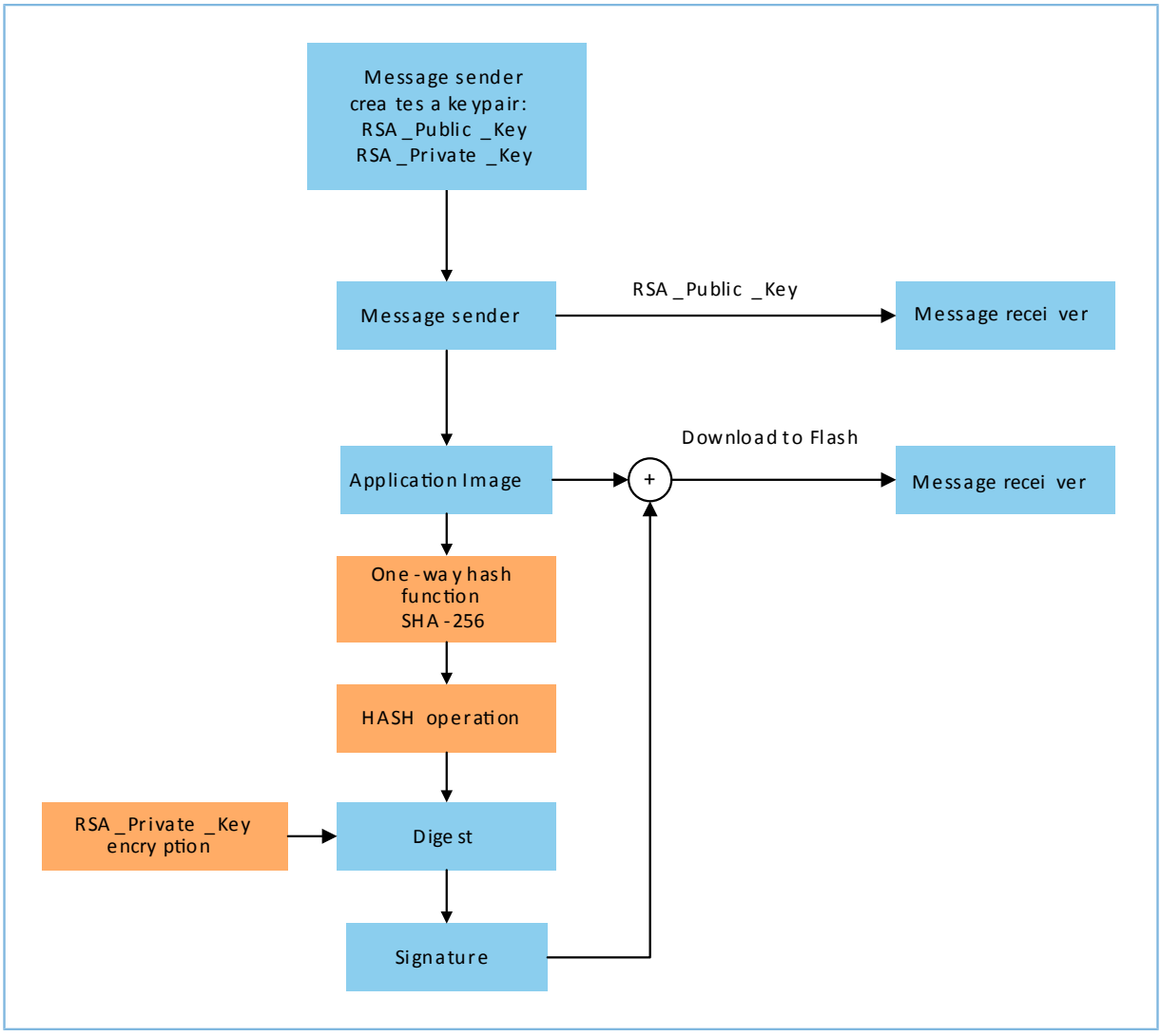

Figure 3-2 Signing process

Process description:

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 (GR551x) verifies the signatures with the public key (RSA_Public_Key).
2. RSA_Public_Key is stored in the Application Image and is passed to GR551x 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. Generate hash values by using one-way hash functions; perform HASH operation on the hash values to generate a digest, and encrypt the digest with RSA_Private_Key. A digital signature is therefore generated.

3.2 Verification Process

The signature verification process is shown in the figure below.

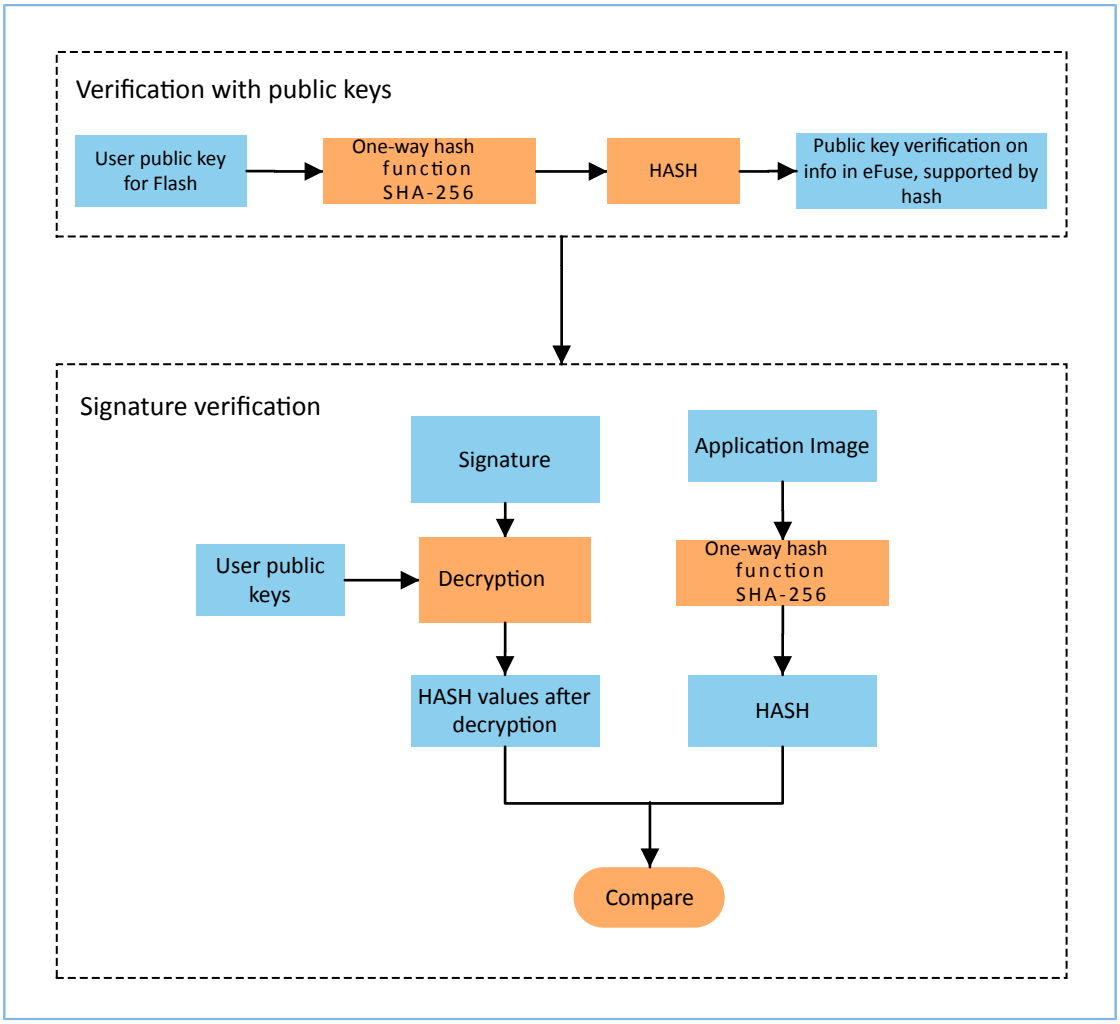

Figure 3-3 Verification process

Process description:

1. Public key verification

   Users pass the RSA_Public_Key to the 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 one-time programmable (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

   GR551x decrypts a firmware signature with RSA_Public_Key to obtain the decrypted hash value, and calculates the Application Image with one-way hash function to obtain the hash value. Compare the hash value obtained from decryption and the value obtained from calculation by calling one-way hash functions. If the hash values are consistent, the signature passes verification.

4 Security Mode Application

GR551x is a series of Bluetooth low energy (Bluetooth LE) SoCs that support multiple mainstream cryptographic algorithms, and that protects user data and user firmware in memories from being eavesdropped.

The security mode of GR551x features:

1. Secured key storage

   GR551x stores keys in eFuse, which means the key information in eFuse cannot be directly accessed through MCUs. When key information is required by encryption modules, an independent hardware unit in the SoC exports the keys from eFuse to the encryption modules automatically.

2. Preventing firmware from being eavesdropped

   GR551x combines the efficiency of a symmetric-key cryptosystem with the convenience of a public-key cryptosystem. For decryption, ECC algorithm shall be used in combination with the private key stored in eFuse, to calculate the necessary keys 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 private key.

3. Preventing malicious attacks

   To prevent malicious attacks, GR551x provides Serial Wire Debug (SWD) locks and secured DFU approaches for application 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 firmware information of GR551x from being read or modified. When SWD is disabled, developers can also upgrade firmware through DFU, during which the encrypted firmware is verified, to prevent the firmware from being maliciously modified.

4. One data key for one device

   GR551x stores the keys for firmware and those for data separately in different zones in eFuse. The same application firmware can be programmed in GR551x SoCs with different keys for data, so that one device owns its unique keys for data.

4.1 Firmware Encryption and Signing with GProgrammer

In general, apply security mode at the stage of mass production. At the stage of development, apply non-security mode.

In security mode, GR551x SoCs need eFuse to store information on product configuration and security mode control, and information on keys for encryption and signing.

When using GProgrammer, users can generate eFuse files by specifying product names, IDs, and firmware keys, and by configuring security mode and SWD interfaces.

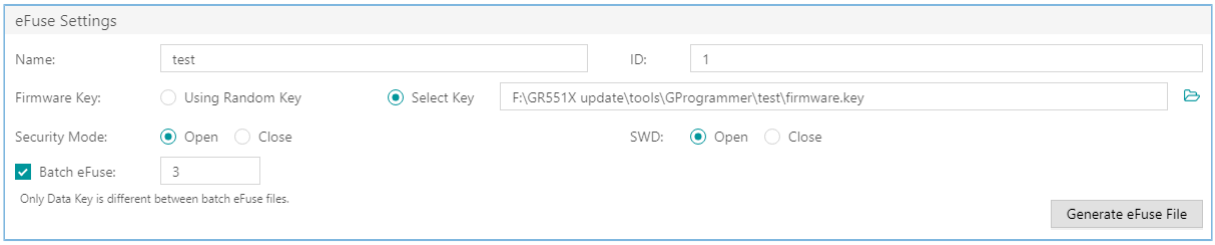

Figure 4-1 Generating eFuse files

Note:

- eFuse is a one-time programmable (OTP) memory with random access interfaces in GR551x SoCs.
- **Firmware Key** can be random keys automatically generated by GProgrammer. Users can also add key files themselves.
- **Security Mode**: Selecting **Open** enables security mode for the SoC. Note that you cannot disable the security mode once the mode is enabled.
- **SWD**: Selecting **Close** disables SWD. Alternatively, developers can upgrade firmware files through DFU.
- **Batch eFuse**: Select this item to generate certain batches of files on data keys on demand, which helps building the application scenarios requiring one data key per device. Clearing this item means that only one data key is generated.

The files are generated as below:

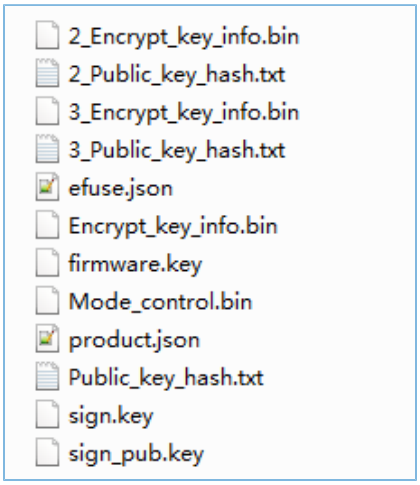

Figure 4-2 Generated files

- *efuse.json*: a temporary file
- *Encrypt_key_info.bin*: a file covering information on products, encryption, and signing. This file shall be downloaded to and stored in eFuse. *2_Encrypt_key_info.bin* and *3_Encrypt_key_info.bin* are firmware files generated by GProgrammer when **Batch eFuse** is selected and set to **3**.
- *firmware.key*: a private key for encrypting firmware

- *Mode_control.bin*: a file covering information on security mode and SWD. This file shall be downloaded to and stored in eFuse.
- *product.json*: a product information file. This file shall be imported to GProgrammer when encrypting or signing firmware.
- *sign.key*: a private key to generate signatures
- *sign_pub.key*: a public key to verify signatures
- *Public_key_hash.txt*: a public key hash file to verify signatures

---

Tip:

Please keep the files properly, and avoid file leakage or loss. These files are necessary to “program eFuse” and to “encrypt and sign firmware”.

To help users download files to eFuse or encrypt and sign firmware, the paths for the *Encrypt_key_info.bin* file and the *Mode_control.bin* file are added to the **Download** area by default; the path for the *product.json* file is added to the **Product Info** pane in the **Encrypt and Sign** area by default. Click **Download to eFuse** to program information on security mode control and keys to eFuse. Avoid programming with two different keys to the eFuse.

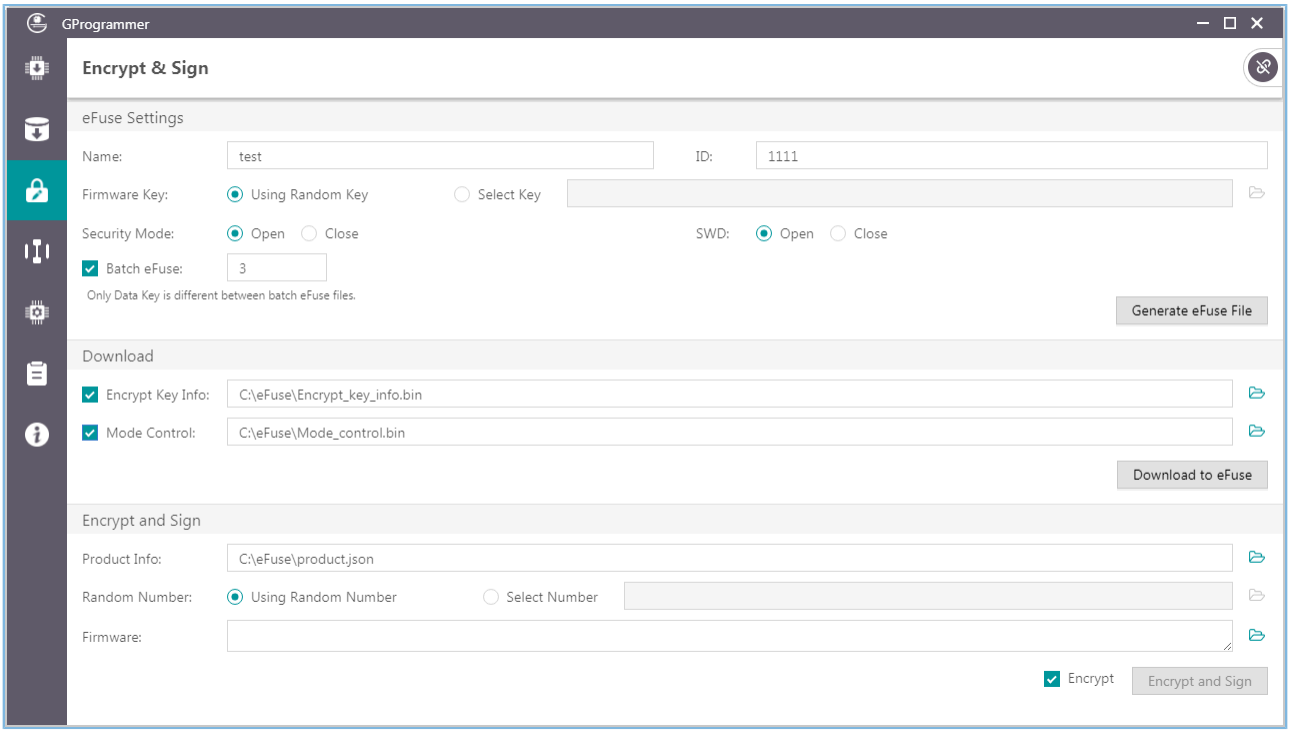

Figure 4-3 Paths for automatically loaded files

In the **Firmware** bar, choose the unencrypted firmware that needs to be encrypted and signed, or signed only.

- To encrypt and sign the firmware, check the **Encrypt** box, and the button changes from **Sign** to **Encrypt and Sign**;
- To sign the firmware only, clear the **Encrypt** box, and the button changes back to **Sign**.

Choose the directory to save the (encrypted and) signed firmware, click the **Encrypt and Sign**/**Sign** button.

GProgrammer allows importing unencrypted firmware files in both HEX and BIN formats while only exports firmware files in BIN format. For more information on operations in GProgrammer, see *GProgrammer User Manual*.

Note:

No modification of eFuse-generated files is allowed because any modification may lead to firmware encryption and signing failures.

4.2 Hardware SWD Interface

A GR551x SoC provides an SWD interface. Disabling the SWD interfaces can prevent unauthorized external access to chips. In security mode, when users download security mode control files configured with forbidden SWD interfaces to eFuse, the SWD interfaces can never be enabled. Users can disable the SWD interfaces when generating the file to be downloaded to eFuse (*Mode_control.bin*) with GProgrammer.

5 FAQ

This chapter describes the possible problems, reasons, and solutions when enabling security mode in a GR551x SoC.

5.1 Why Do I Fail to Execute BIN/HEX Firmware Files?

- Description

  The BIN/HEX files compiled and generated with Keil MDK5 cannot be executed on encrypted chips.

- Analysis

  The BIN/HEX files compiled and generated with Keil MDK5 are not encrypted firmware files.

- Solution

  1. Convert the BIN/HEX files into encrypted firmware with suffix name "_encryptedandsign" or “_encrypted” through GProgrammer.
  2. Programme the files into Flash memories through GProgrammer or 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.

5.2 Why Does GProgrammer Fail to Encrypt or Sign Firmware Files?

- Description

  GProgrammer fails to import HEX firmware files or prompts as **Encrypt and sign “xxx.hex” failed** when I encrypt or sign firmware files through GProgrammer.

- Analysis

  - Importing HEX firmware files is only available on GProgrammer V1.2.25 and later versions.
  - The to-be-encrypted firmware files have been processed by using *after_build.bat*.

- Solution

  - Install GProgrammer V1.2.25 or later versions.
  - Stop using and disable *after_build.bat*, and then select a BIN or HEX file that is regenerated in the current project for encryption and/or signing.

5.3 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 and encrypted firmware.