Application Note

Discussion on Pairing Issues in the Application of TI Bluetooth Chip CC2642R-Q1

Simon Huo *Wireless Connectivity Solution*

ABSTRACT

This document was translated from a simplified Chinese source. (ZHCAFM5)

In the development of Bluetooth Low Energy (BLE) systems based on CC2642R-Q1, pairing and bonding issues represent a typical technical bottleneck that troubles developers due to the coupling of multiple factors. These problems crop up intensively during the middle and late stages of connection establishment: upstream, they correlate with the configuration of basic Bluetooth parameters and connection event management mechanisms; downstream, they directly impact the execution of user business logic in encrypted channels. Their causes often involve multidimensional factors such as connection event scheduling conflicts, anomalies in service discovery mechanisms, and insufficient memory resource allocation, with the core processing logic highly dependent on the encapsulation implementation of the vendor protocol stack.

Currently, the automotive electronics field faces dual challenges: on one hand, it must satisfy stringent security compliance requirements such as ISO 21434 and WP.29 R155; on the other hand, it is limited by the computational resources of embedded platforms, requiring the achievement of multi-standard compatibility through a lightweight software architecture. This poses a severe challenge to root cause localization and solution design—developers frequently must conduct problem tracing under the triple constraints of a black-box protocol stack, real-time requirements, and security regulations.

Centering on the pairing and bonding implementation mechanism of the CC2642R-Q1 chip, and combined with the technical requirements of the Bluetooth Core Specification, this paper systematically reviews engineering debugging experiences in typical abnormal scenarios, focusing on:

1. Security characteristics and implementation essentials of differentiated pairing modes
2. Diagnostic paths for high-frequency issues such as MIC failure, DHKey check fail, and PIN or Key missing
3. A lightweight solution proposed to address encryption resource conflicts

Table of Contents

List of Figures

List of Tables

1 Analysis of Bluetooth Pairing Technology Principles

Device pairing in Bluetooth Low Energy technology is the core mechanism for building secure communication channels, the essence of which lies in establishing a defense system against unauthorized eavesdropping of wireless signals through encrypted links. In an open wireless communication environment, data packets transmitted by Bluetooth devices carry an inherent risk of interception by nearby unauthorized devices. Although signal interception at the physical layer is difficult to completely eliminate, the application of cryptographic algorithms can effectively guarantee the confidentiality of data content. The implementation of this security mechanism relies on the negotiation and exchange of key materials by both communication parties during the initial interaction stage, which is precisely the core value of the pairing process—it not only requires coordination to generate session keys for data encryption, but also mutual determination of security parameters such as encryption algorithms and key lengths.

1.1 LE Legacy Pairing and LE Secure Connections Pairing

Two main technical implementation paths currently exist in the Bluetooth protocol stack: LE Legacy pairing, introduced as a foundational security scheme in the Bluetooth Core Specification v4.0, protects data through a Short Term Key (STK) generation mechanism; while LE Secure Connections, introduced as a security scheme in the Bluetooth Core Specification v4.2, is based on a more advanced Elliptic Curve Diffie-Hellman (ECDH) algorithm to provide stronger resistance against man-in-the-middle attacks during key negotiation, marking a significant evolution of the Bluetooth security architecture from traditional symmetric encryption to an asymmetric cryptographic framework. For more details, please refer to the Bluetooth Core Specification and the TI BLE5-Stack User's Guide (GAP Bond Manager and LE Secure Connections sections).

1.2 Analysis of Association Mode Security Characteristics

As mentioned above, the LE Legacy mode supports three association modes, while LE Secure Connections supports four association modes. The critical difference between the two protocol versions lies in the security foundation of the key generation mechanism: LE Legacy relies on short-term key exchanges that are vulnerable to cracking, whereas LE Secure Connections achieves permanent forward secrecy through the cryptographically stronger ECDH algorithm. Developers can enforce the LE Secure Connections mode to circumvent known vulnerabilities of LE Legacy at the protocol layer. Two devices that have established a Bluetooth connection will negotiate the association mode to be adopted by both parties at the early stage of connection establishment. For specific rules to follow, please see GAP Bond Manager and LE Secure Connections — SimpleLink™ CC13XX/CC26XX SDK BLE5-Stack User's Guide 2.2.11.00 documentation. The association modes supported by BLE devices are divided into four types, and their characteristics and differences are shown in the table below:

Table 1-1. Comparison of Four Association Modes

Association Mode	Just Works	Passkey Entry	Numeric Comparison	Out of Band
Core Mechanism	No user authentication; pairing completes automatically.	The user enters a 6-digit numeric password displayed on one device into the other device.	Both devices display a 6-digit number, and the user confirms whether they are identical.	Pairing information is securely exchanged in advance via non-BLE methods (such as NFC or QR codes).

Table 1-1. Comparison of Four Association Modes (continued)

Association Mode	Just Works	Passkey Entry	Numeric Comparison	Out of Band
User Interaction	None	The user must input the password.	The user only needs to **confirm** whether the displayed values match.	Usually only requires bringing the devices close together or scanning once.
Security Level	Lowest	Medium to High	High	Highest
Mitigation of MITM Attacks	None	Requires the attacker to know and input the correct password in real time to successfully execute an MITM attack.	Requires the attacker to simultaneously tamper with the values displayed to the user on both devices without being detected.	Relies on the security of the OOB channel (usually physical security).
Applicable Connection Method	Legacy Pairing / Secure Connections	Legacy Pairing / Secure Connections	Secure Connections	Legacy Pairing / Secure Connections
Key Generation Foundation	STK/LTK	STK/LTK	LTK	LTK
Applicable Scenarios	Devices with low security requirements, or devices lacking input/ output capabilities.	One device has display capability (displays password) and the other has input capability (keyboard to input password).	Devices where both sides possess display capabilities.	Scenarios requiring the highest security or pursuing ultimate convenience; devices supporting identical OOB technologies on both sides.

In the security ranking of association modes, Just Works offers the lowest security, while the OOB method provides the highest security. The system will automatically select the most secure association mode for pairing based on the capabilities supported by both devices. For the complete secure association mode selection flow and detailed specifications, please refer to the Bluetooth Core Specification version v5.2 [Vol 3] Part H, Section 2.3.5.1, Table 2.7 and Table 2.8.

1.3 Privacy Protection and Address Management

Bluetooth devices are identified by their Bluetooth Device Address (BDA). However, to prevent tracking, BLE devices can enable a privacy feature that substitutes the fixed identity address by periodically generating new addresses, thereby effectively hiding the identity address of the device. Specifically, the address types of Bluetooth devices mainly include the following:

- Public Address: Uses the permanent identity address of the device, which remains fixed and unchanged.
- Random Static Address: The device generates a random address at each power-up, and this address does not change during the operation of the device.
- Resolvable Private Address (RPA): The device periodically generates a random address, which needs to be resolved into its true identity address through the device's Identity Resolving Key (IRK), enhancing privacy.
- Non-resolvable Private Address: The device periodically generates a completely random address that cannot be resolved to the device's true identity address, used in certain scenarios that do not require identity authentication.

For the detailed process of private address generation and resolution, please refer to the Bluetooth Core Specification and the TI BLE5-Stack User's Guide (GAP Bond Manager and LE Secure Connections sections) to obtain deeper information.

2 Common Issues in Pairing Encryption

Pairing problems within the aforementioned pairing scenarios include both obvious issues that are easy to troubleshoot and more complex ones whose root causes are not apparent. Understanding Bluetooth pairing issues and their causes can help developers localize faults more efficiently, and improve the connection stability and interaction reliability of devices. Below, we will review some typical difficult Bluetooth pairing problems frequently encountered in daily applications and debugging.

2.1 Integrity Check Fault—MIC Verification Failure (MIC failure)

The MIC (Message Integrity Check) is an important tool used to verify the integrity of Bluetooth communication data. It is typically 4 bytes long and is used as a message integrity check under link encryption conditions. Possible causes for this type of error include:

- Key mismatch: The two communicating parties may have used different encryption keys (such as Long Term Keys or Short Term Keys - LTK/STK).
- Packet tampering: During transmission, communication data packets may become incomplete or inconsistent due to noise, interference, or malicious tampering.
- Protocol stack implementation issues: Bugs exist in the underlying implementation, leading to inconsistent data generated during encryption and decryption processing.
- Timeout or communication conflict: The two communicating parties are in different session states, resulting in MIC verification failure.
- Encryption resource conflict issue: For CC2642, both RPA resolution and the pairing/encryption flows need to occupy the crypto engine. If this crypto engine is occupied by other requests when a pairing request is processed, it may lead to an encryption resource conflict, resulting in a MIC verification resource error.

As can be seen, although a MIC error appears superficially as a 4-byte verification problem, the point of occurrence can reside in various processes ranging from over-the-air packet interactions on the link to chip resource management.

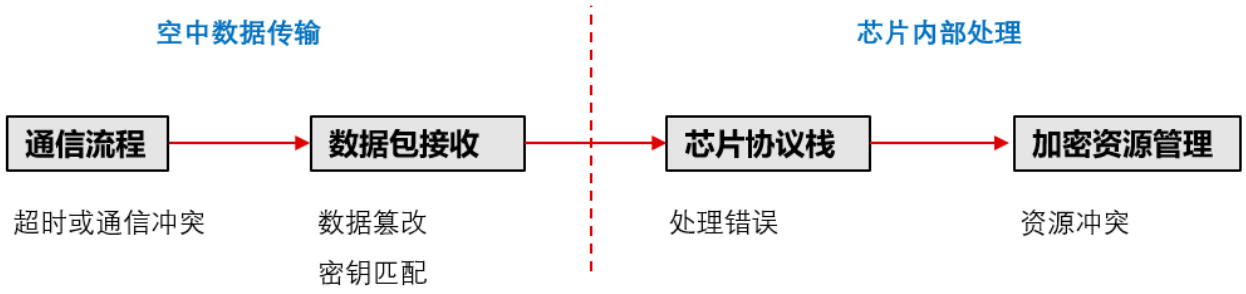

Figure 2-1. Possible problem scenarios for MIC failure

For this type of problem, it is recommended to adopt a layered, progressive diagnostic strategy, drilling down step by step from the physical layer to the protocol layer:

- Test the RF environment: Conduct replication experiments in an environment with low interference (a shielded room or a quiet test environment) to eliminate signal interference. Simultaneously perform an RF inspection of the environment where the problem occurs, such as conflicts with surrounding WiFi or other 2.4GHz devices, whether there are special Bluetooth data packets, and so forth.
- Check communication parameters: Capture protocol events of both devices during the communication process (using packet capture tools such as Ellisys, nRF Sniffer) to verify whether the communication parameters match. The use of Bluetooth packet capture tools can be combined with checking the enabled state of the chip's RF TX/RX to help confirm whether both communicating devices are in an RF non-responsive state when the problem occurs, and whether corrupted or erroneous data packets were received at the receiving end.
- Check key generation: Trace the key negotiation process during the pairing flow, verify the consistency of the LTK/STK generation algorithm, debug and check the output value of the random number generator during the key exchange phase, and cross-verify the synchronization state of the encryption initialization vector (IV) counter.
- Synchronize encryption states of both sides: Parse the Pairing Feature Exchange field in the over-the-air packets

Verify the encryption mode negotiation flow against the Bluetooth protocol specification.

Case 1:

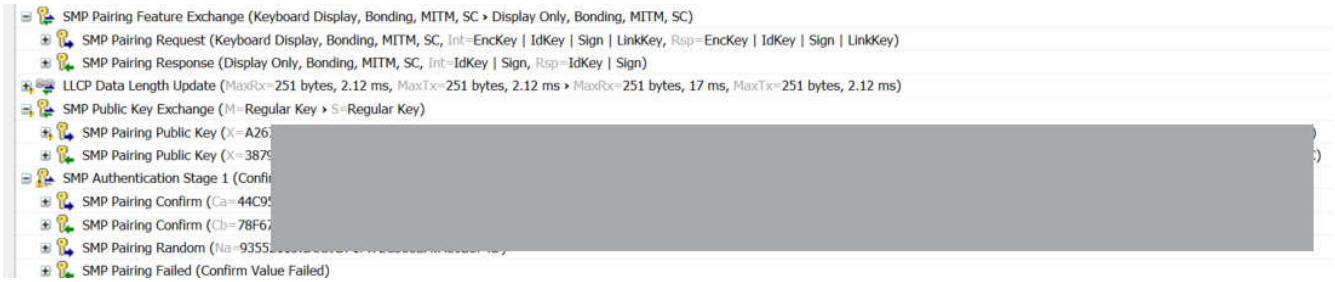

Figure 2-2. Sniffer log of over-the-air packets in Case 1

In the actual over-the-air packets of this case, it can be clearly seen that during the pairing feature exchange phase, the central node and the peripheral node negotiate to use the passkey entry method for pairing. However, in SMP Authentication stage 1, the peripheral node responds with a refusal to continue executing the encryption instruction to the first pairing random packet from the central node. This rejection and its error code reflect that the peripheral considers the pairing random data from the central node to be incorrect. In practice, we can take checking and calculating whether the Na data contained in the pairing random packet is correct as an entry point. If this value complies with the calculation requirements of Ca and Cb mentioned earlier, then the peripheral node made an incorrect judgment here, and subsequent debugging will focus on investigating why the peripheral node calculated incorrectly; if it does not comply with the calculation requirements, then a problem occurred in the calculation of Ca and Cb on the central side.

- Check encryption resource utilization: For CC2642, it contains an AES security module that can accelerate the encryption and decryption processes using hardware resources. See the TI BLE5-Stack User's Guide for details. In common Bluetooth scenarios, the application of the AES security module can occur during the resolution of the peer device address (RPA type address) before connection, during the pairing and encryption processes during connection, and during potential hash calculations at the application layer.

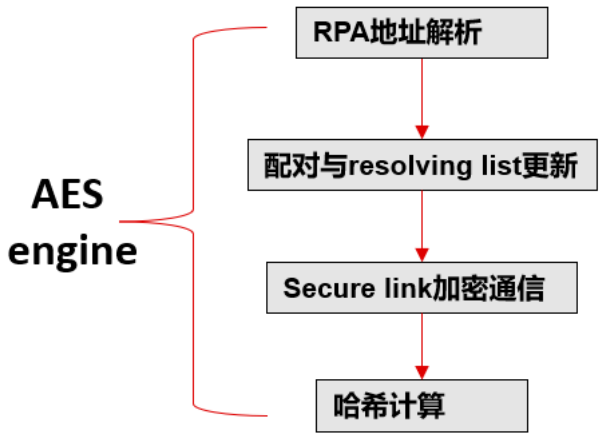

Figure 2-3. Processes covered by AES hardware resources

Therefore, the troubleshooting of such causes should take into account all scenarios involved in the current Bluetooth application. If the environment is complex and cannot be easily analyzed, logs or IO toggles should be added at the invocation of the AES engine in the code to experimentally check whether the current application has unexpected usage of encryption resources. Concurrently, judge the cause of the problem based on the time and location where the log appears. An example method for adding IO toggling is as follows:

Modify the AESCCM_startOperation function in AESCCMCC26xx.c (where IO8 and 9 are merely IOIDs for the example application)

```
#include <inc/hw_gpio.h>
#include <driverlib/ioc.h>

static int_fast16_t AESCCM_startOperation(AESCCM_Handle handle,
                                          AESCCM_Operation *operation,
                                          AESCCM_OperationType operationType)
{
IOCPinTypeGpioOutput( IOID_8);
IOCPinTypeGpioOutput( IOID_9);

GPIO_writeDio(IOID_8, 1);

  DebugP_assert(handle);
 …
 …
  /* Load the key from RAM or flash into the key store at a hardcoded and reserved
location */
  if (AESWriteToKeyStore(keyingMaterial, keyLength, AES_KEY_AREA_6) != AES_SUCCESS)
{
  GPIO_writeDio(IOID_9, 1);

  /* Release the CRYPTO mutex */
  SemaphoreP_post(&CryptoResourceCC26XX_accessSemaphore);
 GPIO_writeDio(IOID_9, 0);

  return AESCCM_STATUS_ERROR;
  }
  …
…
  AESStartDMAOperation(operation->input, operation->inputLength, operation->output,
operation->inputLength);
GPIO_writeDio(IOID_8, 0);
return AESCCM_waitForResult(handle);

}
```

Case 2:

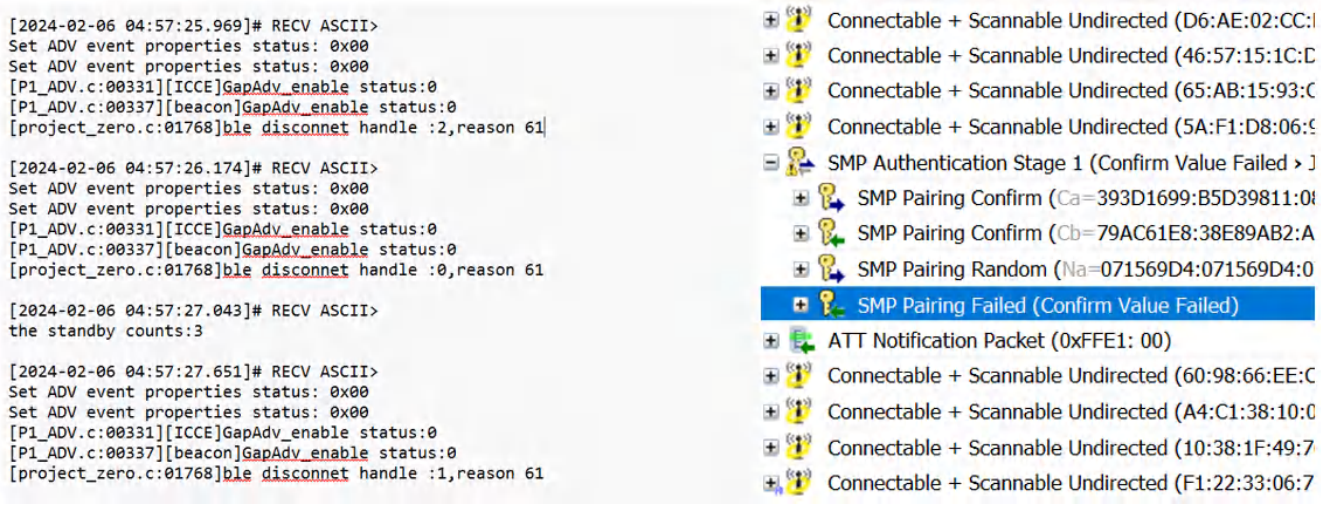

Figure 2-4. UART log and over-the-air packet sniffer log in Case 2

In the log of this case, the disconnection error code 0x3D continuously appears for the encrypted connection, which stands for the 0x3D MIC failure. Although it can recover immediately after a reboot, it reoccurs very quickly. The customer reported that when the problem occurred, the cc2642 was connected to only one test device, and the test device ran a basic example program without extra hash operations or repeated encryption. We aligned the over-the-air packet sniffer log, the AES engine startup log captured by the logic analyzer, and the debugging log output by the UART at the same timestamp, and found that the location where the problem occurred (a gas station) possessed a very large volume of Bluetooth RPA addresses undergoing scanning.

Once the CC2642 was scanned, a connection attempt would be made. The moments of frequent AES engine invocations at the problem location also basically coincided with the timestamps of being scanned. The problem was ultimately located as the operation to resolve RPA addresses being invoked too frequently, causing the encryption operation to fail to be allocated encryption resources. The problem was also resolved by adjusting the connection logic when being scanned. This has played a very important inspiring role in our system design as well. Just as debugging requires high alertness to the impacts caused by the external environment, designing a robust Bluetooth system should fully consider compatibility with special external environments.

2.2 Abnormal Key Negotiation—DHKey Verification Failure (DHKey check fail)

A DH Key (Diffie-Hellman Key) verification failure during the BLE pairing process is essentially an anomaly in the ECDH (Elliptic Curve Diffie-Hellman) key negotiation flow, which is a core step in establishing a secure connection. Although such faults frequently manifest as key mismatches, in practical applications, we are often not facing a key pair mismatch problem within the narrow definition of the DH Key verification flow itself. More often, the problem occurs when other behaviors of the chip during key resolution interfere with key verification. I will classify the main reference troubleshooting directions into two categories according to application-side impacts and protocol-stack-side impacts, listed as follows:

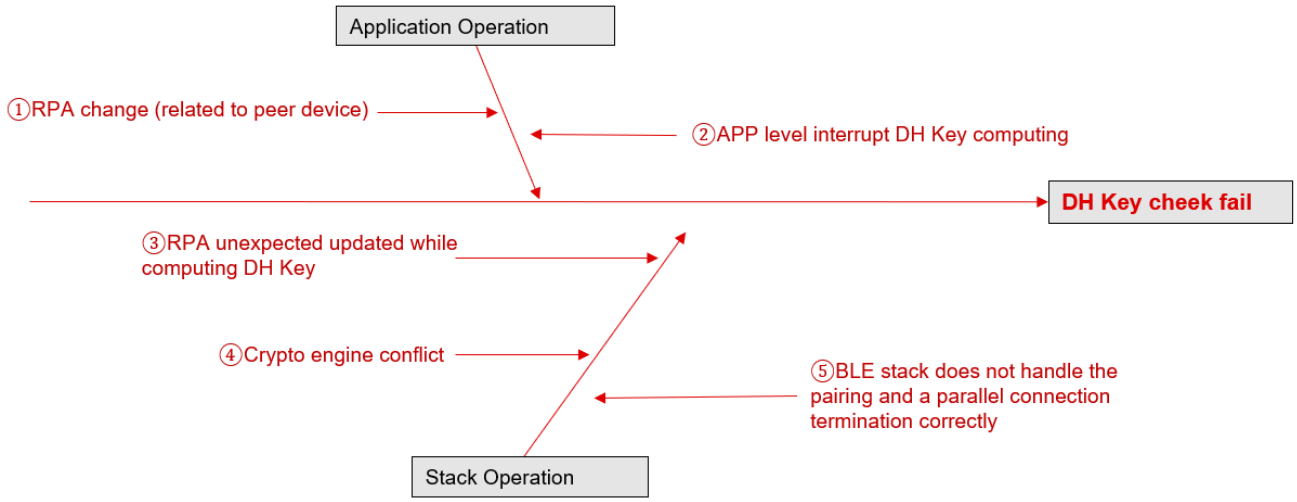

Figure 2-5. Root cause analysis of DH Key check fail

Similarly, during debugging, troubleshooting should proceed from the outside inward, corresponding to the logical relationships of the location where the problem occurs:

- Test the RF environment: Conduct replication experiments in an environment with low interference (a shielded room or a quiet test environment) to eliminate signal interference. Simultaneously perform an RF inspection of the environment where the problem occurs, such as conflicts with surrounding WiFi or other 2.4GHz devices, whether there are special Bluetooth data packets, and so forth.
- Verify secure connection support: Capture full-link interaction data through an Ellisys sniffer, focusing on: the integrity of public key data during the Pairing Public Key exchange phase; whether both parties synchronously enable the LE Secure Connections mode; the negotiation consistency of the ECDH algorithm identifier (such as the P-256 curve). Compare the protocol logs of both parties to verify whether the public key byte order processing complies with specifications (big-endian/little-endian), and the synchronization state of the key derivation counter (Counter).
- Check abnormal behavior of the chip when the problem occurs: such as whether a disconnection occurred (⑤), whether an address change occurred (①③), and whether there was a reset or a high-priority interrupt (②).

Case 3 (Solved issue of TI BLE stack**)**:

Problem Replication:

Device A connects and pairs with CC2642 as a master node, and phone B repeatedly connects and disconnects with CC2642 as a master node. A DH Key check fail occasionally occurs during the pairing of device A.

Root Cause of the Problem:

When the connection was disconnected, the Bluetooth protocol stack failed to check whether the connection handle of the disconnection was identical to the connection handle undergoing pairing, thereby causing the protocol stack to erroneously delete the information of the device undergoing pairing, which in turn resulted in the corresponding public key not being found during the DH key match calculation, leading to a DH Key check fail.

The analysis of this problem began with the discovery that the pairing records were lost, and subsequently, by adding AES engine logs to check the source and message of the delete pairing operation when the problem occurred, it was located to be exactly coincident in time with the disconnection behavior. This issue has been fixed in SimpleLink CC13xx CC26xx SDK v7.10.02.23. And it was horizontal-rolled to all old SDK versions in the form of a fix patch.

Case 4 (Solved issue of APP code**):**

Problem Replication:

While the pairing process is ongoing, the RPA of the CC2642 device (or the peer) changes. That is, a DH key check fail occurs.

Root Cause of the Problem:

The calculation is performed using the new RPA when the DH key check is carried out, rather than the RPA at the time of connection establishment. This appears to conform to Bluetooth logic, but in fact, it violates the definition of the address during pairing in official Bluetooth documents. In fact, it is explicitly stipulated in Bluetooth Core Specification Vol 3 Part H Section 2.2.7: "*BD_ADDR_C is the device address of the Central and BD_ADDR_P is the device address of the Peripheral. The device addresses are the values used during connection setup. The least significant bit in the most significant octet in both BD_ADDR_C and BD_ADDR_P.*" Therefore, the solution to the problem is to add a dedicated unit to store the address at the time of connection establishment.

- Check encryption resource utilization: Same as the discussion in Section 2, check the hardware resource competition between the encryption engine preemptions and operations such as RPA resolution and scan responses (such as the CC2642 AES module).

2.3 Key Management Defect—PIN/Key Loss (PIN or Key missing)

This error can appear either when pairing fails, typically indicating that the device cannot find the PIN code or key required for pairing. It can also appear when two already paired devices undergo encryption, typically indicating that the device cannot find the required LTK.

2.3.1 Key Missing During Pairing Phase

Regarding the PIN or Key missing problem occurring during pairing, one should trace the chip -> air -> peer chip path of the pairing code from the logic of pairing code negotiation -> generation -> interaction -> processing, combined with over-the-air packets and device print information, focusing on:

- Mismatch of pairing modes between devices: Inconsistent pairing modes between both parties (such as Passkey Entry vs Just Works), and incompatibility between Security Requests and Pairing Capabilities.
- Pairing initialization error: Security requirements initialization of the device failed to complete successfully, resulting in the PIN/key not being generated or saved. Inability to pair normally with the peer device.
- Packet tampering: During transmission, communication data packets may become incomplete or inconsistent due to noise, interference, or malicious tampering.
- Pairing flow error: For example, the peer device has a pairing record with this device, but it has been deleted on this device and the pairing process was not restarted with the peer device, resulting in pairing failure due to no available key locally. Multiple pairing modes can be tested (such as Just Works, Passkey Entry, OOB Pairing).

2.3.2 LTK Missing During Encryption Phase

On the other hand, the PIN or Key missing problem occurring during the encryption process of already paired devices generally exhibits the following characteristics:

- It is not a system-level design problem, making it difficult to find loopholes and compensate from a simple review of engineering logic.
- High element of randomness: The laboratory replication rate is typically lower than 1%.
- Environmental correlation: Possesses a potential coupling with client application logic (such as rapid address refreshing).
- Interference with self-healing mechanisms: Under normal circumstances, for pairing failure or encryption failure type problems, the device will self-repair after the problem occurs by deleting pairing records and re-pairing during the next connection. Therefore, the core scene conditions when the problem occurs may have been destroyed, causing difficulties in information acquisition.

Hence, when dealing with this type of problem, one should grasp the core logic of the PIN or Key missing error code. For TI CC2642, the sole reason for reporting a PIN or Key missing error during the encryption process is that the LTK cannot be found. Therefore, troubleshooting directions can still be provided from the two dimensions of application-side code impact and protocol-stack-side impact targeting the possibility of LTK loss:

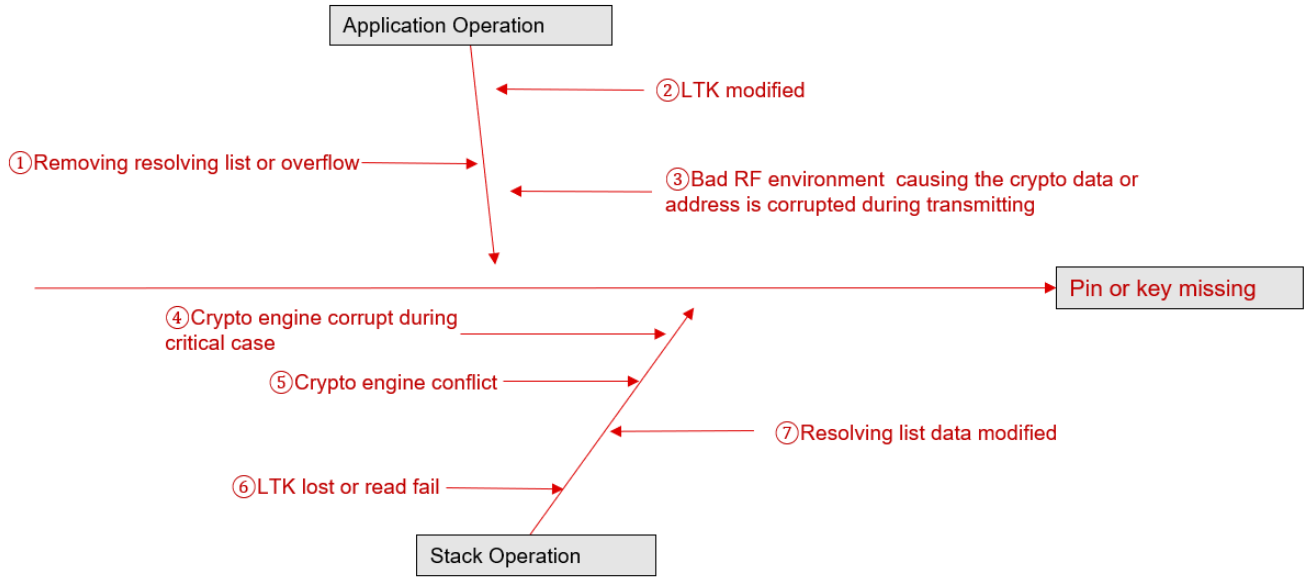

Figure 2-6. Root cause analysis of PIN or Key missing

- Check whether the problem stems from a physical loss of the LTK: Inspect the flash storage areas before and after the problem occurs, read the storage location of the LTK in flash, and check whether errors or tampering occurred. And inspect the operation source accordingly. (②⑥)
- Test the RF environment: Conduct replication experiments in an environment with low interference (a shielded room or a quiet test environment) to eliminate signal interference. Simultaneously perform an RF inspection of the environment where the problem occurs, such as conflicts with surrounding WiFi or other 2.4GHz devices, whether there are special Bluetooth data packets, and so forth. (③)
- Test whether the resolving list used in RAM to resolve the peer address experienced errors or tampering when the problem occurred: Since un-interacted address changes during the pairing process will cause the peer address currently undergoing pairing operations to be unfound during pairing, the legitimacy of the pairing operation must be checked. (①⑦)

Case 5:

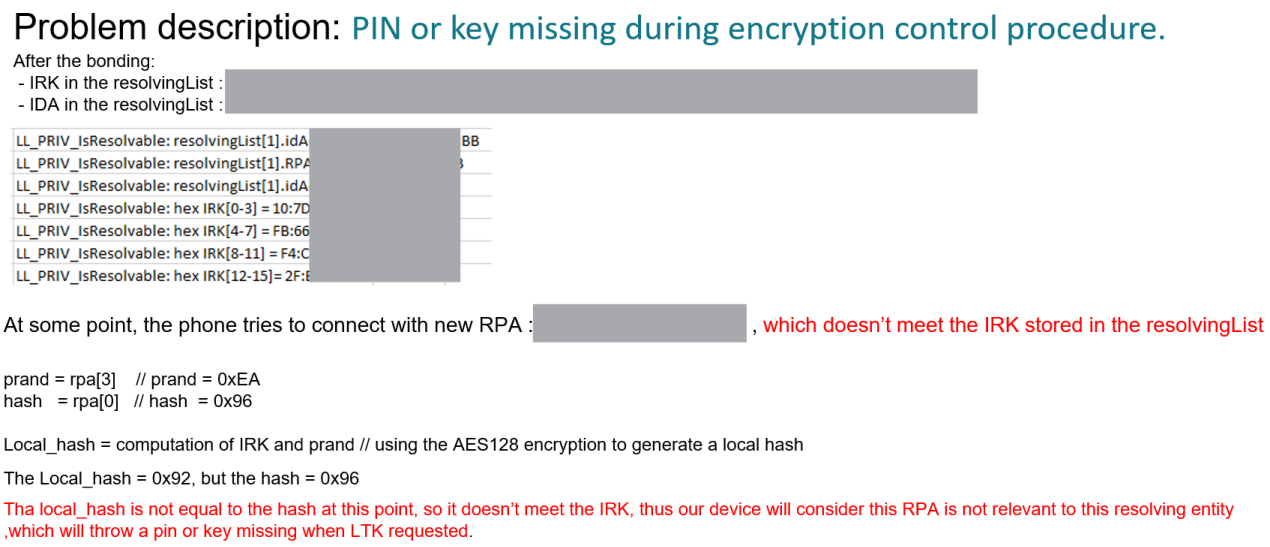

Figure 2-7. Address analysis of Case 5 sniffer log

In the example of the figure above, the phone acts as a master node to connect, pair, and bond with CC2642, but a PIN or Key missing encryption failure occasionally occurs during repeated disconnections and reconnections. By checking the abnormal conditions when the problem occurred, we found that the phone as a master node not only updated its own RPA, but also changed the generated hash value of the RPA. This alteration prevented the phone's IRK recorded by CC2642 from identifying the current phone as an already paired system, thereby generating the PIN or key missing problem.

- Check encryption resource utilization: Same as the discussion in Section 2 (⑤).
- Other associated applications: In an actual project, the identification of a peer device is sometimes not only related to Bluetooth pairing and bonding, but also involves other specifications or verifications that the customer needs to pass for identification. If there is a user verification flow being conducted during an encryption event among these, it is also possible for that verification error to lead to a PIN or Key missing problem.

Case 6:

When using CC2642, the customer designed the following two-step verification system:

Problem Replication:

1. Pair CC2642 with an external central node ->
2. The Central node delivers its own carried authentication information ->
3. CC2642 stores the pairing bonding information (hereinafter referred to as R1) and the authentication information (hereinafter referred to as R2) together in the flash corresponding to this central node
4. A PIN or key missing occasionally occurs during repeated disconnections and reconnections

Root Cause of the Problem:

The analysis of this problem was initially confined to whether the pairing bonding information was lost and whether there were read errors. However, under circumstances where repeated verifications of storage and reading were unlikely to fail for the mature CC2642 chip, we ultimately focused our attention on the difference between R2 and R1. The flash driver of CC2642 divides into two steps when deleting a certain piece of stored information

1. Set the valid bit of the original storage location of the information to invalid
2. Rewrite the information once with the header of the information, but with the content as all zeros

However, for the R2 information operated by the user themselves, the operation of rewriting all zeros was not performed. This causes R1 to remain valid when a pairing record deletion occurs, except its value is all zeros,

but R2 is invalid. At this time, if a new pairing occurs, the handle corresponding to its new R1 and R2 will fail to match. Leading to the failure of the customer's second-step verification, which in turn leads to PIN or key missing.

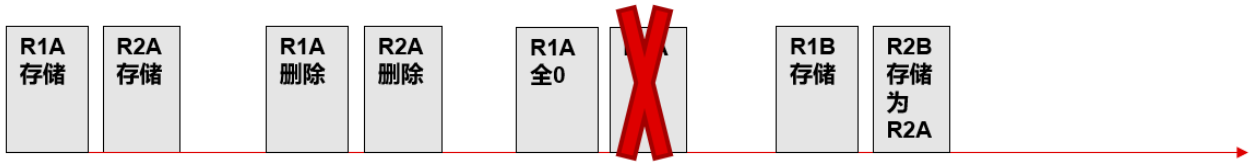

Figure 2-8. Flow analysis of Case 5 root cause

3 A Solution for Encryption Resource Conflicts

Based on the analysis of the various pairing anomalies described above, it is evident that the encryption resource competition of CC2642 has become one of the cores of systemic security issues. The hardware encryption module of this chip adopts an exclusive architecture design, which must simultaneously carry out critical tasks such as RPA address resolution, pairing key negotiation, and link data encryption, whereas the application layer can neither monitor its workload status in real time nor implement priority scheduling strategies. Targeting this hardware limitation, this paper proposes a software-based AES encryption collaborative scheme—by building an observable and schedulable software encryption channel, it forms a complementary architecture with the hardware module. This scheme fully utilizes the computing power of the CC2642's Cortex-M4 core to implement the following innovative designs at the application layer:

- Encryption task grading mechanism: Real-time demanding operations such as RPA resolution can be retained in the hardware module for execution, while non-real-time tasks such as ECDH calculation and LTK derivation during the pairing flow are migrated to software AES for implementation.
- Dynamic resource arbiter: Based on the priority inheritance feature of FreeRTOS, it provides the possibility of optional software channel shunting when hardware module overload is detected.
- Mixed encryption pipeline: Optimizes the AES-CCM encryption link at the instruction level. This hybrid architecture not only circumvents hardware resource conflict risks, but also provides developers with advanced debugging capabilities such as encryption task timing analysis and conflict event tracing through the status visualization interface at the software layer, significantly enhancing the security and robustness of Bluetooth connections in complex RF environments.

3.1 Environment Selection and Configuration

To implement AES-256 encryption on FreeRTOS, the following work needs to be completed:

- Select an encryption library. In a FreeRTOS environment, a lightweight encryption library can be chosen, such as mbedTLS, WolfSSL, or TinyAES. This paper uses TinyAES because it is lightweight, easy to use, and particularly suitable for embedded systems.
- Download the source code from the TinyAES GitHub project. TinyAES supports modes such as ECB (Electronic Codebook), CBC (Cipher Block Chaining), and CTR (Counter Mode). The example in this paper uses the CBC mode.
- Introduce the following files into the FreeRTOS project directory: aes.h/aes.c
- Modify FreeRTOS configuration: Ensure that the stack and task priority settings in the FreeRTOS configuration file (FreeRTOSConfig.h) are sufficient to support encryption operations. For example, the stack size can be adjusted according to system requirements, and a sufficient total memory size can be provided.

3.2 Implementing AES-256 Encryption and Decryption Using TinyAES Library

- Add implementation of AES encryption task

```
 encrypt_task.c
#include "FreeRTOS.h"
#include "task.h"
#include "aes.h"
#include <stdio.h>
#include <string.h>

#define AES_BLOCK_SIZE 16

// AES-256 密钥(必须为 32 字节)
static uint8_t key[32] = {xx};

// 初始化向量(IV), 必须为 16 字节
static uint8_t iv[AES_BLOCK_SIZE] = {yy};

void EncryptTask(void *pvParameters) {
  uint8_t plaintext_block[AES_BLOCK_SIZE] = {0};
  memcpy(plaintext_block, plaintext, strlen(plaintext));

  uint8_t ciphertext_block[AES_BLOCK_SIZE] = {0};
  uint8_t decrypted_block[AES_BLOCK_SIZE] = {0};

  // AES CBC 加密上下文
  struct AES_ctx ctx;
  AES_init_ctx_iv(&ctx, key, iv);

  // 加密
  AES_CBC_encrypt_buffer(&ctx, plaintext_block, AES_BLOCK_SIZE);
  memcpy(ciphertext_block, plaintext_block, AES_BLOCK_SIZE);

  // 解密(重新初始化上下文以确保 IV 被重置)
  AES_init_ctx_iv(&ctx, key, iv);
  AES_CBC_decrypt_buffer(&ctx, ciphertext_block, AES_BLOCK_SIZE);
  memcpy(decrypted_block, ciphertext_block, AES_BLOCK_SIZE);

  vTaskDelay(pdMS_TO_TICKS(1000)); // 任务延时
```

- Create task in main function

```
 main.c
#include "FreeRTOS.h"
#include "task.h"
#include "encrypt_task.h"

int main(void) {
  // 初始化硬件
  // 创建加密任务
  xTaskCreate(EncryptTask,     // 任务函数
              "Encrypt Task",  // 任务名称
               512,            // 堆栈大小(字节)
               NULL,          // 任务参数
               1,            // 优先级(任务越重要越高)
               NULL);       // 任务句柄
  // 启动 RTOS 调度器
  vTaskStartScheduler();
}
```

The advantages of this method lie in:

- The lightweight software library provides an encryption option that consumes almost no CPU computing or storage power. In fact, the TinyAES library occupies very lightweight code space, requiring approximately ~2-3 KB of code space in the absence of floating-point operations (including AES implementation and initialization vector support), which also depends on the selected encryption mode (the CBC mode takes slightly more code than the CTR mode). If the application code contains critical print statements (such as printf) and local data, it will occupy approximately ~2-4KB of extra Flash and ~2.5KB of extra RAM.
- The software method allows developers to optimize algorithms according to specific needs. For example, optimizing confusion through look-up tables, introducing chaotic mapping to enhance key security, or dynamically adjusting key generation strategies. Hardware implementations usually solidify algorithm logic, making it difficult to modify. Adopting different software libraries can support more encryption modes (such as ECB, CBC, CTR, etc.), and developers can flexibly choose and extend encryption algorithms not supported by the AES engine according to requirements.
- The encryption logic implemented in software can quickly fix vulnerabilities or update algorithms through firmware upgrades (such as OTA), whereas the hardware accelerator requires adjusting the design logic of the entire application to seek circumvention as introduced previously. The cost is greater.

3.3 System-Level Optimization Exploration

Choosing to use a software method for encryption and decryption work, compared to simply invoking the resources opened by TI's instance code, requires discussing some system burdens and optimization effects added to application development, including:

3.3.1 Memory Allocation Inspection

- Stack size and task priority: Ensure that sufficient stack space is allocated to the encryption task to avoid task crashes when executing encryption algorithms, and set appropriate priorities based on the importance of the tasks.
- On resource-constrained systems, try to reuse memory areas to store encrypted data to avoid excessive dynamic memory allocation. Consideration can be given to using dynamically allocated buffers to handle larger lengths of plaintext and ciphertext according to the required algorithms. Considering the important position of memory analysis in software design, the method for estimating RAM resources taking the example code as a case is provided as follows:

1. RAM Occupation of FreeRTOS Kernel

   Task Control Block (TCB): The TCB structure required by each task, including the task stack pointer, status information, and task priority, etc. The size of a TCB is typically about 60-100 bytes.

   Task Stack: The stack space for each task. It should be noted that the size of the stack should be set sufficient to run the instructions required by the task, including interrupt services, local variables of function calls, and intermediate operations, etc. In the example code, a stack space of 512 words is allocated for EncryptTask (specified as 512 x 4 bytes = 2KB in xTaskCreate()).

Global Heap Space (if pvPortMalloc() dynamic allocation mode is used): The FreeRTOS kernel requires dynamic memory allocation implementation, such as memory required by queues, semaphores, or other functions. This part of the requirement depends on the specific system implementation, but it is not involved in extra dynamic allocation in this example.

2. RAM Occupation of TinyAES Library

   AES Context Structure (struct AES_ctx): Contains the key and initialization vector, used to perform encryption and decryption operations. About 60 bytes.

   AES Block Key Expansion (internal variables used by the key scheduling algorithm): For AES-256, a total of 240 bytes of key expansion data needs to be stored.

3. Application Data

   There is global data in the application code: AES key (32 bytes), initialization vector IV (16 bytes), plaintext, ciphertext, and decryption buffers each allocated 16 bytes (which is the aligned AES block size in the example). Totaling 96 bytes.

3.3.2 Padding Processing and Thread Safety

If the plaintext data is not a multiple of the AES block size (16 bytes), padding logic needs to be manually added, such as PKCS7 padding. If the actual stack requirement is less than 512 words, the task stack size can be shrunken. For example, evaluate the stack usage of the task at runtime through a stack analysis tool (uxTaskGetStackHighWaterMark), and gradually adjust it to an appropriate size (a typical AES task may only require a stack of around 300 words). In addition, the TinyAES library itself is not thread-safe. If multiple tasks invoke the encryption algorithm simultaneously, synchronization mechanisms between tasks (such as Mutex) can be used.

4 Conclusion

This paper succinctly analyzes the core principles and typical fault modes of the Bluetooth Low Energy (BLE) pairing security mechanism, systematically constructs a full-link diagnostic framework from protocol logic to hardware resource management for some common difficult problems, and provides design recommendations for system design developers in combination with instances. Concurrently, targeting the common encryption resource conflict dilemma among several difficult pairing encryption problems of the TI CC2642R-Q1 chip, a software-alternative high-freedom solution is proposed to achieve resource competition circumvention. It provides system design developers with the feasibility of coping with system design requirements using a software-hardware collaborative encryption pairing scheme.

5 References

1. Bluetooth Core Specification v4.2 tiny-AES-c
2. CC13x2, CC26x2 SimpleLink Wireless MCU Technical Reference Manual (Rev. G)

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025