Google Keystore API and Crypto Keys

The Android Keystore API ensures cryptographic keys are generated and remain inside a secure element through a combination of hardware abstraction, developer-specified flags, and cryptographic attestation. It doesn't guarantee generation in a specific "Secure Element" (SE) per se, but rather in a hardware-backed keystore, which is most commonly a **Trusted Execution Environment (TEE)** and can sometimes be a dedicated SE.

Here’s a technical breakdown of how it works:

# 1. Hardware Abstraction & the Keymaster HAL

The Android Keystore system doesn't interact with the secure hardware directly. Instead, it communicates through the **Keymaster Hardware Abstraction Layer (HAL)**. This HAL is a standard interface that device manufacturers implement to allow the Android OS to use their specific secure hardware (the TEE or SE).

When the app calls the Keystore API to generate a key, the request is passed to the Keymaster HAL. The HAL then forwards this request to the secure hardware, which is responsible for executing the key generation command. This design ensures that the main Android OS never handles the key material itself.

# 2. Developer Specifications: setIsStrongBoxBacked()

When generating a key, a developer can explicitly request that it be stored in the most secure hardware available. This is done using the setIsStrongBoxBacked(true) method on the KeyGenParameterSpec.Builder.

* **StrongBox Keymaster** is a specific implementation of the Keymaster HAL that resides in a dedicated, tamper-resistant secure element (SE). It's designed for the highest security needs, like storing private keys for financial transactions.
* **TEE Keymaster** is the standard implementation running in the phone's Trusted Execution Environment. While highly secure and isolated from the main OS, it's generally considered a step below a dedicated SE like StrongBox.

By calling setIsStrongBoxBacked(true), we are telling the Android system to use the StrongBox Keymaster. If the device doesn't have a StrongBox-compliant SE, the key generation will fail. This provides a strict guarantee to the developer.

Java

// Example of requesting key generation in StrongBox

KeyGenParameterSpec spec = new KeyGenParameterSpec.Builder(

"my\_strongbox\_key\_alias",

KeyProperties.PURPOSE\_SIGN | KeyProperties.PURPOSE\_VERIFY)

.setAlgorithmParameterSpec(new ECGenParameterSpec("secp256r1"))

.setDigests(KeyProperties.DIGEST\_SHA256)

// This is the crucial line

.setIsStrongBoxBacked(true)

.build();

KeyPairGenerator.getInstance("EC", "AndroidKeyStore").run {

initialize(spec)

generateKeyPair()

}

# 3. Key Attestation: Cryptographic Proof 🛡️

So, how can we be certain the hardware actually followed the rules? This is where **key attestation** comes in.

When we generate a key, we can ask the Keystore to provide a signed certificate chain for that key. This attestation certificate contains detailed information about the key and the environment in which it was generated.

The process works like this:

1. We generate a key pair and request attestation.
2. The secure hardware (TEE or StrongBox) generates the key pair. It then creates and signs an attestation certificate using a special **attestation key** that was fused into the hardware during manufacturing.
3. This certificate includes crucial extensions that describe the key's properties. Our server can then inspect this certificate to verify:
   * **teeEnforced / strongBoxEnforced Flags:** The certificate data will contain lists of properties that are cryptographically guaranteed by either the TEE or StrongBox. This proves that properties like isStrongBoxBacked were enforced by the secure hardware.
   * **Root of Trust:** By validating the certificate chain up to a trusted root certificate (like a Google root certificate), our server can confirm that the attestation was signed by genuine, certified hardware.

By parsing the attestation certificate on our backend server, we get cryptographic proof that the key was generated and is protected by the device's secure hardware, fulfilling the requirements we set in the KeyGenParameterSpec. This prevents a compromised Android OS from faking a hardware-backed key.