WV Modular DRM Version 14 Delta

Changes from Version 13 to 14

December 14, 2017

Table of Contents

References

DASH - 23001-7 ISO BMFF Common Encryption, 3rd edition.

DASH - 14496-12 ISO BMFF, 5th edition.

W3C Encrypted Media Extensions (EME)

WV Modular DRM Security Integration Guide for Common Encryption (CENC) : Android Supplement

WV Modular DRM Security Integration Guide for Common Encryption (CENC)

Audience

This document is intended for SOC and OEM device manufacturers to upgrade an integration with Widevine content protection using Common Encryption (CENC) on consumer devices. In particular, if you already have a working OEMCrypto v13 library, and want to upgrade to OEMCrypto v14, then this document is for you. If you are starting from scratch, you should read WV Modular DRM Security Integration Guide for Common Encryption (CENC) Version 14.

Overview

There are several new features required for OEMCrypto version 14. The following sections discuss the main new features and give some idea why the new feature is being added. You should refer to WV Modular DRM Security Integration Guide for Common Encryption (CENC) for the full documentation of the API -- this document only discusses changes to the API. In this document, when we say "OEMCrypto shall …" we mean that "an implementation of the OEMCrypto library shall …".

Definitions

CENC - Common Encryption

DASH - Dynamic Adaptive Streaming over HTTP

CDM - Content Decryption Module -- this is the software that calls the OEMCrypto library and implements CENC.

PST - Provider Session Token - the string used to track usage information and off line licenses.

SRM - System Renewability Message. Contains blacklist of revoked HDCP keys.

API Version Number

```
uint32_t OEMCrypto_APIVersion();
```

This function shall now return 14. If it returns less than 14, then the calling code will assume that OEMCrypto does not support the new v14 features. Depending on the platform, the library may be run in backwards compatibility mode, or it may fail. See the supplemental document for your platform to see which version of OEMCrypto is required.

Encryption Scheme Signaling

The parameter Cipher Mode will no longer be available when the license is loaded. Instead, it will be available when the key is first used. That means the parameter cipher_mode will no longer be in the structure OEMCrypto_KeyObject used in OEMCrypto_LoadKeys. Instead, it will be a new parameter to OEMCrypto_SelectKey. Although it will be rare for content to change the cipher mode during playback, it is possible. Content providers will be warned that using different cipher modes with the same key could cause performance problems because crypto hardware may need to be re-initialized.

The array of key data passed into OEMCrypto_LoadKeys, key_array, will now be:

```
typedef struct {
  const uint8_t* key_id;
  size_t key_id_length;
  const uint8_t* key_data_iv;
  const uint8_t* key_data;
  size_t key_data_length;
  const uint8_t* key_control_iv;
  const uint8_t* key_control;
} OEMCrypto_KeyObject;
```

The signature of SelectKey will now be:

```
OEMCryptoResult OEMCrypto_SelectKey(OEMCrypto_SESSION session,
                                    const uint8_t* key_id,
                                    size_t key_id_length,
                                    OEMCryptoCipherMode cipher_mode);
```

This change is occuring because the cipher mode used to encrypt the content is typically stored with the content itself in the PSSH, rather than with license data on the server.

Report Analog Output

OEMCrypto will report whether the device supports analog output or not. This information will be sent

to the license server, and may be used to determine the type of license allowed.

```
uint32_t OEMCrypto_GetAnalogOutputFlags();
```

Returns a bitwise OR of the following flags.

- 0x0 = OEMCrypto_No_Analog_Output -- the device has no analog output.
- 0x1 = OEMCrypto_Supports_Analog_Output - the device does have analog output.
- 0x2 = OEMCrypto_Can_Disable_Analog_Ouptput - the device does have analog output, but it will disable analog output if required by the key control block.
- 0x4 = OEMCrypto_Supports_CGMS_A - the device supports signaling 2-bit CGMS-A, if required by the key control block

This new function is for reporting only. It is paired with the existing key control block flags Disable_Analog_Output and CGMS.

SHA-1 Collision

This is not a change, but an explanation. Some questions have been raised regarding recently announced practical SHA-1 collision attacks. The question is whether the discovery of a method to generate SHA-1 collisions weakens any of the Widevine protocols, and whether we are planning to phase out SHA-1. SHA-1 is used with the HMAC algorithm to generate a signature for some messages. The HMAC algorithm is not dependant on collision resistance. With this in mind, Widevine is not planning to remove usage of SHA-1 at this time.

RSA Keys Using Carmichael Numbers

OEMCrypto should accept a certificate that uses the Carmichael totient. Factors of the Carmichael totient can be used as the public and private RSA keys instead of factors of the Euler totient. By using the Carmichael totient, we can improve performance and security of the RSA algorithm. The RSA algorithm itself does not change, but some verification algorithms will need modification.

Random Number Verification

OEMCrypto_GetRandom and OEMCrypto_GenerateNonce should be random enough. Widevine will distribute a series of tests that verify these random numbers are close to uniform.

Pointer Alignment

Pointers to integer values, (such as uint32_t* nonce), can be unaligned, because these pointers point to values contained in a message from the server. We cannot guarantee that these values will be aligned on a word boundary. TIt will be the responsibility of OEMCrypto to copy and align any data if it is necessary for the target architecture.

In particular, implementers should be careful with the nonce in the key control block, and the nonce pointer arguments in OEMCrypto_RewrapDeviceRSAKey30 and OEMCrypto_RewrapDeviceRSAKey.

Error Reporting

The following are new requirements for reporting error conditions.

If RefreshKeys or SelectKey is given a key id that is not loaded in the session, OEMCrypto will return the error OEMCrypto_KEY_NOT_LOADED.

The header and docs will be updated to list all of the error codes that could be returned by each function.

C++ and C Interface

This is not a change, but an explanation. The OEMCrypto API is specified as a C interface so that we can minimize minor compiler version differences on platforms. In particular, on Android, we use dlopen to load liboemcrypto.so, which is difficult if there is C++ mangling of function names.

With that said, the reference implementation, which is also used as a mock for testing, is written in C++. Google is not restricting the language that the OEMCrypto library is written in.

Load Test Keybox

The function OEMCrypto_LoadTestKeybox will now take a parameter containing a test keybox in binary form. The binary keybox format is defined in the full OEMCrypto API document under the heading “Keybox Definition”. This change is being made to avoid having any test keybox embedded in production code or in the reference implementation.

```
OEMCryptoResult OEMCrypto_LoadTestKeybox(const uint8_t *buffer, size_t length);
```

The keybox will be in binary format. OEMCrypto **cannot** assume that this keybox is the same as previous keyboxes used for testing.

Devices that use an OEM Certificate instead of a keybox (i.e. Provisioning 3.0) do not need to support this functionality, and may return OEMCrypto_ERROR_NOT_IMPLEMENTED.

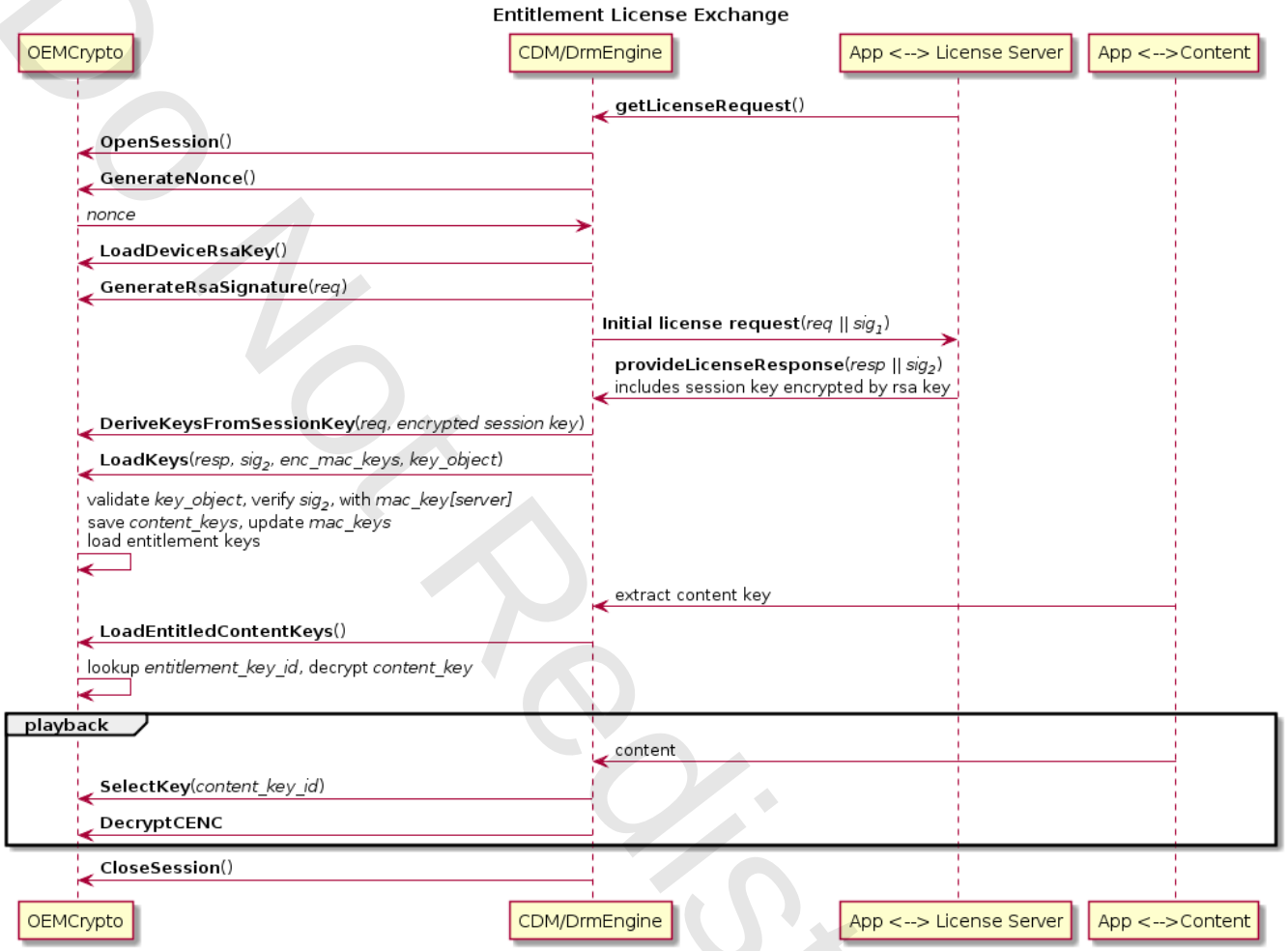
Key Control Block Changes

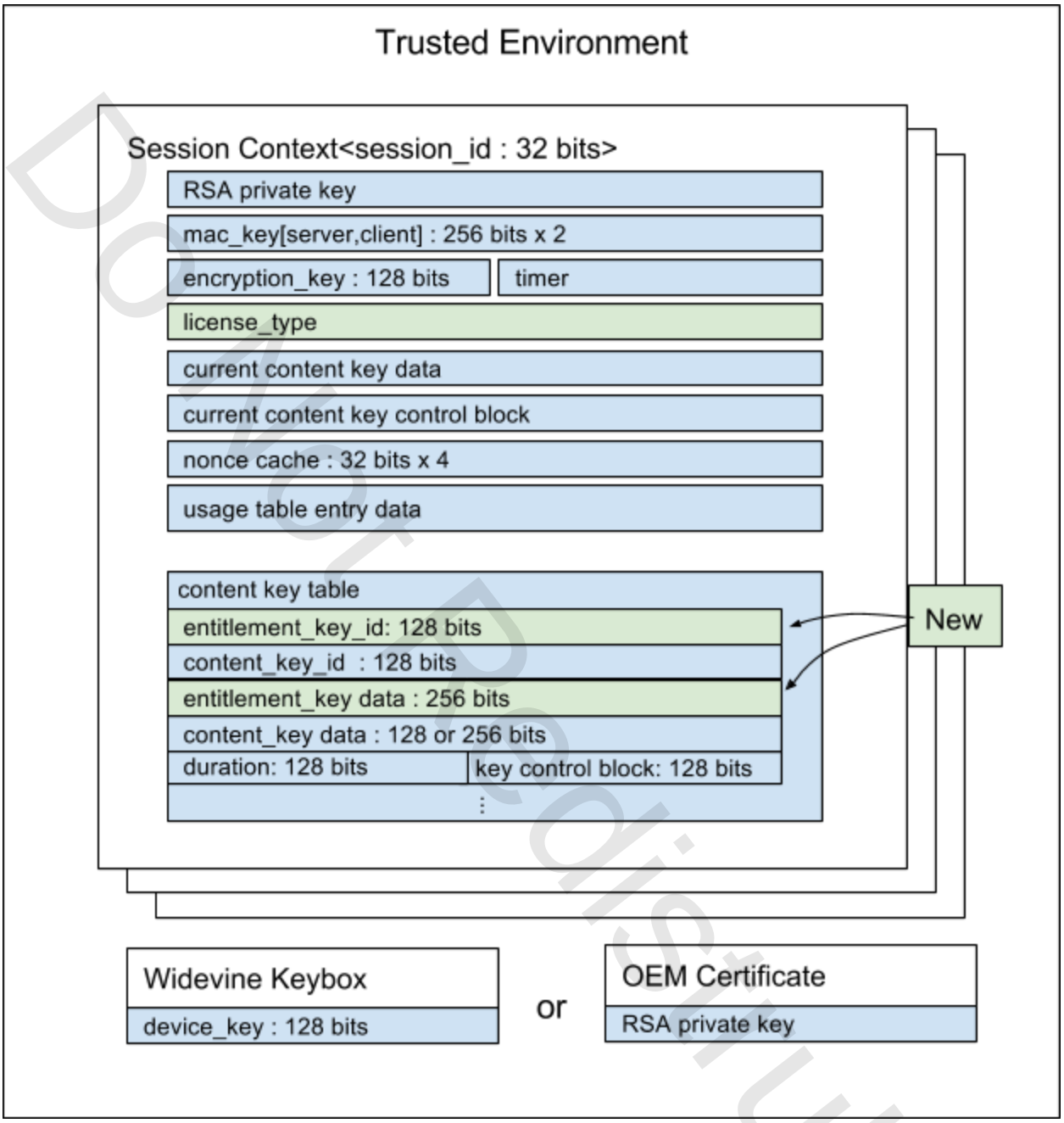
OEMCrypto should now accept the verification string “kc14”. OEMCrypto can expect the server to set the verification code to “kc14” in all new licenses. Devices that are updated in the field are expected to load older offline licenses with previous values of the verification string: “kctl”, “kc09”, … “kc13”.

Entitlement and Secondary License Support

An entitlement license is a new way to group multiple licenses together. Each piece of content in the group will have its content keys encrypted by an entitlement key and embedded in the content. The device does not have to send a license request to a server for each piece of content. Instead, the entitlement key will be delivered to the device in a single entitlement license. The device is then entitled

to decode all of the content covered by that entitlement license. An entitlement license may have several entitlement keys. The session's key table will now contain both a content key and an entitlement key. The content key and the entitlement key will each have a key id. The content key and entitlement key will share a duration and key control block.





Let us consider an example that illustrates why the number of entitlement keys may not match the number of content keys. There might be one entitlement key for HD video, one for SD video, and one for audio. A piece of content might have several keys embedded in it:

- One content key encrypted by the HD video entitlement key that is for decrypting HD video content.
- One content key encrypted by the SD video entitlement key that is for decrypting SD video content.
- One content key encrypted by the audio entitlement key that is for decrypting audio content.

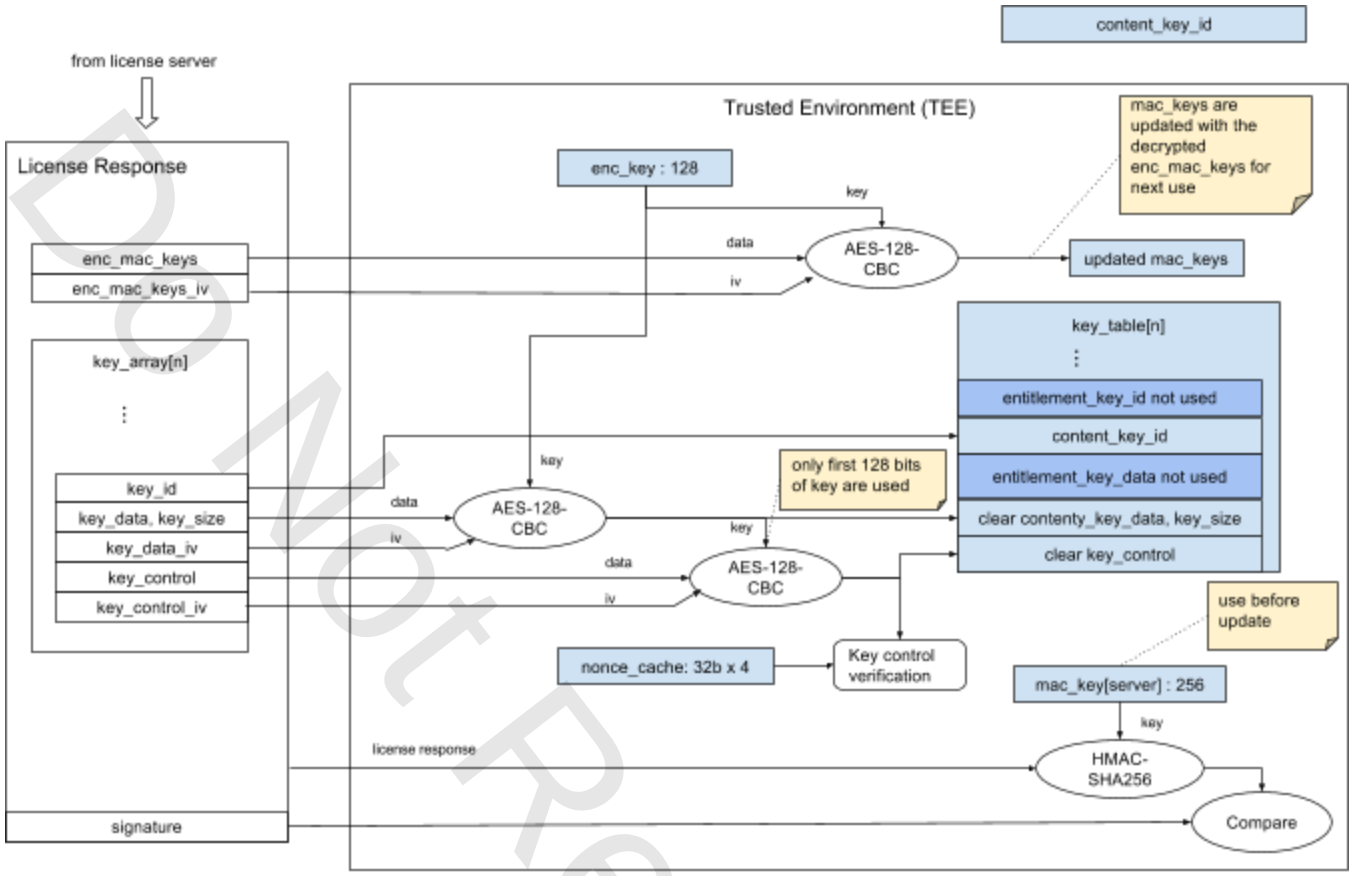
Some content may have a subset of these keys -- for example some content may have only audio and

SD video. In this case, the entitlement license might have all three keys, but only two content keys would be loaded into the session. Some entitlement licenses may have only a subset of these keys -- for example the customer did not pay for HD content. In this case, the entitlement license would only have two keys, and only two of the content keys will be loaded.

The application, and the CDM layer are responsible for deciding if the license is a traditional widevine content license, or an entitlement license. It will signal to OEMCrypto which type of license to load using the parameter license_type to OEMCrypto_LoadKeys. Otherwise, the parameters to LoadKeys are unchanged.

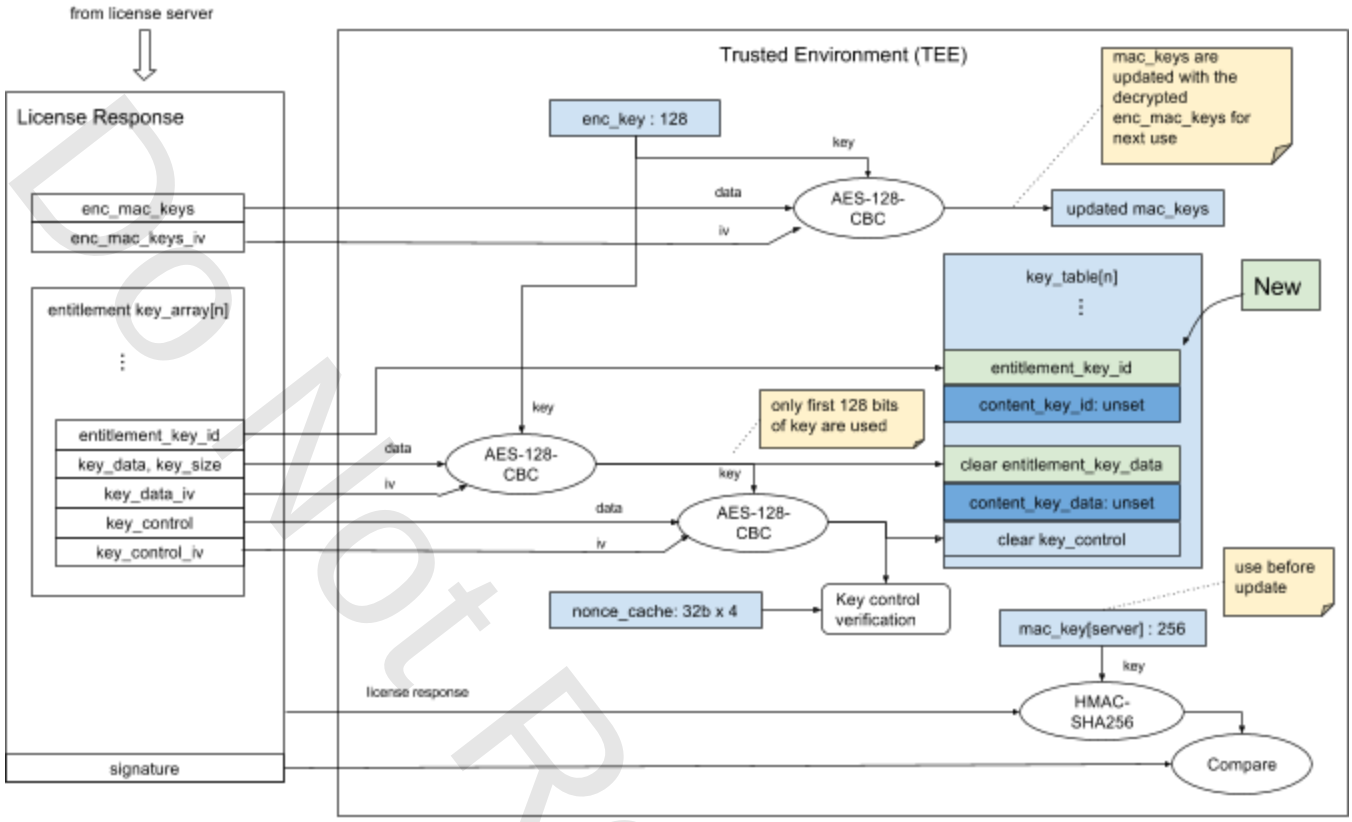
```
typedef enum OEMCrypto_LicenseType {
  OEMCrypto_ContentLicense = 0,
  OEMCrypto_EntitlementLicense = 1
};

OEMCryptoResult OEMCrypto_LoadKeys(
    OEMCrypto_SESSION session,
    const uint8_t* message,
    size_t message_length,
    const uint8_t* signature,
    size_t signature_length,
    const uint8_t* enc_mac_keys_iv,
    const uint8_t* enc_mac_keys,
    size_t num_keys,
    const OEMCrypto_KeyObject* key_array,
    const uint8_t* pst,
    size_t pst_length,
    const uint8_t* srm_requirement,
    OEMCrypto_LicenseType license_type
    );

typedef struct {
  const uint8_t* key_id;
  size_t key_id_length;
  const uint8_t* key_data_iv;
  const uint8_t* key_data;
  size_t key_data_length;
  const uint8_t* key_control_iv;
  const uint8_t* key_control;
} OEMCrypto_KeyObject;
```

If license_type is OEMCrypto_ContentLicense, then OEMCrypto_LoadKeys shall perform as in the v13 API. It performs the usual verification and loads the keys into the session's array of content keys. The key_id in the OEMCrypto_KeyObject is the content key id, and the key_data is the content key data.



OEMCrypto_LoadKeys() with
license_type = OEMCrypto_ContentLicense

If license_type is OEMCrypto_EntitlementLicense, then OEMCrypto_LoadKeys shall load an array of entitlement keys. OEMCrypto_LoadKeys with a license_type of OEMCrypto_EntitlementLicense behaves the same as if license_type is OEMCrypto_ContentLicense, above, except the keys being loaded are entitlement keys instead of content keys. The key_id in the OEMCrypto_KeyObject is the entitlement key id, and the key_data is the entitlement key data. The entitlement key is an AES 256 bit key, and will be used to decrypt the content key. Verification is performed the same as it was in v13 based on each entitlement key's key control block. As before, nonce verification is performed if needed. The key control block is copied from this entitlement key, and is used for all output and decrypt restrictions with the associated content key -- after the content key is loaded. The duration is also copied as part of the key control block from this entitlement key to the session's key table.



OEMCrypto_LoadKeys() with
license_type = OEMCrypto_EntitlementLicense

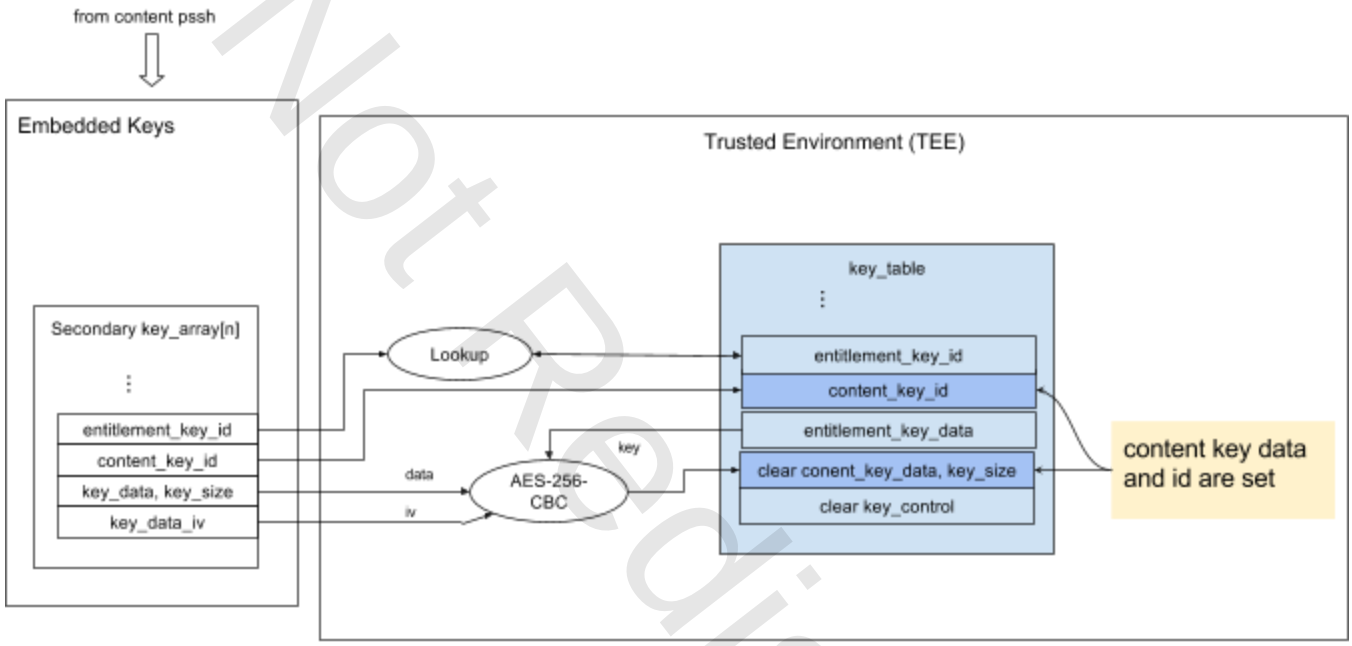
For an entitlement license, decryption cannot happen before the content keys are loaded, which is done by a call to OEMCrypto_LoadEntitledContentKeys. This function loops through each OEMCrypto_EntitledContentKeyObject in the key_array and performs:

1. The entry in the session's key table is found with the matching entitlement_key_id.
    a. If no matching entitlement key is found, the error OEMCrypto_KEY_NOT_LOADED is returned.
2. The key_data from the OEMCrypto_EntitledContentKeyObject is decrypted using the entitlement key data using AES 256.
3. The resulting key data is copied to the session's key table entry as the content key data.
4. The content key id is copied from the OEMCrypto_EntitledContentKeyObject to the session's key table entry.

OEMCrypto_LoadEntitledContentKeys will not wipe out the key table before executing. It is possible that the application will use several calls to OEMCrypto_LoadEntitledContentKeys to load a full collection of content keys. It is also possible that OEMCrypto_LoadEntitledContentKeys will be called to load new content keys into a key table entry that already has existing content keys. The old keys will be erased and replaced with the new keys. This overwrite feature will be used to perform key rotation.

```
OEMCryptoResult OEMCrypto_LoadEntitledContentKeys(
    OEMCrypto_SESSION session,
    size_t num_keys,
```

```
    const OEMCrypto_EntitledContentKeyObject* key_array);

typedef struct {
  const uint8_t* entitlement_key_id;
  size_t entitlement_key_id_length;
  const uint8_t* content_key_id;
  size_t content_key_id_length;
  const uint8_t* content_key_data_iv;
  const uint8_t* content_key_data;
  size_t content_key_data_length;
} OEMCrypto_EntitledContentKeyObject;
```



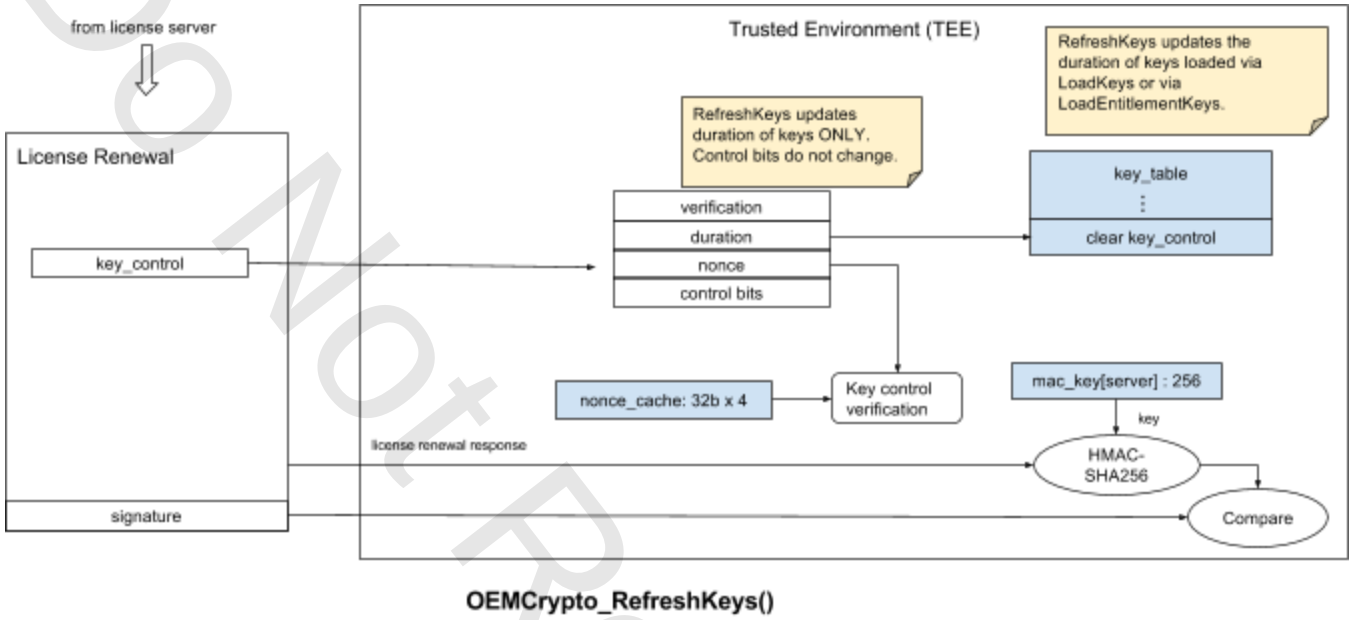
OEMCrypto_LoadEntitledContentKeys()

OEMCrypto_RefreshKeys has the same call parameters as in v13. Nonce verification is performed as in v13. The duration is updated as in v13. The difference is that if LoadKeys was called with a license type of OEMCrypto_EntitlementLicense, then the key_id specified in the KeyRefreshObject will match the entitlement_key_id of the entry in the key table. If the license type was OEMCrypto_ContentLicense, then the content_key_id will be used.

```
OEMCryptoResult OEMCrypto_RefreshKeys(OEMCrypto_SESSION session,
                      const uint8_t* message,
                      size_t message_length,
                      const uint8_t* signature,
                      size_t signature_length,
                      size_t num_keys,
                      const OEMCrypto_KeyRefreshObject* key_array);

typedef struct {
    const uint8_t* key_id;<-- this is the entitlement key id when called for an entitlement license.
    size_t key_id_length;
```

```
    const uint8_t* key_control_iv;
    const uint8_t* key_control;
} OEMCrypto_KeyRefreshObject
```

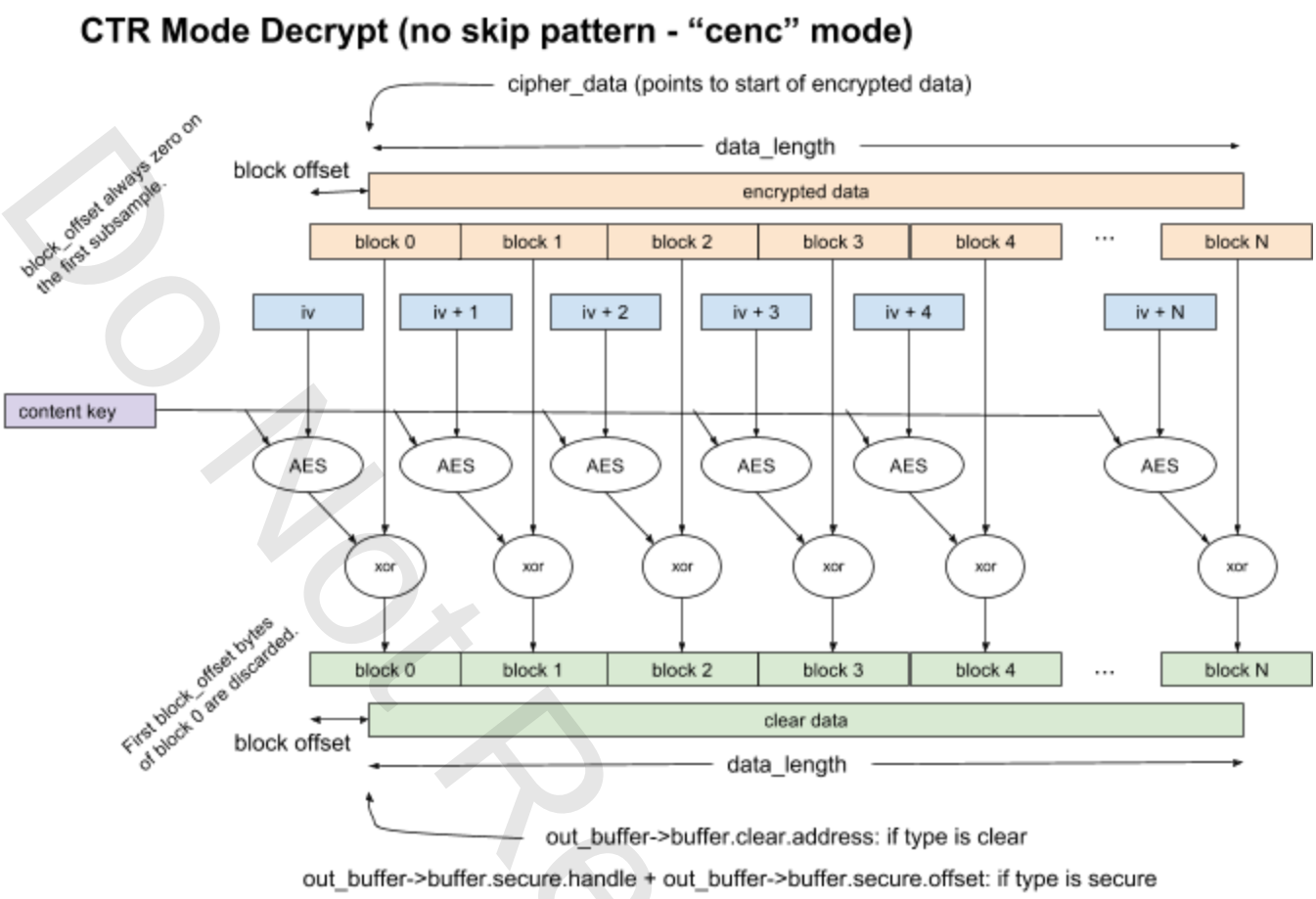
The function OEMCrypto_SelectKey will always use the content_key_id to select the current key for decryption.



OEMCrypto_RefreshKeys()

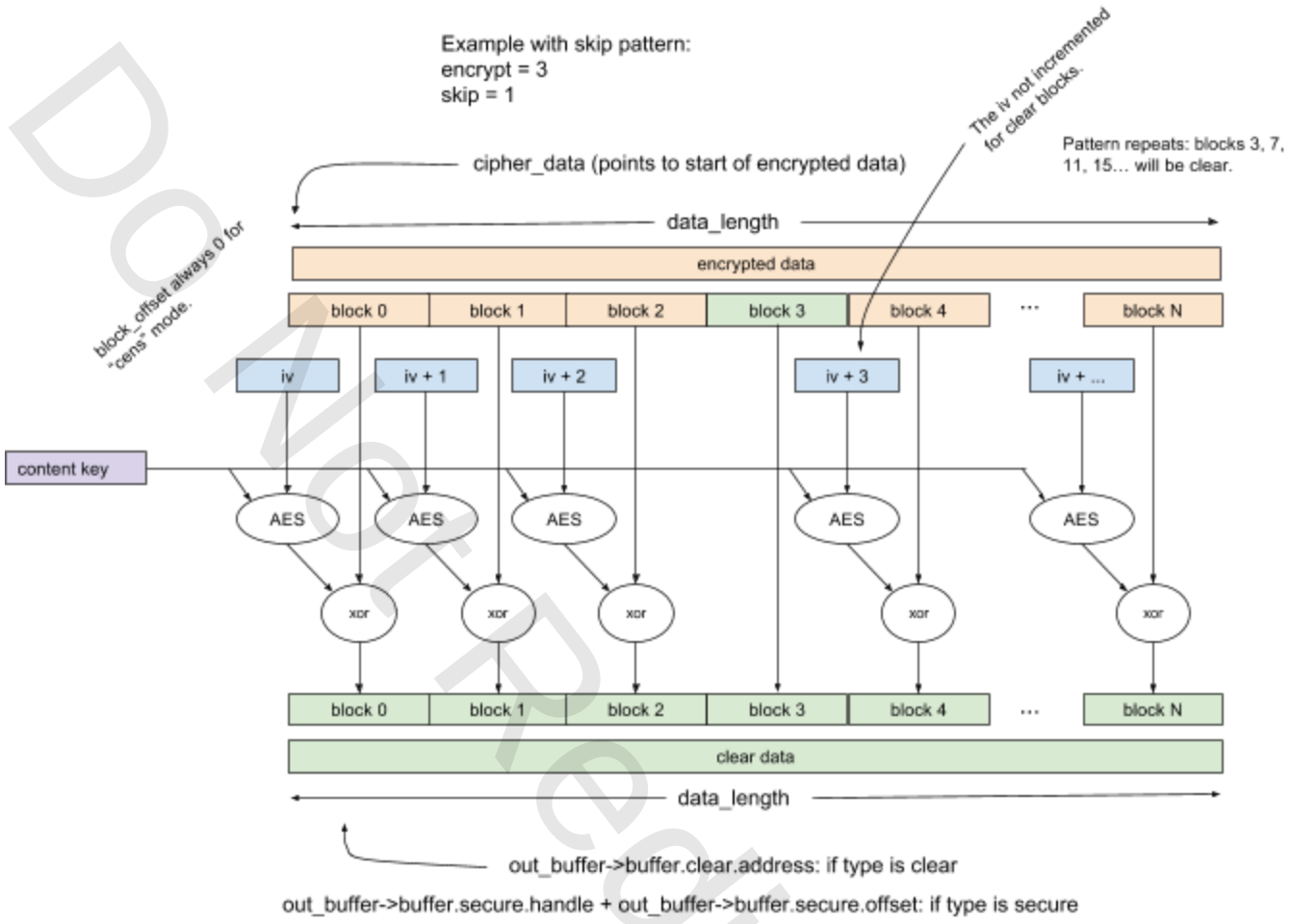
Block Offset Clarification

There is some confusion about the various parameters to DecryptCENC. In particular, there is a pattern offset, a block offset, and a secure buffer offset. These diagrams may help.



If OEMCrypto assembles all of the subsamples into a single buffer and then decrypts, it can assume that the block offset is 0.

CTR Mode Decrypt (with skip pattern - “cens” mode)



CBC Mode Decrypt (with skip pattern - “cbcs” mode)