FairPlay Streaming Programming Guide

Developer

Contents

Figures

About FairPlay Streaming

Apple FairPlay Streaming (FPS) securely delivers keys to Apple mobile devices, Apple TV, and Safari on macOS, which will enable playback of encrypted video content. This content is delivered over the Web using HTTP Live Streaming (HLS) technology.

FPS protects the delivery of keys that decrypt streamed audio and video media. An Apple device or computer can securely acquire a key from a content provider's key server. The operating system uses the key to decrypt the media before playback.

FPS key delivery offers the following features:

- AES 128-bit content keys are generated by the key server.
- Every key is known only to the key server and to the Apple device.
- When playback is stopped, the key for the iOS device, Apple TV, or Safari on macOS is permanently discarded from memory.
- The key server can specify the duration of the key's validity for iOS and Apple TV.
- MPEG-2 file formats are supported for protection.

FPS allows the device to stop playback based on expiration information sent with the content key. Using FPS on an Apple device, the key is transmitted securely and used securely for media decryption.

At a Glance

As an approved Apple FPS developer, you implement FPS by writing code to run on your key server and in your playback app, so that both recognize FPS messages. When an FPS-specific tag is included in the playlist of a media stream that the Apple device is asked to play, the operating system asks the app to obtain the decryption key. To do so, the app calls an API that invokes FPS, causing the operating system to prepare an encrypted request for the key for that media. When the app sends the request to the server, the FPS code on the server wraps the required key in an encrypted message and sends it to the app. The app then asks the operating system to unwrap the message and decrypt the stream, so the iOS device, Apple TV, or Safari on macOS can play the media.

The implementation process requires three programming tasks:

- Writing a *Key Security Module* that is installed in a key server's software. This module trades messages with the iOS device, Apple TV, or Safari on macOS during the FPS process.
- Adding code to make an Apple device playback app *FPS-aware*. The app communicates with a server that can deliver the key to decrypt the content, such as a movie. The FPS system, including an FPS-aware playback app, is diagrammed in Figure 1-1.

- Creating the formatting and encryption software for the media content server. This software prepares the encrypted content stream according to the Apple HTTP Live Streaming (HLS) specification.

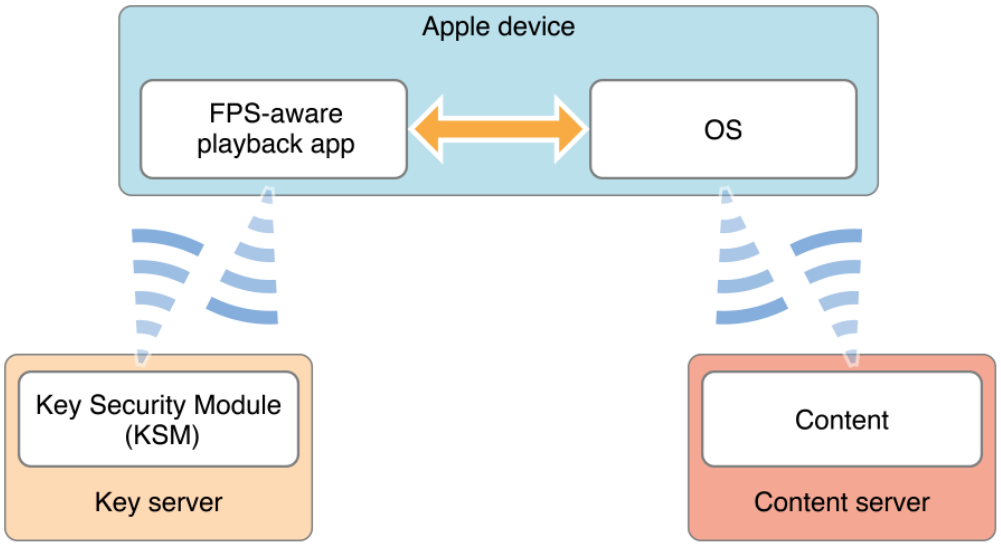


Figure 1-1 FPS exchanges

A Key Security Module Wraps a Key for Delivery

The Key Security Module (KSM) implements FPS algorithms that can interpret an encrypted key request message from an iOS device, Apple TV, or Safari on macOS and create an encrypted response containing the key for the specified media. You must write code for the key server enabling those algorithms.

An FPS-Aware Playback App Asks the Key Security Module for a Key

A FPS-aware playback app uses an operating system programming interface to obtain the encrypted request for the key, send it to the KSM, and receive the key for the media asset decryption. Your app will also need code supporting user interactions.

FPS Sends Content Keys Securely

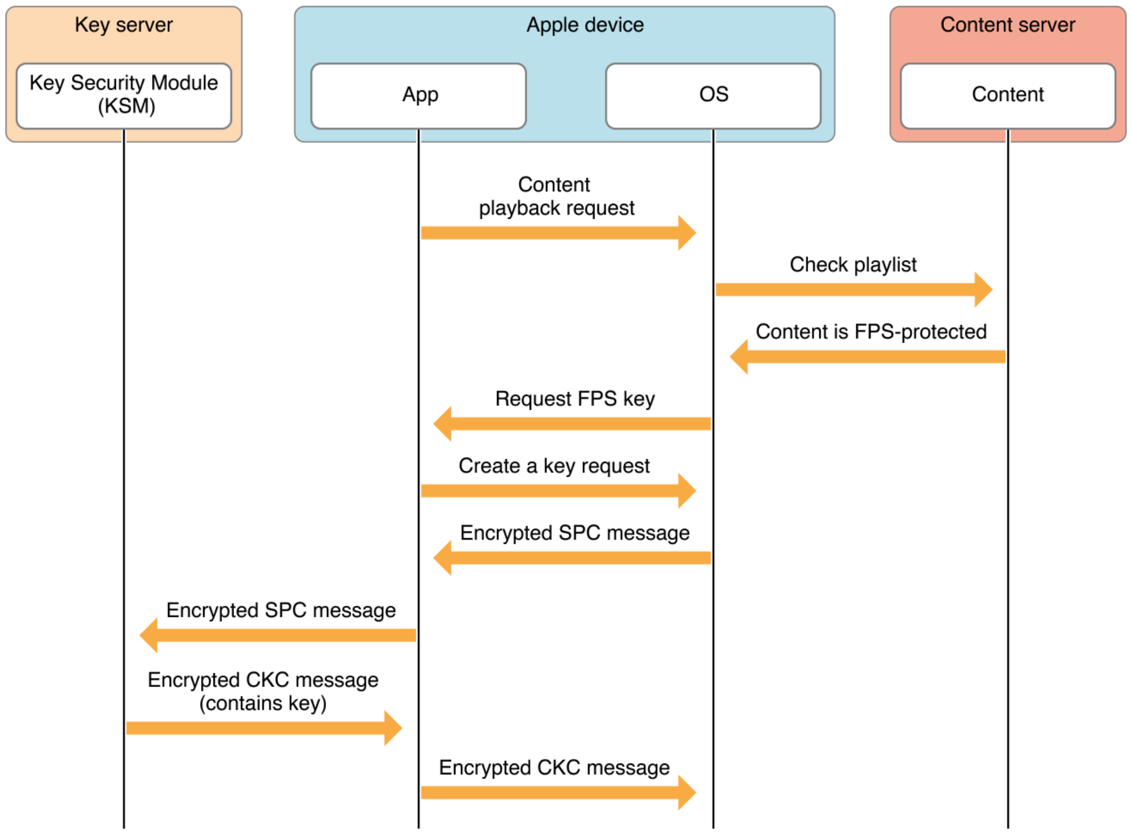
An FPS messaging session delivers a content key to the player app. Typically, the session proceeds as follows:

1. The app asks the operating system to play specific content identified by a URL.
2. The operating system accesses the content and checks its playlist.
3. An attribute in the playlist identifies the content as encrypted by a content key obtainable through FPS.
4. The operating system informs the app that the content is encrypted using FPS.

5. The app asks the operating system to prepare an FPS message that requests the content key.
6. The operating system delivers an encrypted Server Playback Context (SPC) message to the app.
7. The app sends the SPC to a key server that contains a KSM.
8. The KSM decrypts the SPC and gets the requested content key from the key server.
9. The KSM wraps the content key inside an encrypted content key context (CKC) message, which it sends to the app.
10. The app delivers the CKC to FPS software integrated in the operating system. The CKC is used to decrypt the media content, as described below.

These steps move information between FPS modules as shown in Figure 1-2.

Figure 1-2 FPS information flow



After the CKC is received, the FPS software extracts the content key and provides the key to the OS. The OS uses the key to decrypt and play the content requested in step 1 of Figure 1-2.

The streaming media playlist contains a list of versions of FPS that the key server supports. The operating system discerns and lists the mutually recognized FPS versions in the operating system's encrypted key request. The key server then picks which version to use.

Throughout this process, the app and the server can communicate through any transport link chosen by the app developer. The content streams in conformance with the HLS protocol, using H.264 video and audio formats.

Content Server Delivers the Content Stream

The content server delivers the formatted and encrypted content stream to the iOS device, Apple TV, or Safari on macOS. Only the Apple device, using keys delivered by FPS, can decrypt the stream and recover its content.

FairPlay Streaming SDK Contents

FairPlay Streaming SDK from Apple is provided in two parts:

Part 1, the FairPlay Streaming Server SDK contains a reference implementation of the Key Security Module, client sample, a specification and a set of test vectors. The test vectors can help establish and test the Key Security Module.

Part 2, the FPS Deployment Package contains the D Function and specification along with instructions about how to generate the FairPlay Streaming Certificate, private key and Application Secret key (ASk).

Both parts are required to perform a FPS transaction round-trip between the server Key Security Module and a client. The FPS Deployment Package materials must be requested by the content owner and are needed to complete the Key Security Module for production deployment. The keys and certificate should be generated once you have a tested Key Security Module.

See Also

The following documents contain specifications and instructions that supplement the material in this programming guide:

- See **HTTP Live Streaming Overview** for general guidance on Apple streaming technology used with FairPlay Streaming.
- See **MPEG-2 Stream Encryption Format for HTTP Live Streaming** for information on the FairPlay Streaming media formats.
- See **HTTP Live Streaming Protocol** for the IETF Internet-Draft of the HLS specification.
- See the following industry standards, relevant to FPS.
  - **Information technology—generic coding of moving pictures and associated audio information: Systems** is the ITU-T Recommendation H.222.0 document, also published as **ISO/IEC International Standard 13818-1:2013**.

- **Advanced video coding for generic audiovisual services** is the ITU-T Recommendation H.264 document, also published as **ISO/IEC International Standard 14496-10:2014**.
- **Information technology—Coding of audio-visual objects—Part 3: Audio** is the **ISO/IEC International Standard 14496-3:2009**.
- **Digital Audio Compression Standard (AC-3)** is the Advanced Television Systems Committee (ATSC) standard A/52:2012.

Programming the Key Security Module

The Key Security Module (KSM) is the part of FairPlay Streaming (FPS) technology that resides in the software of a content provider's key server. Its code must run on the server platform and implement the algorithms described in this chapter.

The KSM serves as a liaison between the playback app and the device. The initial message from app to device contains the server playback context (SPC). The device's operating system parses the SPC, generates the content key context (CKC). The KSM encrypts and delivers the content key to the device. The Apple device uses the content key to decrypt the FPS media sent from the content server. Requesting a Content Key from the Key Server describes this SPC exchange.

Overview of Processing Steps

Table 2-1 summarizes a typical sequence of actions that a server and its KSM might perform to support FPS.

Table 2-1 Typical server program steps

Step	Server action
1	Receive an SPC message from an app running on an Apple device and parse it. See *The SPC Message*.
2	Check the SPC's certificate hash value against the AC. See *Identifying Your FPS App with an Application Certificate* and Table 2-3.
3	Decrypt the SPC payload. See *SPC Payload Decryption*.
4	Verify that the Apple device is using a supported version of FPS software. See *Protocol Version Blocks*.
5	Decrypt the session key and random value block in the SPC payload. See *Decrypting the [SK...R1] Payload*.
6	Check the integrity of the SPC message. See *Session Key and Random Value Integrity Block*.
7	Encrypt the content key. See *Encrypting the Content Key*.
8	Assemble the contents of the CKC payload. See Table 2-11.
9	Encrypt the CKC payload. See *Encrypting the CKC Payload*.
10	Construct the CKC message and send it to the app on the Apple device. See Table 2-10.

Cryptographic Formula Syntax

The following conventions are used in this chapter to formulate cryptographic processes:

- Square brackets denote an encrypted value. For example, `[Info]` means that the plaintext value `Info` is encrypted.
- `e(Info)`$_K$ denotes the encryption of the plain text value Info, using the key $_K$. The convention may also specify the encryption algorithm: `RSA e(Info)`$_K$ indicates the use of RSA encryption, whereas `AES_CBC`$_{IV}$ `e(Info)`$_K$ indicates the use of AES encryption in cipher block chaining (CBC) mode with an initialization vector (IV).
- Similarly, `d(Info)`$_K$ denotes the decryption of the encrypted value `Info` using the key $_K$, and `AES_ECB d(Info)`$_K$ indicates the use of AES decryption in electronic codebook (ECB) mode.

For an example of this syntax, see *SPC Payload Decryption*.

SPC and CKC Messages

The SPC message that the playback app sends to the key server, and the CKC message that the KSM generates in reply, have these common characteristics:

- Each message consists of a fixed-length header followed by a variable-length payload.
- In both cases the payload is encrypted; in the SPC, part of the header is encrypted as well.
- The payloads in both the SPC and the CKC are divided into structures called tag-length-length-value (TLLV) blocks. This data layout is described in *TLLV Block Structure*.
- TLLV blocks are tightly packed into the payload fields, but the blocks are located in random sequence.
- TLLV blocks are found in each payload by searching for their unique 8-byte tags, which begin each block. For each tag value, only one block with that tag is permitted in an SPC or CKC message.
- The contents of TLLV blocks and all other FPS data structures are in clear text AES format.
- All numeric fields in the SPC, CKC, and TLLVs are stored in network (big-endian) order.

TLLV Block Structure

All TLLV blocks are built using the basic structure shown in Figure 2-1. The fields in this structure are defined in Table 2-2.

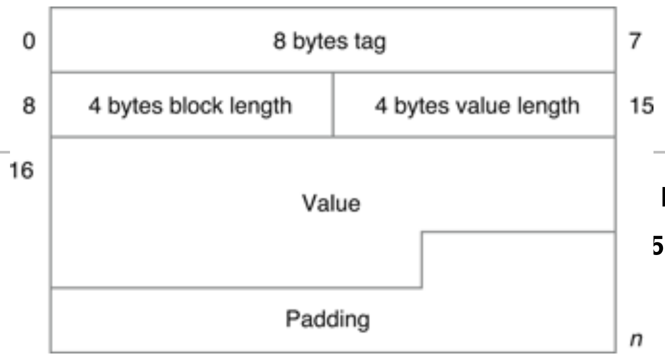


Figure 2-1 TLLV block structure

Table 2-2 TLLV block fields

Field content	Byte range	Description
Tag	0-7	A sequence of bytes that is unique within an SPC or CKC payload.
Block length	8-11	The number of bytes in the value plus padding fields of the TLLV (following the tag, block length, and value length fields). The block length must be filled out to a multiple of 16 bytes by extending the padding field.
Value length	12-15	The number of bytes in the value field. This number may be any amount, including `0x0000`.
Value	16 . . . *k*	The payload of the TLLV, starting with byte 16 of the block.
Padding	*k*+1 . . . *n* (*padding_size*)	A field that begins with the next byte after the value field (byte *k*+1). It must fill out the TLLV to a multiple of 16 bytes, but may be randomly extended in increments of 16 bytes. Thus the following relation holds: *padding_size = block_length - value_length* The padding field must contain random values, not all `0x00` or `0xFF` bytes.

Note: An SPC message may contain reserved TLLV blocks with tag values not covered in this documentation. Such blocks should be ignored by the KSM.

The SPC Message

The key server must be configured so that it delivers to its KSM every SPC message that the Apple device generates. Each SPC message is a container with a header of fixed-length fields and a variable-length data payload, as listed in Table 2-3. A sample SPC message is included in the FPS development support package; see *Using the FPS SDK and Tools.*

Table 2-3 SPC container structure

Field content	Byte range	Description
SPC version	0-3	The version number of the SPC. The version number covered by this programming guide is `0x00000001`.
Reserved	4-7	Reserved for Apple; ignore these bytes.
SPC data initialization vector (IV)	8-23	A CBC initialization vector that has a unique value for each SPC message. See *SPC Payload Decryption*.
Encrypted AES-128 key	24-151	The key for decrypting the SPC payload. This key is itself encrypted, using RSA public key encryption with Optimal Asymmetric Encryption Padding (OAEP), as described in *SPC Payload Decryption*.
Certificate hash	152-171	The SHA-1 hash value of the encrypted Application Certificate, which identifies the private key of the developer that generated the SPC. See *Identifying Your FPS App with an Application Certificate*.
SPC payload length	172-175	The number of bytes in the encrypted SPC payload. Because the payload consists of blocks whose lengths are multiples of 16 bytes, this number is a multiple of 16.
SPC payload	176 . . . *n*	A variable-length set of TLLV blocks, as described in *TLLV Block Structure*. The whole payload is AES-128 encrypted using the encrypted key contained in bytes 24-151 of the SPC message. The minimum set of TLLV blocks that the KSM must extract from this payload is specified in *SPC Payload Contents*.

The SPC Payload

The SPC payload begins at byte 176 of the SPC message and runs for the byte length specified by the value of the SPC payload length field (bytes 172-175). The SPC payload must be decrypted as specified in SPC Payload Decryption. Read about the decrypted value in *SPC Payload Contents*.

SPC Payload Decryption

The payload of the SPC message, which begins at byte 176, must be decrypted using the AES-128 cryptography standard with a cipher block chaining (CBC) mode of operation. The first block of the chain is initialized with the CBC initialization vector (IV) contained in bytes 8-23 of the SPC message.

The key for the AES decryption of the SPC payload, called SPCK, is obtained by decrypting bytes 24-151 of the SPC message, using the RSA cryptography standard with Optimal Asymmetric Encryption Padding (OAEP). The key for the RSA decryption of the SPCK is the server's RSA private key. An example of such a key is included in the FPS developer support package; see Using the FPS SDK and Tools.

In cryptographic formula syntax, decrypting the payload of the SPC consists of the following two decryption processes:

`SPCK = RSA_OAEP d([SPCK])`$_{\texttt{Prv}}$ where

`[SPCK]` represents the value of SPC message bytes 24-151.
`Prv` represents the server's private key.

`SPC payload = AES_CBC`$_{\texttt{IV}}$ `d([SPC data])`$_{\texttt{SPCK}}$ where

`[SPC data]` represents the remaining SPC message bytes beginning at byte 176 (175 + the value of SPC message bytes 172-175).
`IV` represents the value of SPC message bytes 8-23.

SPC Payload Contents

The decrypted SPC payload contains two TLLV types:

- Defined TLLVs listed in Table 2-4
- Undefined TLLVs

Any TLLV must appear only once in the SPC payload. TLLVs cannot be repeated in the same SPC payload.

Table 2-4 TLLV blocks in the SPC payload

TLLV content	Tag value	Description
[SK...R1]	`0x3d1a10b8bffac2ec`	A combination of values that the KSM will use to encrypt the content key and the CKC payload. This block must be parsed and decrypted as described in *Session Key and Random Value Block*. The R1 value must be returned to FPS in the payload of the CKC message, as described in *CKC Payload*.
[SK...R1] integrity	`0xb349d4809e910687`	A 16-byte value used to check the integrity of the contents of the [SK...R1] block. See *Session Key and Random Value Integrity Block*.

TLLV content	Tag value	Description
Anti-replay (AR) seed	`0x89c90f12204106b2`	A 16-byte value used in the encryption of the CKC payload. See *Encrypting the CKC Payload.*
R2	`0x71b5595ac1521133`	A 21-byte value used in decrypting the payload of the [SK...R1] block. See *Decrypting the [SK...R1] Payload*.
Tag return request	`0x19f9d4e5ab7609cb`	A TLLV block that contains zero or more concatenated 8-byte values, each of which is the tag for a different TLLV block in the SPC. All of the TLLV blocks listed must be retrieved and returned as is in the CKC payload. See *Returning SPC Blocks in the CKC Payload*.
Asset ID	`0x1bf7f53f5d5d5a1f`	A content provider ID that tells the key server which content needs to be decrypted. This value may be generated by the playback app or created by the FPS implementer. Its length can range from 2 to 200 bytes, inclusive. The asset ID content must be padded to a multiple of 16 bytes, regardless of the original length.
Transaction ID	`0x47aa7ad3440577de`	An 8-byte value that identifies the current FPS transaction. The KSM does not need to process this information.
Protocol versions supported	`0x67b8fb79ecce1a13`	A concatenation of 4-byte values identifying the Apple device-supported versions of FPS. See *Protocol Version Blocks*.
Protocol version used	`0x5d81bcbcc7f61703`	A 4-byte value that identifies the version of FPS that the Apple device is using for this FPS transaction. See *Protocol Version Blocks*.
Streaming indicator	`0xabb0256a31843974`	A single 8-byte value. See Table 2-5.
Media playback state	`0xeb8efdf2b25ab3a0`	Media playback information for rental and lease. See *TLLV for Rental and Lease*.
Capabilities	`0x9c02af3253c07fb2`	Client capabilities. See *Capabilities*.

Table 2-5 Streaming indicator values

Value	Description
`0xabb0256a31843974`	Content will be sent by AirPlay to an Apple TV box.
`0x5f9c8132b59f2fde`	Content will be sent to an Apple digital AV adapter.
`Any other value`	Content playback will occur on the requesting device.

Session Key and Random Value Block

The SPC delivers these two values in a single TLLV block called the [SK...R1] block:

- The 16-byte session key, SK, which is used to encrypt the content key as described in *Encrypting the Content Key*.
- A 44-byte random number, R1, which is used in the encryption of the CKC payload as described in *Encrypting the CKC Payload*. These bytes are also returned to the Apple device in the CKC payload, as shown in Table 2-11.

The structure of the whole [SK...R1] block, including its tag and length fields, is listed in Table 2-6. The payload contained in this block must be decrypted as described in *Decrypting the [SK...R1] Payload*.

Table 2-6 [SK...R1] TLLV block

Field content	Byte range	Description
TLLV tag	0-7	An 8-byte value of `0x3d1a10b8bffac2ec`.
Total length	8-11	The total length of this TLLV block in bytes. The total length is determined by the amount of padding at the end of the block, if any; it must be a multiple of 16 and greater than 127.
Value length	12-15	The length of the content of this TLLV block in bytes, `0x00000070` (decimal 112). The content consists of the initialization vector and payload fields.
Initialization vector (IV)	16-31	A 16-byte CBC initialization vector used in decrypting the next 96 bytes; see *Decrypting the [SK...R1] Payload*.

Field content	Byte range	Description
Payload	32-127	The 96-byte payload of the block. This payload must be decrypted as described in *Decrypting the [SK...R1] Payload* to yield its contents.
Padding	128-*n*	Random values to fill out the TLLV to a multiple of 16 bytes. See the description of the `Padding` field in Table 2-2.

Decrypting the [SK...R1] Payload

To recover the SK and the R1 value, the KSM must decrypt the payload of the block listed in Table 2-6, using AES-128 decryption with cipher block chaining (AES_CBC). The KSM must use the initialization vector (IV) contained in bytes 16-31 of the [SK...R1] block (see Table 2-6) to initialize the first block of the AES_CBC chain.

The DASk is obtained by following the instructions in a separate document, *D Function Computation Guide* provides instructions on calculating the DASk. The computation of the DASk requires input of the R2 block contents and the application secret key (ASk). The DASk value differs for each SPC request.

In cryptographic formula syntax, decrypting the payload of the [SK...R1] block consists of the following process:

`DASk = D(R2, ASk)` where

`R2` represents the contents of the R2 block in the SPC payload.
`ASk` represents the playback app's secret key.
`D` represents the function described in *D Function Computation Guide*.

$\texttt{Payload = AES\_CBC}_{\texttt{IV}}\ \texttt{d([[SK...R1] payload])}_{\texttt{DASk}}$ where

`IV` represents the value of [SK...R1] block bytes 16-31.
`[[SK...R1] payload]` represents the value of [SK...R1] block bytes 32-127.
`SK` and `R1` are integrity numbers that represent fields in `payload`.

Table 2-7 Decrypted [SK...R1] payload

Field content	Byte range	Description
Session key (SK)	0-15	A 16-byte value used in the encryption of the content key. See *Encrypting the Content Key*.

Field content	Byte range	Description
HU	16-35	A 20-byte value that represents the anonymized unique ID of the playback device. However, if the value of the streaming indicator TLLV (Table 2-4) in the SPC payload is `0x5f9c8132b59f2fde`, then this value represents the ID of the Apple digital AV adapter and should not be used for device management (see Table X-Y).
R1	36-79	A 44-byte random number that is used in the encryption of the CKC payload and is also returned to the Apple device in the CKC payload. See *Encrypting the CKC Payload*.
Integrity bytes	80-95	16 bytes used to check the integrity of this SPC message, as explained in *Session Key and Random Value Integrity Block*.

Session Key and Random Value Integrity Block

The [SK...R1] integrity block contains a 16-byte value that is used to check the integrity of the SPC. The KSM should compare the SPC contents with the 16-byte integrity number value in bytes 80-95 of the decrypted [SK...R1] payload; see Table 2-7. If the two numbers are not identical, the SPC is not valid and the KSM should reject it.

Protocol Version Blocks

The integrated FPS code in each iOS device uses two TLLV blocks in the SPC (listed in Table 2-4) to tell the KSM about the device versioning.

- The protocol versions supported block lists all the versions of FPS that the Apple device supports.
- The protocol version used block identifies the one version that the Apple device is using for the current transaction.

The purpose of sending versioning information in the SPC is to ensure that the key server and the Apple device are using the same version of FPS, and that it is the latest version that they both support.

The streaming content's playlist contains a list of the FPS versions that the key server supports for that content. As a good practice, your KSM should compare this list with the list of FPS versions in the protocol versions supported block. The protocol version used block should contain the ID of the most recent version common to both platforms. If the Apple device is not using the most recent common version, the app may be trying to attack FPS security. If there is no common version, the Apple device should not have generated an SPC and the KSM should reject the transaction.

Table 2-8 displays some recommended version configurations.

Note: Versioning should be decided between the server and the Apple device. The playback app should never contain embedded version information.

Table 2-8 Version configurations that follow good practices

Server	Apple device	SPC information

Version 1	Version 1	Used = 1
		Supported = 1

Versions 1 and 2	Version 1	Used = 1
		Supported = 1

Versions 1 and 2	Versions 1 and 2	Used = 2
		Supported = 1 and 2

Version 2	Versions 1 and 2	Used = 2
		Supported = 1 and 2

Table 2-9 displays other configurations that do not follow good practices and why these configurations should be avoided.

Table 2-9 Version configurations that do NOT follow good practices

Server	Apple device	SPC information	Implications

Version 1	Versions 1 and 2	Used = 1
		Supported = 1 and 2
			The server verifies the version used and uses version 1. However, a newer version of FPS is available, so the server should be updated.

Version 2	Version 1	Used = 1
		Supported = 1
			A mismatch exists between the FPS version on the Apple device and on the server; reject the SPC.

Versions 1 and 2	Versions 1 and 2	Used = 1
		Supported = 1 and 2
			An app may have tried to exploit the server by forcing it to use an old version of FPS.

Constructing the CKC Message

The KSM must respond to every SPC message by returning a corresponding CKC message to the Apple device that sent it. Each CKC message is a container with a header of fixed-length fields and a variable-length data payload, as listed in Table 2-10. The *FPS Developer Support Package* packaged with the SDK contains a sample CKC message.

Table 2-10 CKC container structure

Field content	Byte range	Description
CKC version	0-3	The version number of the SPC. The version number covered by this programming guide is `0x00000001`.
Reserved	4-7	Reserved by Apple; ignore these bytes.
CKC data initialization vector	8-23	A random 16-byte initialization vector, generated by the KSM, that has a unique value for each CKC message. The vector is used to initialize the first block of the AES_CBC chain, as described in *Encrypting the CKC Payload*.
CKC payload length	24-27	The number of bytes in the encrypted CKC payload. Because the payload consists of blocks whose lengths are multiples of 16 bytes, this number is a multiple of 16.
CKC payload	28 . . . *n*	A variable-length set of contiguous TLLV blocks, as described in *CKC Payload*. The CKC payload is AES-128 encrypted as described in *Encrypting the CKC Payload*.

CKC Payload

The KSM uses the session key (SK) to encrypt the content key. The payload of a CKC message contains the content key that the Apple device uses to decrypt the media for playback.

Table 2-11 lists the TLLV blocks in the CKC payload. The order of these blocks should be random.

Table 2-11 Contents of the CKC payload

TLLV content	Tag value	Description
Encrypted CK	`0x58b38165af0e3d5a`	Mandatory. A TLLV block containing a content initialization vector and a 16-byte encryption of the content key provided by the server. See *Encrypting the Content Key.*

TLLV content	Tag value	Description
R1	`0xea74c4645d5efee9`	Mandatory. A TLLV block containing the 44-byte R1 value that was sent to the KSM in the SPC payload. See *Session Key and Random Value Block*.
Content key duration	`0x47acf6a418cd091a`	A TLLV that specifies the period of validity of the content key. This TLLV may be present only if the KSM has received an SPC with a Media Playback State TLLV. See *Establishing the Rental and Lease Period*.
Blocks specified by a tag return request		The CKC must return, unchanged, the TLLV blocks that the SPC requested in a tag return request. See *Returning SPC Blocks in the CKC Payload.*
HDCP enforcement	`0x2e52f1530d8ddb4a`	An optional TLLV that specifies whether HDCP enforcement is required. The absence the TLLV enforces HDCP Type 0. See Table 2-12.

Because all the blocks listed above are padded to multiples of 16 bytes, the CKC payload as a whole does not require further padding.

HDCP Enforcement

Table 2-12 HDCP level values

Value	Description
`0xEF72894CA7895B78`	HDCP not required.
`0x40791AC78BD5C571`	HDCP Type 0 is required.
`0x285A0863BBA8E1D3`	HDCP Type 1 is required.

Encrypting the Content Key

The content provider creates the content key, later used to decrypt the media on the Apple device. The content key must be encrypted before it is put into the CKC payload, using AES-128 encryption. The session key that FPS sent to the KSM in the SPC payload serves as the encryption key.

In crypto-formula syntax, encrypting the content key consists of the following process:

> `[CK]` = `AES_ECB e(CK)`$_{SK}$ where
> `CK` is the content key provided by the key server.
> `SK` is the content of the session key block from the SPC payload.

The encrypted content key must be 16 bytes long. It becomes the content of the content key TLLV, shown in Table 2-13, which is encrypted and made part of the CKC payload (see Table 2-11). The CKC payload is further encrypted as described in *Encrypting the CKC Payload*.

Table 2-13 Content Key TLLV

Field name	Byte range	Description
TLLV tag	0-7	An 8-byte value of `0x58b38165af0e3d5a`.
Total length	8-11	The total length of this TLLV block in bytes. The total length is determined by the amount of padding at the end of the block, if any; it must be a multiple of 16 and greater than 31.
Value length	12-15	The length of the content of this TLLV block in bytes, `0x00000020` (decimal 32).
Initialization vector (IV)	16-31	A 16-byte CBC initialization vector used in AES encryption and decryption of audio and video assets.
Content key (CK)	32-47	The 16-byte content key encrypted using the SK.
Padding	48-*n*	Random values that fill out the TLLV to a multiple of 16 bytes. See the description of the `Padding` field in Table 2-2.

Returning SPC Blocks in the CKC Payload

The SPC payload contains a tag return request. This TLLV contains a list of the tags of other TLLVs. Each requested block must be returned verbatim, with its original tag and contents. The KSM must return those TLLVs at the end of the payload of the CKC.

Encrypting the CKC Payload

The CKC blocks, that the KSM encrypted, form the CKC payload. The blocks include the following, in random order:

- An encrypted content key TLLV block containing the 16-byte encrypted content key, as described in *Encrypting the Content Key*.
- The R1 TLLV block from the SPC payload; see *Session Key and Random Value Block*.
- All TLLV blocks from the SPC payload that are to be returned in the CKC payload, as described in *Returning SPC Blocks in the CKC Payload*.

To encrypt the CKC payload, an `AR_key` value is computed by taking the first 16 bytes of an SHA-1 digest of the R1 value that was sent in the payload of the SPC; see *Session Key and Random Value Integrity Block*. Then that `AR_key` value is used as a key to encrypt the AR seed obtained from the SPC payload (see Table 2-4).

The resulting encrypted AR seed is used as the key to encrypt the CKC data section using AES-128 with cipher block chaining (CBC). The KSM generates the CKC data initialization vector, sent to the Apple device in bytes 8-23 of the CKC message, shown in Table 2-10.

In cryptographic formula syntax, encrypting the payload of the CKC consists of the following process.

> `AR_key = first 16 bytes of SHA-1(R1)` where
>
> > `R1` represents the content of R1 block from the SPC payload.
>
> `[AR] = AES_ECB e(AR Seed)`$_{AR\_key}$ where
>
> > `AR Seed` represents the content of AR seed block from the SPC payload.
>
> `[CKC data] = AES_CBC`$_{IV}$ `e([CK] block, R1 block, Requested SPC blocks)`$_{[AR]}$ where
>
> > `IV` represents the random initialization vector generated by the KSM.
> > `[CK] block` represents the TLLV block containing the encrypted content key.
> > `R1 block` represents the R1 block from the SPC payload.
> > `Requested SPC blocks` represents the TLLV blocks listed in an SPC tag return request.
> > `[CKC data]` represents the CKC payload.

Capabilities

The Capabilities TLLV was introduced in iOS 11 and communicates features supported by the client to KSM. The Capabilities TLLV is sent in the SPC to the KSM.

Table 2-14 Capabilities TLLV

Field name	Byte range	Description
TLLV tag	0-7	An 8-byte value of `0x9c02af3253c07fb2`.
Total length	8-11	The total length of this TLLV block in bytes. The total length is determined by the amount of padding at the end of the block, if any; it must be a multiple of 16 and greater than 31.
Value length	12-15	The length of the content of this TLLV block in bytes, `0x00000010` (decimal 16).
Capabilities bits (high)	16-23	An 8-byte field containing capability bits 64-127. See Table 2-15.
Capabilities bits (low)	24-31	An 8-byte field containing capability bits 0-63. See Table 2-15.
Padding	32-*n* (*padding_size*)	Random values that fill out the TLLV to a multiple of 16 bytes. See the description of the `Padding` field in Table 2-2.

The two capabilities bits fields are effectively a 128-bit long value which is split into two 64-bit values for ease of processing. Each capability bit indicates whether a specific feature is supported by the client.

Table 2-15 Features

Feature	Capability bit	Description
HDCP Enforcement	0	When set means that the client can enforce the HDCP restrictions given in the HDCP Enforcement TLLV. See Table 2-12.
Offline Key	1	When set means that the client is capable of handling and enforcing the Offline key TLLV. See the document *Offline Rental Support for FairPlay Streaming*.

Note - iOS 10 supports HDCP Enforcement even though it does not provide a Capabilities TLLV.

Testing the Key Security Module

The FPS SDK includes a CKC verification tool, `verify_ckc`, with SPC and CKC test vectors. Use the `verify_ckc` tool and test vectors to verify that the KSM implementation can produce a valid CKC. See *Using the FPS SDK and Tools*.

Once you have a working KSM you can request the FPS Deployment Package at **https://developer.apple.com/contact/fps/** which contains the D Function, instructions about how to generate your Application Certificate and Application Secret key (ASk) values. These are needed to test your KSM implementation with an FPS client.

Developing an FPS-Aware App

Any media playback app that runs on an iOS device or Safari on macOS 10.10.3 or later can implement FPS. This chapter covers programming required for an iOS app to obtain content keys that decrypt FPS media.

Typically, playback apps provide a user interface for browsing and selecting the content to be streamed; support user identification; and facilitate other user and content provider transactions. Additionally, an FPS-aware playback app must establish two-way communication between the Apple device and a key server to support FPS functions.

⚠️ **Warning:** FPS cannot be run on iOS Simulator.

For general information about writing apps for Apple iOS devices, visit the **Developer Center**.

Identifying Your FPS App with an Application Certificate

As part of registering an FPS playback app, you provide Apple with an X.509 Certificate Signing Request linked to your private key. In return, you receive an Application Certificate encoded with the X.509 standard with distinguished encoding rules (DER). Bytes 152-171 of the SPC message contain a secure hash algorithm (SHA-1) digest of that encoded certificate.

Every playback app that uses FPS must find the media's key server and establish communication with that server. When messages can be exchanged between the iOS device and the key server, the app must send the server an FPS-created SPC message. This message contains a hash of the Application Certificate identifying your private key.

The recommendations below help you ensure FPS security.

- Do not hard-code the Application Certificate in the playback application.
- Verify that the hash value in bytes 152-171 of the SPC correctly identifies the private key of the developer from which the module expects to receive SPC messages.
- Do not enforce the expiration date of the Application Certificate within your app. FPS does not enforce the expiration date.

In the code sample shown in the iOS FPS Client sample (included in the SDK), `kTestAppCert` contains the Application Certificate.

Integrating FPS with the iOS Decryption Process

To use FPS for iOS apps, the playback app must implement the `AVAssetResourceLoaderDelegate` protocol. For each `AVURLAsset` subclass required by FPS, the app must create an appropriate object that implements this protocol as the `AVAssetResourceLoader` delegate for that subclass. `AVAssetResourceLoader` will invoke this delegate to examine URL requests that the operating system cannot handle by itself, including requests for content keys.

The app uses the `resourceLoader` property of the `AVURLAsset` subclass to obtain the instance of `AVAssetResourceLoader` associated with the class. It uses the `AVAssetResourceLoader` method `-setDelegate:queue:` to set the delegate and the dispatch queue on which `AVAssetResourceLoader` will invoke the delegate.

Integrating FPS in Safari on macOS

The FPS content that you author for iOS and Apple TV also plays in Safari on macOS 10.10.3 and later. Encrypted Media Extensions (EME) provide FPS support on Safari on macOS. The EME specification is available at **http://www.w3.org/TR/encrypted-media/**. The EME extended `HTMLMediaElement` APIs manage the FPS process with secure message exchanges analogous to FPS on iOS devices and Apple TV.

EME Message Exchange

For web pages in Safari on macOS, FPS supports a *Key System* identified by the string `com.apple.fps.1_0`.

A Key Session needs to be created to provide a context for message exchange with the Content Decryption Module (CDM)/Key System. Per the EME specification, each Key Session is associated with a single instance of *Initialization Data* provided in the `createSession()` call.

For FPS, this Initialization Data must be the following byte array.

```
AssetID + Certificate
```

In this equation, `AssetId` represents the byte array defined in Table 2-4, `Certificate` represents the Application Certificate provided by Apple, and the `+` indicates concatenation of the two values. The `AssetId` can be any string you choose.

Use the following events in your JavaScript for Safari to support FPS.

`webkitneedkey`
: The `webkitneedkey` event finds the CDM, identified with the string `com.apple.fps.1_0`, and allows for creation of the `keySession`. The event is triggered when a process requests playback of FPS protected content.

`webkitkeymessage`
: The `webkitkeymessage` event sends the SPC and obtains a CKC from the Key Server Module. The `update()` function adds the CKC to the `keySession`.

`webkitkeyadded`
: The `webkitkeyadded` event triggers when a key is successfully added to the `keySession`.

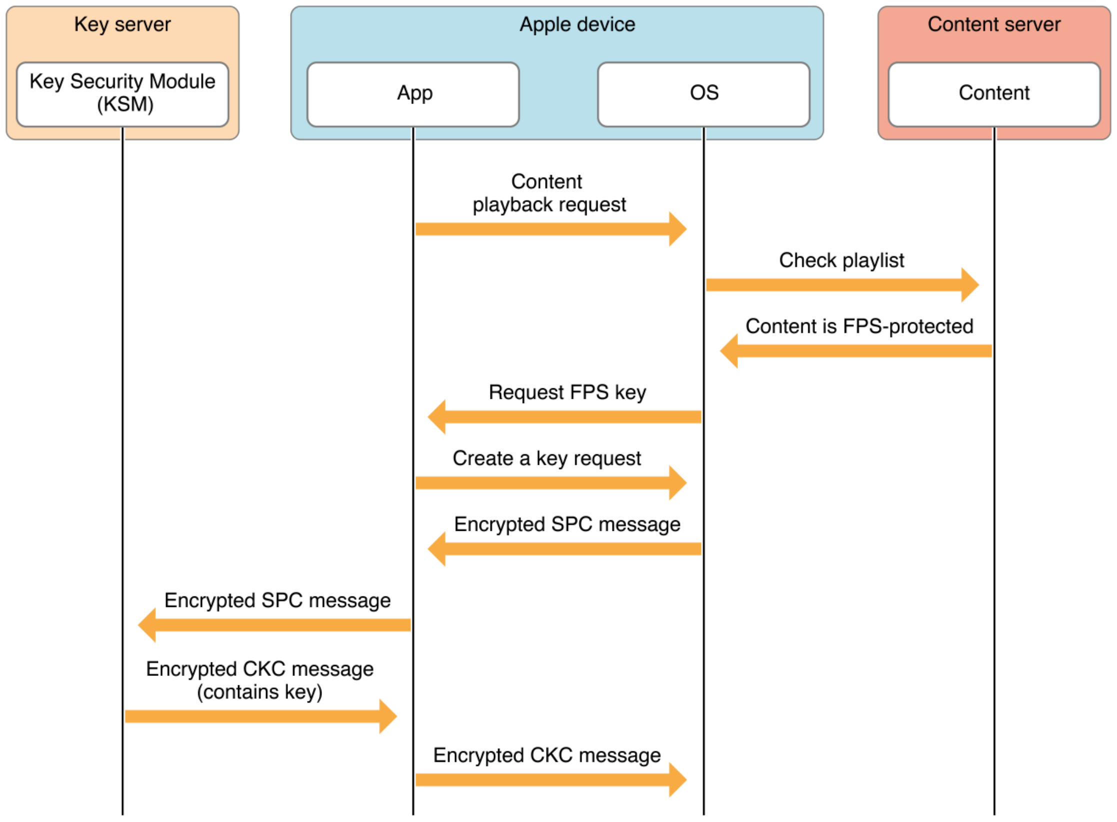
`webkitkeyerror`
: The `webkitkeyerror` event triggers if an error occurs while processing the SPC or CKC. The key is not added to the `keySession`.

The FPS SDK provided by Apple includes a sample JavaScript implementation of the API for Safari on macOS.

Requesting a Content Key from the Key Server

When the operating system asks the app to provide a content key, as shown in Figure 3-1, the app invokes the `AVAssetResourceLoader` delegate's implementation of its `–resourceLoader:shouldWaitForLoadingOfRequestedResource:` method. This method provides the delegate with an instance of `AVAssetResourceLoadingRequest`, which accesses the underlying `NSURLRequest` for the requested resource and support for responding to the request.

Figure 3-1 FPS information flow



When the request is for a content key, the app invokes the delegate `–[AVAssetResourceLoadingRequest streamingContentKeyRequestDataForApp:contentIdentifier:options:error:]` method. This method obtains the SPC message from the operating system. Then the app sends the SPC to the key server, as shown in Figure 3-1, using appropriate transport forms and protocols.

Processing the Key Server's Response

The CKC message, containing the content key, is constructed from the SPC by the KSM, as described in *Programming the Key Security Module*. The key server returns a CKC message in response to the app's SPC message, as shown in Figure 3-1. After receiving this message, the app sends it to the operating system by invoking the `AVAssetResourceLoadingRequest` method `–[AVAssetResourceLoadingRequest finishLoading]`. The device can now decrypt and play the content stream using HLS, summarized in **HTTP Live Streaming Overview**.

Configuring AirPlay Mode

When an iOS device is in AirPlay mode, FPS content will not play on an attached Apple TV unless AirPlay playback is set to mirroring. The FPS-aware app must set the `usesExternalPlaybackWhileExternalScreenIsActive` property of the AVPlayer object to `TRUE` with code such as this:

```
// create AVPlayer object
player = [AVPlayer playerWithURL:movieURL];
// set the property to TRUE
player.usesExternalPlaybackWhileExternalScreenIsActive = TRUE;
```

Interpreting Error Messages

When `–streamingContentKeyRequestDataForApp:contentIdentifier:error:` fails, it returns `nil` and sets the `outError` parameter to an instance of `NSError` that describes the failure. In this case, invoke `–finishLoadingWithError:`, passing the resulting error.

If you want the application to report the error to the user at this stage of the process, each instance of `NSError` provided by `–streamingContentKeyRequestDataForApp:contentIdentifier:error:` has a `localizedDescription` that is suitable. Another `NSError` instance is available through the `NSUnderlyingErrorKey` in the `userInfo` dictionary of the `NSError` instance provided by `–streamingContentKeyRequestDataForApp:contentIdentifier:error:`. This `NSError` instance provides additional information of interest when debugging an application that tries to obtain an SPC. It can contain one of the codes listed in Table 3-1.

Table 3-1 FPS error messages

Message	Description
-42656	Lease duration has expired.
-42668	The CKC passed in for processing is not valid.
-42672	A certificate is not supplied when creating SPC.
-42673	`assetId` is not supplied when creating an SPC.

Message	Description
-42674	Version list is not supplied when creating an SPC.
-42675	The `assetID` supplied to SPC creation is not valid.
-42676	An error occurred during SPC creation.
-42679	The certificate supplied for SPC creation is not valid.
-42681	The version list supplied to SPC creation is not valid.
-42783	The certificate supplied for SPC is not valid and is possibly revoked.

Manually Fetching FPS Error Messages

The following code fetches an underlying error specific to FPS.

Listing 3-1 Manually fetching FPS errors

```
NSError *topLevelError = nil;
NSData *requestData = [loadingRequest
streamingContentKeyRequestDataForApp:appID contentIdentifier:contentID
error:&topLevelError];
if (requestData == nil && topLevelError != nil)
{
     NSError *underlyingError = [[topLevelError userInfo] objectForKey:
NSUnderlyingErrorKey];
     if ([[underlyingError domain] isEqualToString:NSOSStatusErrorDomain])
     {
         NSInteger errorCode = [underlyingError code];
         // check whether this errorCode is specific to FPS as listed in
Table 3-1.
     }
}
```

Formatting and Encrypting Streams

You've seen how the KSM works and how to build an app to communicate with that KSM. Now that you understand some aspects of FPS on the key server and the device, take a look at the content that streams between the two. FPS requires content formatting and encrypting in accordance with the HTTP Live Streaming (HLS) and MPEG-2 stream encryption standards published in **HTTP Live Streaming IETF draft** and **MPEG-2 Stream Encryption Format for HTTP Live Streaming**. This chapter helps you prepare your FPS content for these standards.

Beyond this chapter, the following books provide instruction on formatting and encrypting streams.

- *HTTP Live Streaming IETF draft* details the media formatting required to send your content via HTTP. The advantages of HLS and implementation instructions are provided in **HTTP Live Streaming Overview**.
- *MPEG-2 Stream Encryption Format for HTTP Live Streaming* explains features of the MPEG-2 standard.

Preparing Content for FPS

As shown in **Example Playlist Files for use with HTTP Live Streaming**, HLS extends the `m3u` playlist format with an `EXT-X-KEY` tag. FPS requires that this tag be included in the HLS playlist and that it declare the following attributes:

- `METHOD`: The encryption method. `SAMPLE-AES` indicates AES-128_CBC unpadded encryption of individual samples.
- `URI`: The path for obtaining the content key. An example is `skd://key65`, as shown in Listing 4-1.
- `KEYFORMAT`: A value of `com.apple.streamingkeydelivery` indicates a FPS key; `identity` indicates the original key format of clear text 16-byte AES key.
- `KEYFORMATVERSIONS`: A list of key format versions separated by slashes. For example, 1/2 indicates support for either version 1 or version 2 of the key format.

The following listing shows a sample FPS `EXT-X-KEY` tag in a streaming playlist.

```
#EXT-X-KEY:METHOD=SAMPLE-AES,URI="skd://key65",
KEYFORMAT="com.apple.streamingkeydelivery",KEYFORMATVERSIONS="1"
```

Listing 4-1 Adding FPS to an HLS Playlist

Including Initialization Vectors (IV) in Playlists

There are a few important considerations that apply to the IV in an FPS `m3u8` playlist.

- FPS does not support IVs listed in the `EXT-X-KEY` tag's IV attribute in an `m3u8` playlist. Any IV in the playlist is ignored. The IV is only delivered by the Key Security Module in the content key context (CKC).
- FPS does not support using IVs as media sequence numbers in a `m3u8` playlist. The IV delivered in the CKC is used in AES encryption and decryption of audio and video assets.
    - An `EXT-X-KEY` tag with a `KEYFORMAT` of identity without an IV attribute indicates that the media sequence number should be used as the IV to decrypt a media segment. However, if you specify a `KEYFORMAT` of `com.apple.streamingkeydelivery` to indicate an FPS key (leaving out any IV), IVs are not used as media sequence numbers.
    - For more information, see **http://tools.ietf.org/html/draft-pantos-http-live-streaming**.
- The client doesn't make a key request for every segment in the playlist.
    - For example, if an `EXT-X-KEY` tag is specified in the playlist and three segments are listed, the client will request the key just once. This means the three segments must be encrypted with the same IV.
    - In a similar example, a playlist sequence includes the following items where the key lines are identical. In this scenario, the second key line (with `Key_0`) is not necessary because the key will only be requested once.
        - `Key_0`
        - `Segment_1`
        - `Key_0`
        - `Segment_2`
    - Some special circumstances do require key requests for every segment, such as when using FPS through AirPlay or Digital AV Adaptors.

Using FPS Options with the Media File Segmenter

The following new FPS-specific features were added to the `mediafilesegmenter` tool, included with the HTTP Live Streaming tools download.

- `-P` is the short form of `--streaming-key-delivery`. Either form indicates that the key file is 32 bytes long, where the first 16 bytes is the content key and the second 16 bytes is the initialization vector (IV). This option is required for `KEYFORMAT="com.apple.streamingkeydelivery"` streaming.
- The option `--encrypt-iv` is incompatible with FPS.

- If an existing key file is supplied to `--encrypt-key-file`, it must be 32 bytes long, where the first 16 bytes hold the content key and the second 16 bytes hold the IV. The segment `/tmp/key.bin` in Listing 4-2 represents a 32-byte file.
- The `--stream-encrypt` option is required if the key is to be delivered via FPS.

Listing 4-2 uses the `mediafilesegmenter` command to produce an `m3u8` playlist for use with FPS.

Listing 4-2 Media File Segmenter command

```
mediafilesegmenter --stream-encrypt --streaming-key-delivery --encrypt-key-
file=/tmp/key.bin --encrypt-key-url="skd://example/key" /tmp/source.mov
```

Downloading HTTP Live Streaming tools: You must log in to the iOS developer library to access the **HTTP Live Streaming tools download**.

Renting and Leasing the Content Key

Starting with iOS 9.0, FPS supports time-sensitive content keys. Renting and leasing set specific limits on an app's access to FPS decryption keys. The server may associate a rental period with the media content and/or a lease period with the device. The server's CKC response contains the validity duration of the content key.

Content Key Expiration

FPS's content key expiration creates two modes of time-sensitive exchange: video rental and secure lease. These modes can be used separately or together.

Video Rental

The content key is a rental type. FPS does not start the decryption if the content key has expired. However, FPS continues the user experience if the content key expires during the playback. When started again with an expired key, the client declines the playback.

Secure Lease

The content key is a lease type. Typically, a content provider policy would restrict the number of simultaneous playbacks (slots) for a user account. The server associates a slot to a device, and the server delivers the content key with the expiration that represents the lease. The client may request that the key be renewed by the server before the lease expires. The server provides a new expiration time for the content key, and playback continues uninterrupted. If the content key is not renewed, the client stops the playback when the lease expires. The server recognizes that playback has stopped and frees the device slot.

This design ensures that a device is not orphaned (in a stale state) based on time rather than messaging and garbage collection. The expiration triggers a server event to securely release the device slot. The server knows playback stopped and frees the device slot as soon as the content key expires and the PlayContent is discarded. Using the secure lease and the device identification, the server can implement a robust solution for the management of simultaneous streams maintaining a seamless user experience.

TLLV for Rental and Lease

The SPC includes a specific TLLV to provide the state of the media content playback. The key server uses this TLLV to manage the rental period and the lease period. Details of the media playback state TLLV are provided in Table 5-1.

Table 5-1 Media playback state TLLV

Field name	Byte range	Description
TLLV tag	0-7	An 8-byte value of `0xeb8efdf2b25ab3a0`.

Field name	Byte range	Description
Total Length	8-11	The total length of this TLLV block in bytes. The total length is determined by the amount of padding at the end of the block, if any; it must be a multiple of 16 and greater than 32.
Value Length	12-15	The length of the content of this TLLV block in bytes, `0x00000010` (decimal 16).
Creation Date	16-19	The time in seconds from Jan 1, 1970 to the time when the SPC was created.
Playback State	20-23	The playback state of the Apple device at the time the SPC was created. Possible values are listed in Table 5-2.
Session ID	24-31	An ID that represents the playback of a media content independently of its bit rates and content keys. When a movie is closed and re-opened, a new Session ID is generated to identify the new instance.
Padding	32-*n* (*padding_size*)	Random values to fill out the TLLV to a multiple of 16 bytes. See the description of the `Padding` field in Table 2-2.

Table 5-2 Apple device playback states

TLLV field value	Playback state
`0xf4dee5a2`	State 1: The Apple device is ready to start playing. The response CKC must contain a valid content key.
`0xa5d6739e`	State 2: The playback stream is playing or paused. The KSM must reply with a CKC containing a rent/lease response TLLV, but it does not need to contain a valid content key.
`0x4f834330`	State 3: The playback stream is playing, but the lease is about to expire. The response CKC must contain a valid content key.

Establishing the Rental and Lease Period

When a KSM receives an SPC with a media playback state TLLV, the KSM may include a content key duration TLLV in the CKC message that it returns. If the Apple device finds this type of TLLV in a CKC that delivers an FPS content key, it will honor the terms of the rental or lease or both when the key is used. Table 5-3 lists the fields of the rental and lease response TLLV.

Note: The app is unable to modify or overrule the rental and lease periods specified in the CKC.

Table 5-3 Content key duration TLLV

Field name	Byte range	Description
TLLV tag	0-7	An 8-byte value of `0x47acf6a418cd091a`.
Total Length	8-11	The total length of this TLLV block in bytes. The length is determined by the amount of padding at the end of the block, if any; this value must be a multiple of 16 and greater than 32.
Value Length	12-15	The length of the content of this TLLV block in bytes, `0x00000010` (decimal 16).
Lease Duration	16-19	The duration of the lease, if any, in seconds.
Rental Duration	20-23	The duration of the rental, if any, in seconds.
Key Type	24-27	The key type. Possible values are listed in Table 5-4.
Reserved	28-31	Reserved; set to a fixed value of `0x86d34a3a`.
Padding	32-*n* (*padding_size*)	Random values to fill out the TLLV to a multiple of 16 bytes. See the description of the `Padding` field in Table 2-2.

Table 5-4 Rental and lease key types

TLLV field value	Type of rental or lease
`0x1a4bde7e`	Content key valid for lease only.
`0x3dfe45a0`	Content key valid for rental only.
`0x27b59bde`	Content key valid for both lease and rental.

See *Offline FairPlay Streaming* for additional rental and lease key types to support persistent keys.

Simultaneous Renting and Leasing

A rental and a lease can be combined in one CKC. The combination of renting and leasing follows these general rules:

- A rental period covers the initial delivery of a content key and the start of a stream. If the rental period expires during playback, the stream will continue to flow until the media playback stops.

- A lease period covers the validity of the content key for media playback. If the lease period expires during playback, the media playback stops.
- In a combined renting and leasing arrangement, the mechanism by which leasing registers the playback device may be used to limit the rental to that device only, as a leasing restriction enforced by the KSM.
- If the lease expires before the end of the rental period, the key server should allow the lease to be renewed.

Offline FairPlay Streaming

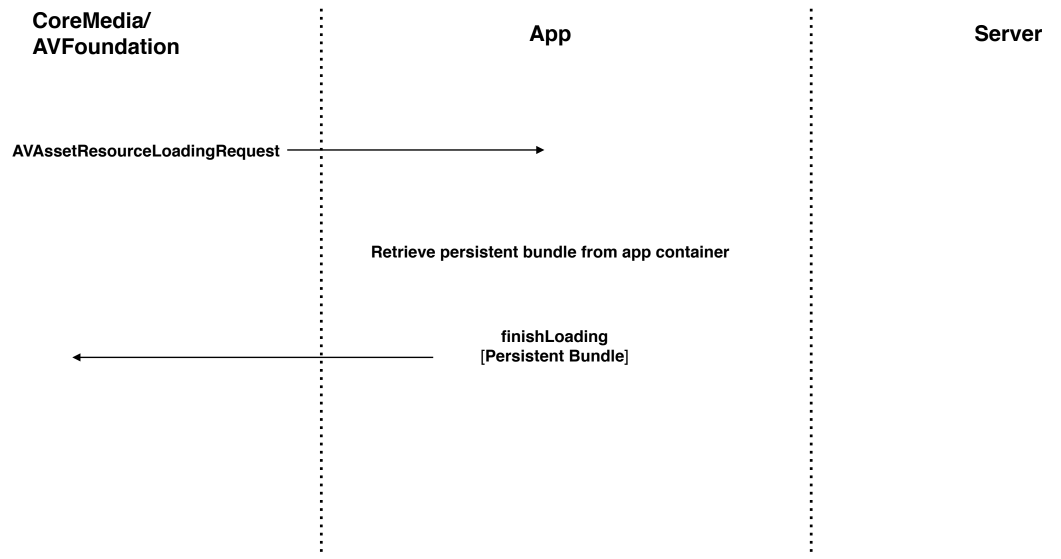
Note: Offline FairPlay Streaming (FPS) is an extension of offline HTTP Live Streaming (HLS). This guide describes how to build on top of the technologies to achieve an Offline FairPlay Streaming solution. The term "HLS" is used to describe technology that is common to both FPS and HLS.

Starting with iOS 10.0, apps can save HTTP Live Streaming assets onto iOS devices. This is known as Offline HLS. This new capability allows users to download and store their HLS movies while they have access to a fast, reliable network, and watch them later without a network connection.

Offline HLS assets can be SAMPLE-AES encrypted. There are a few additional steps required for downloading and managing encrypted Offline HLS assets to ensure the playability of downloaded assets when no network connection is present.

When creating an `AVURLAsset` for use in downloading an Offline HLS asset, apps must install a delegate to handle encryption keys. The persistent keys are not stored with downloaded HLS assets. Instead, apps must store and manage persistent keys, using an `AVAssetResourceLoader` and a delegate object implementing the `AVAssetResourceLoaderDelegate` protocol. Starting with iOS 10, FPS supports content key persistence. The server may enable an iOS device to persist the key either indefinitely or for the provided validity duration. To enable persistence of the content key, the server's CKC response shall contain a Content key duration TLLV (TLLV tag `0x47acf6a418cd091a`). See Table 5-3.

Offline Playback with Persistent Bundle



CoreMedia/ AVFoundation
App
Server
AVAssetResourceLoadingRequest
Retrieve persistent bundle from app container
finishLoading [Persistent Bundle]

Figure 6-1 Offline playback

Use of SESSION-KEY

Playback of Offline HLS assets shall use `EXT-X-SESSION-KEY` to declare all eligible content keys in the master playlist. In addition, the client shall opt into `AVAssetResourceLoader` property `preloadsEligibleContentKeys`. This will enable `AVURLAsset` to preload all the content keys. The client must ensure that all the keys have been loaded before starting a background download task on the `AVURLAsset`.

The final policy decision as to which content keys can be persisted on the device belongs to the Key Security Module vending the CKC. The server has the option to allow the client to persist the key either indefinitely, or for a specified duration. To enable persistence of the content key, the server's CKC response shall contain a Content key duration TLLV.

FairPlay Streaming on the client does not start the decryption if the persisted content key has expired. However, FairPlay Streaming on the client will continue the user experience if the content key expires during playback.

Table 6-1 Rental and lease key types for persistence

TLLV field value	Type of rental or lease
`0x3df2d9fb`	Content key can be persisted with unlimited validity duration.
`0x18f06048`	Content key can be persisted, and its validity duration is limited to the "Rental Duration" value.

The diagram below shows the life cycle of a persistent key request.

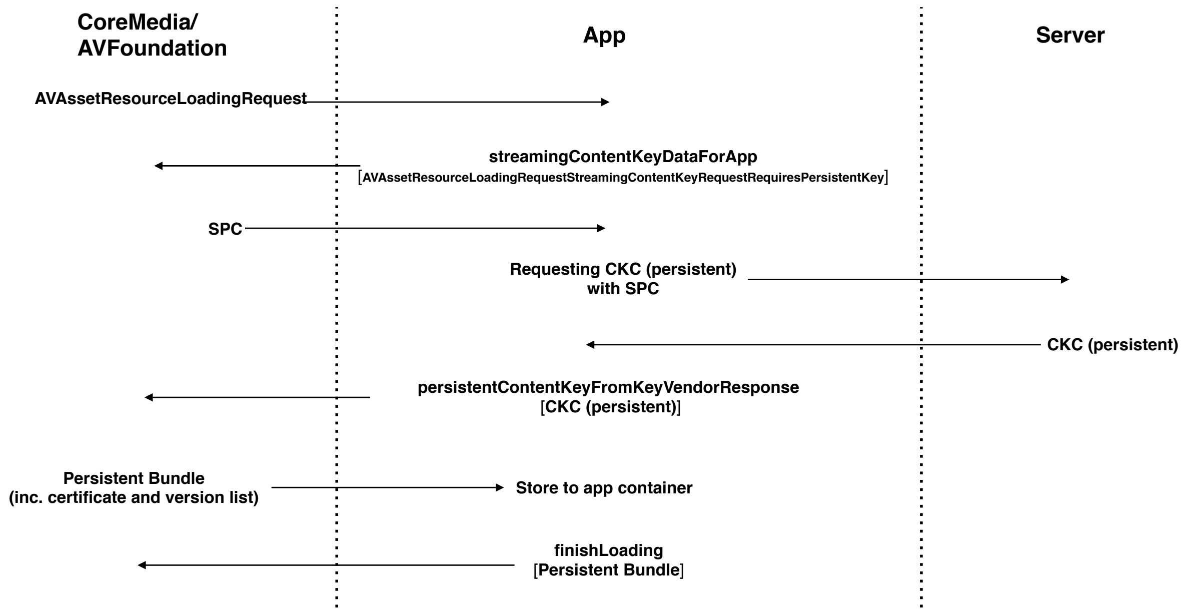
Figure 6-2 Persistent key request

Table 6-2 FPS error messages for presistence

Message	Description
-42799	This error code is returned when a security update is issued and the existing persistent key format is no longer supported. In this case, the application must request a new persistent key from the server.

-42800	This error code is returned when persistent key has expired. The server always has an option to add a validity period for each key it issues for offline playback. Once the validity period is over iOS will refuse to decrypt offline content protected with such key and indicate the error with this error code. It is up to the application developers to decide whether to request a new key from the server or treat this error condition as a permanent expiry; for example, a recorded sporting event must not be playable after 48 hours and no key renewal is possible. To use the best practice and achieve maximum flexibility, always send a key-renewal request to the server and let the server decide whether to allow the renewal.

Persistent Key Request


CoreMedia/ AVFoundation
App
Server
AVAssetResourceLoadingRequest
streamingContentKeyDataForApp
[AVAssetResourceLoadingRequestStreamingContentKeyRequestRequiresPersistentKey]
SPC
Requesting CKC (persistent) with SPC
CKC (persistent)
persistentContentKeyFromKeyVendorResponse [CKC (persistent)]
Persistent Bundle (inc. certificate and version list)
Store to app container
finishLoading [Persistent Bundle]

Offline Rental Support

Starting with iOS 11.0, FPS supports a new feature to enable offline rental. This feature allows the server to specify two expiration times for content. This “dual expiry” is similar to how the rental of iTunes movies functions.

Storage and Playback Expiry

The server can use the new Offline Key TLLV (see below) to specify two different duration periods for the downloaded content:

1. Storage duration (seconds). This specifies the maximum time the key stays valid prior to playback being started. Measured from license acquisition time.
2. Playback duration (seconds). This specifies the maximum time the key stays valid after playback has been started. Measured from the first playback start time.

After content is downloaded and the content license is acquired, the app may report to the server the remaining key validity duration using a Sync SPC (see below). This allows the server and the client to be synchronized on when the license to play the downloaded asset will expire.

Example: A user rents a movie; the server sets the storage duration to 2,592,000 (30 days) and the playback duration to 86,400 (24 hours). The user has up to 30 days to start watching the movie and 24 hours to finish watching it after starting the playback. If the app requests a Sync SPC to be created prior to user starting playback, the “Duration to expiry” field of the Sync TLLV will be set to 2,592,000 minus the number of seconds passed since license was downloaded. The same SPC requested after playback has started will contain a Sync TLLV with the “Duration to expiry” field set to 86,400 minus the number of seconds since playback was started.

Offline Key

Table 7-1 Offline Key TLLV

Field name	Byte range	Description
TLLV tag	0-7	An 8-byte value of `0x6375d9727060218c`.
Total length	8-11	The total length of this TLLV block in bytes. The total length is determined by the amount of padding at the end of the block, if any; it must be a multiple of 16 and greater than 31.
Value length	12-15	The length of the content of this TLLV block in bytes.
Version	16-19	TLLV version. Currently supported version is 1.

Field name	Byte range	Description
Reserved	20-23	This field is reserved and must be set to 0.
Content ID	24-39	Unique content ID of the downloaded asset assigned by the server. This value will be returned to the server in a Sync TLLV.
Storage duration	40-43	Asset storage validity duration in seconds. Starts at license acquisition time. A value of zero means no limit.
Playback duration	44-47	Asset playback validity duration in seconds. Starts at asset first playback time. A value of zero means no limit.
Padding	48-*n* (*padding_size*)	Random values that fill out the TLLV to a multiple of 16 bytes. See the description of the `Padding` field in Table 2-2.

To enable dual expiry for an asset the KSM should add an Offline Key TLLV to the returned CKC.

Note: An Offline Key TLLV cannot be used in the same CKC payload as an already existing Content Key Duration TLLV. If both of these TLLVs found in the CKC the processing of the CKC will stop and an Invalid CKC error will be returned.

Since the CKC is not interpreted by the client app, the server should notify the app that the asset is using dual expiry so that the client knows that a Sync SPC can be generated.

Sync

Table 7-2 Sync TLLV

Field name	Byte range	Description
TLLV tag	0-7	An 8-byte value of `0x77966de1dc1083ad`.
Total length	8-11	The total length of this TLLV block in bytes. The total length is determined by the amount of padding at the end of the block, if any; it must be a multiple of 16 and greater than 31.
Value length	12-15	The length of the content of this TLLV block in bytes.
Version	16-19	TLLV version. Currently supported version is 1.
Reserved	20-23	This field is reserved and must be set to 0.

Field name	Byte range	Description
Content ID	24-39	Unique content ID of the downloaded asset received in Offline Key TLLV.
Duration to expiry	40-43	Remaining license validity time in seconds. It will be set to 0 if the license has expired, and to 0xFFFFFFFF if the license doesn't have an expiry date.
Padding	44-*n* (*padding_size*)	Random values that fill out the TLLV to a multiple of 16 bytes. See the description of the `Padding` field in Table 2-2.

An app can obtain a Sync SPC via the method `[AVContentKeySession makeSecureTokenForExpirationDateOfPersistableContentKey:completionHandler:]`.

This method will fail unless the persistable content key was constructed from a CKC that included an Offline Key TLLV.

The app should send the resulting SPC to the KSM. The KSM should be modified to distinguish between Sync SPCs and other SPCs. Any SPC may contain a Sync TLLV, so you need to check the Sync TLLV for validity. Valid Sync TLLVs will have the Version field set to one. Any other value is invalid. A Sync SPC is one with a valid Sync TLLV.

Using the FPS SDK and Tools

Apple provides registered FPS developers with a server software development kit (SDK) containing reference materials, code, and tools to support FPS development. Optionally, registered developers may obtain additional tools and test streams from Apple that support the creation and testing of encrypted HLS streams. See http://developer.apple.com/streaming/ for those helpful downloads.

FairPlay Streaming SDK Contents

The FPS SDK contains the following items:

- This *FairPlay Streaming Programming Guide*
- A KSM reference implementation written in C
- A set of test streams
- A sample iOS app demonstrating the implementation of an `AVAssetResourceLoader` delegate to handle FPS key requests
- A set of development keys
- A CKC verification tool that contains SPC and CKC test vectors
- A sample JavaScript implementation of FPS for Safari on OS X

Testing the Key Security Modules

In order to perform testing of your KSM implementation, the FPS Server SDK package contains a number of pre-generated SPC and CKC test vectors and a verification utility `verify_ckc`.

Note that this utility can only be used to test your KSM implementation when using the test values for the private key and fixed DASk value. It cannot be used to test client side application nor to validate CKCs produced with your KSM implementation using production credentials.

1. Use the private key provided in the Development Certificate and Private Key file in the `pKeyPem` variable in the `SKDServerUtils.c` file
2. Use the DASk provided in the `spc_internal_values_v2.txt` file in the Verify CKC tool (`verify_ckc`). Populate this value into `spcData->DAS_k` in the `SKDServer.c:SKDServerProcessEncrypted_SK_R1` function as a replacement for the `Dfunction`.
3. Use the `verify_ckc` tool and test vectors to ensure the KSM can produce a valid CKC.

Make use of the SPC test vectors to simulate the client and exercise the KSM logic. The CKCs returned by the KSM are verified with the `verify_ckc` command-line tool.

Verifying your SPC processing in your KSM

The `verify_ckc` utility will parse the SPC and print a report with the details about each TLLV present in the SPC, as shown in *Parsing the SPC*.

Listing 8-1 Parsing the SPC

```
$ ./verify_ckc -s SPC-CKC-Tests/FPS/spc1.bin
CKC/SPC Sanity Test v. 1.07
========================= Begin SPC Data ===============================
     SPC container size 2688
SPC Encryption Key -
  92 66 48 b9   86 1e c0 47   1b a2 17 58   85 1c 3d da
SPC Encryption IV -
  5d 16 44 ea   ec 11 f9 83   14 75 41 e4   6e eb 27 74
================ SPC TLLV List ================
 [SK ... R1] Integrity Tag -- b349d4809e910687
    Tag size:     0x10
    Tag length:     0x40
    Tag value:
                        54 a1 6b e0   13 7e f2 59   ab 3e 4f c7   96 90 82 5f

 [SK ... R1] Tag -- 3d1a10b8bffac2ec
    Tag size:     0x70
    Tag length:     0x100
    Tag value:
                        4f 45 d8 5c   e2 62 73 10   1a 97 f3 30   81 c1 d0 4a
                        93 b2 dd 03   55 e3 63 72   9d 92 a4 5a   45 ce 8d 25
                        8b 0c 08 aa   65 1c 09 64   97 6b f0 94   4d 28 25 f3
                        ac 8d de 7e   d2 31 4f a0   ef 3f b4 5b   97 a2 26 e8
                        c5 36 6d ef   e5 f1 e1 2b   d7 b7 21 98   a4 a8 f2 65
                        3a 0e f0 de   8c 37 a4 7c   3c 40 f0 12   e1 5c 8b 59
                        3d f1 2d 4b   01 60 3a 97   35 7e 6a e0   a1 1c a3 e3

 AR Tag -- 89c90f12204106b2
    Tag size:     0x10
    Tag length:     0xd0
    Tag value:
                        f3 c6 9d 1e   8c c4 27 5a   6d 32 86 d3   32 61 3e 13

 R2 tag -- 71b5595ac1521133
    Tag size:     0x15
    Tag length:     0xb0
    Tag value:
                        11 f7 be 61   2c a9 5e f5   e0 07 ce 51   89 6a e4 50
                        2c a3 d8 80   1b -- -- --   -- -- -- --   -- -- -- --

 Asset ID Tag -- 1bf7f53f5d5d5a1f
    Tag size:     0x12
    Tag length:     0x80
    Tag value:
                        aa bb cc dd   ee ff aa bb   cc dd ee ff   aa bb cc dd
                        ee ff -- --   -- -- -- --   -- -- -- --   -- -- -- --

 Transaction ID Tag -- 47aa7ad3440577de
    Tag size:     0x08
    Tag length:     0x70
    Tag value:
                        14 73 e5 cc   53 e1 e5 d6   -- -- -- --   -- -- -- --

 Return Request Tag -- 19f9d4e5ab7609cb
    Tag size:     0x38
    Tag length:     0x60
    Tag value:
                        1b f7 f5 3f   5d 5d 5a 1f   47 aa 7a d3   44 05 77 de
                        f9 11 f0 4d   a5 4b f5 99   ba 08 cc 74   da c9 17 6d
                        13 0d 99 4c   b8 94 b9 e3   66 c8 23 f3   79 b8 7b b5
                        18 d4 2c 5f   8e 54 5a 4b   -- -- -- --   -- -- -- --

DASk Value:
    d8 7c e7 a2   60 81 de 2e   8e b8 ac ef   3a 6d c1 79

SPC SK Value:
    af b4 6e 7b   f5 f3 15 96   c1 c6 76 dc   15 e1 4d c6

SPC [SK..R1] IV Value:
    4f 45 d8 5c   e2 62 73 10   1a 97 f3 30   81 c1 d0 4a

=========================== End SPC Data =================================
```

You can use this information when debugging your KSM implementation. For example you can print out TLLVs decrypted by your KSM and compare these values against the data reported by `verify_ckc` utility.

Verifying your CKC creation in your KSM

The `verify_ckc` utility can also be used to check the validity of the CKC created by your KSM. You can run the `verify_ckc` utility as shown in Validating the CKC. Please note that in order to decrypt the CKC the utility must be invoked with both SPC and matching CKC files. For your reference the package includes the sample CKCs, which also can be used in the `verify_ckc` utility.

Listing 8-2 Validating the CKC

```
$ ./verify_ckc -s SPC-CKC-Tests/FPS/spc1.bin -c SPC-CKC-Tests/FPS/ckc1.bin
CKC/SPC Sanity Test v. 1.07
========================= Begin SPC Data ===============================
     SPC container size 2688
SPC Encryption Key -
  92 66 48 b9   86 1e c0 47   1b a2 17 58   85 1c 3d da
SPC Encryption IV -
  5d 16 44 ea   ec 11 f9 83   14 75 41 e4   6e eb 27 74
================ SPC TLLV List ================
 [SK ... R1] Integrity Tag -- b349d4809e910687
    Tag size:     0x10
    Tag length:     0x40
    Tag value:
                        54 a1 6b e0   13 7e f2 59   ab 3e 4f c7   96 90 82 5f

 [SK ... R1] Tag -- 3d1a10b8bffac2ec
    Tag size:     0x70
    Tag length:     0x100
    Tag value:
                        4f 45 d8 5c   e2 62 73 10   1a 97 f3 30   81 c1 d0 4a
                        93 b2 dd 03   55 e3 63 72   9d 92 a4 5a   45 ce 8d 25
                        8b 0c 08 aa   65 1c 09 64   97 6b f0 94   4d 28 25 f3
                        ac 8d de 7e   d2 31 4f a0   ef 3f b4 5b   97 a2 26 e8
                        c5 36 6d ef   e5 f1 e1 2b   d7 b7 21 98   a4 a8 f2 65
                        3a 0e f0 de   8c 37 a4 7c   3c 40 f0 12   e1 5c 8b 59
                        3d f1 2d 4b   01 60 3a 97   35 7e 6a e0   a1 1c a3 e3

 AR Tag -- 89c90f12204106b2
```

```
    Tag size:     0x10
    Tag length:     0xd0
    Tag value:
                        f3 c6 9d 1e   8c c4 27 5a   6d 32 86 d3   32 61 3e 13

 R2 tag -- 71b5595ac1521133
    Tag size:     0x15
    Tag length:     0xb0
    Tag value:
                        11 f7 be 61   2c a9 5e f5   e0 07 ce 51   89 6a e4 50
                        2c a3 d8 80   1b -- -- --   -- -- -- --   -- -- -- --

 Asset ID Tag -- 1bf7f53f5d5d5a1f
    Tag size:     0x12
    Tag length:     0x80
    Tag value:
                        aa bb cc dd   ee ff aa bb   cc dd ee ff   aa bb cc dd
                        ee ff -- --   -- -- -- --   -- -- -- --   -- -- -- --

 Transaction ID Tag -- 47aa7ad3440577de
    Tag size:     0x08
    Tag length:     0x70
    Tag value:
                        14 73 e5 cc   53 e1 e5 d6   -- -- -- --   -- -- -- --

 Return Request Tag -- 19f9d4e5ab7609cb
    Tag size:     0x38
    Tag length:     0x60
    Tag value:
                        1b f7 f5 3f   5d 5d 5a 1f   47 aa 7a d3   44 05 77 de
                        f9 11 f0 4d   a5 4b f5 99   ba 08 cc 74   da c9 17 6d
                        13 0d 99 4c   b8 94 b9 e3   66 c8 23 f3   79 b8 7b b5
                        18 d4 2c 5f   8e 54 5a 4b   -- -- -- --   -- -- -- --

DASk Value:
    d8 7c e7 a2   60 81 de 2e   8e b8 ac ef   3a 6d c1 79

SPC SK Value:
    af b4 6e 7b   f5 f3 15 96   c1 c6 76 dc   15 e1 4d c6

SPC [SK..R1] IV Value:
    4f 45 d8 5c   e2 62 73 10   1a 97 f3 30   81 c1 d0 4a

=========================== End SPC Data =================================
=========================== Begin CKC Data =================================
AES IV value:
    0x00000000000000000000000000000000
```

```
AR Key Value:
    0xcb0d802549396b9c8d636d5e64594cbe

CKC Data Length 768
 CK Tag -- 58b38165af0e3d5a
    Tag size:     0x20
    Tag length:     0x20
    Tag value:
                        d5 fb d6 b8   2e d9 3e 4e   f9 8a e4 09   31 ee 33 b7
                        3d 56 43 97   87 8b 70 43   e1 54 31 f1   f8 6b c5 62

 R1 Tag -- ea74c4645d5efee9
    Tag size:     0x2c
    Tag length:     0x40
    Tag value:
                        27 52 00 8e   1c 11 e2 24   e8 eb 07 ee   c4 a0 9d 17
                        44 0a 63 72   d5 dc 21 09   e5 50 ec ac   98 60 61 3f
                        8b 7a 8b e6   b4 5a 69 83   2d 9e 8c e7   -- -- -- --

 Asset ID Tag -- 1bf7f53f5d5d5a1f
    Tag size:     0x12
    Tag length:     0x80
    Tag value:
                        aa bb cc dd   ee ff aa bb   cc dd ee ff   aa bb cc dd
                        ee ff -- --   -- -- -- --   -- -- -- --   -- -- -- --

 Transaction ID Tag -- 47aa7ad3440577de
    Tag size:     0x08
    Tag length:     0x70
    Tag value:
                        14 73 e5 cc   53 e1 e5 d6   -- -- -- --   -- -- -- --

MATCHED! Return Request Tag 1bf7f53f5d5d5a1f
MATCHED! Return Request Tag 47aa7ad3440577de
MATCHED! Return Request Tag f911f04da54bf599
MATCHED! Return Request Tag ba08cc74dac9176d
MATCHED! Return Request Tag 130d994cb894b9e3
MATCHED! Return Request Tag 66c823f379b87bb5
MATCHED! Return Request Tag 18d42c5f8e545a4b

Info: SPC and CKC R1 key values match.

Info: CKC decryption and parsing was successful.

========================== End CKC Data =================================
```

Debugging KSM

The `verify_ckc` utility helps you debug your KSM implementation by printing all the TLLVs included in the CKC.

For example you can inspect CK Tag (`58b38165af0e3d5a`) TLLV which contains Content Key encrypted with the session key against values printed by your KSM. If you are using the Server Reference Implementation then the Content Key and encrypted Content Key can be found in `SKDServer.c`, function `SKDServerFillCKCData()`, variable `ckcData->ck`. This variable contains the unencrypted Content Key prior to calling `SKDServerAESEncryptDecrypt()`, and the encrypted Content Key after this call is completed.

Document Revision History

This table describes the changes to *FairPlay Streaming Programming Guide.*

Date	Notes
2017-08-21	**Added Capabilities TLLV and sections on Offline FairPlay Streaming and Offline Rental Support.**
2015-09-15	**Added macOS support and Lease/Rental Support features. Added clarification concerning the initialization vector (IV) in an FPS m3u8 playlist and behavior of the client when making key requests for segments in a playlist.**
2015-06-08	**New document that describes how to implement FairPlay Streaming encryption in HTTP Live Streaming media.**

Apple Inc.
Copyright © 2017 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, AirPlay, Apple TV, FairPlay, macOS, and Safari are trademarks of Apple Inc., registered in the U.S. and other countries.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.