FairPlay Streaming Server SDK

Key Server Module Guide for Rust

v5.1

Contents

Introduction

Apple FairPlay Streaming (FPS) is a protocol to securely deliver content keys and their policies from a key server module (KSM) to client devices to enable playback of FairPlay Streaming protected content. To learn more about FPS and to download the latest software development kit (SDK), see https://developer.apple.com/streaming/fps/.

Apple provides registered FPS developers with an SDK that contains reference material, code, and tools to support FPS development. The SDK is available in Swift and Rust for macOS and Linux. You can choose to build your server using either operating system in the steps below. The Swift examples use an Apache HTTPD server and the Rust examples use a Rust HTTP server. Optionally, you may obtain additional tools and test streams from Apple that support the creation and testing of encrypted HLS streams.

This guide covers the KSM portion of the server SDK. For information about the client portion, refer to the `README.md` files in the `Client` folder.

The `Key Server Module` folder contains the following items:

- The KSM reference.
- A precompiled library with a corresponding header file. The library performs sensitive cryptographic operations and is only available in binary format. The library is compiled for macOS (universal binary) and Linux (x86_64 and arm64), and exposes the following function: `FPSStatus KSMCreateKeyPayload(KSMKeyPayload* ksmPayload);`
- A set of development credentials:
  - The FPS certificate bundle, for both 1024- and 2048-bit certificates.
  - The FPS private keys, for both 1024- and 2048-bit certificates.
  - The FPS provisioning data.
- Server playback contexts (SPCs) created using development credentials.
- A content key context (CKC) verification tool for use with the development test vectors.

Important: You can use development credentials only to test sample key requests. They don't work with Apple devices.

1. Build your server for testing on Linux

This process builds a FairPlay Streaming key server module as a Rust library that exposes a C-compatible API, and uses a Rust HTTPD server.

Before starting, ensure the following:

1. You install Rust. See https://www.rust-lang.org/tools/install for instructions.
2. You install perl using the following command:

   ```
   apt-get install perl
   ```

   or

   ```
   yum install perl
   ```

3. You install gcc using the following command:

   ```
   apt-get install gcc
   ```

   or

   ```
   yum install gcc
   ```

Build the library

In Terminal, use the following commands to compile the Rust library:

```
cd Development/Key_Server_Module/Rust

cargo build --features test_credentials --lib
```

After building the library, run test cases to ensure the library behaves as you expect. You can only use the provided test cases with the provided development credentials. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command runs all test inputs:

```
cargo test --test regression_tests --features test_credentials
```

Optionally, you can run a test on a single input, as in the following example:

```
cargo run --bin fpssdk_local --features test_credentials \
../Test_Inputs/iOS/spc_ios_hd_lease_2048.json
```

Build and run the server

After you build the Rust library for testing, you can build the server using the following command:

```
cargo build --features test_credentials --bin fpssdk_server
```

Then use the following command to run the server:

```
cargo run --features test_credentials --bin fpssdk_server
```

Test your server

To ensure the server behaves as you expect, use curl to send an SPC to the server. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command sends a test SPC to the test server:

```
// In a new terminal window:

curl -d @../Test_Inputs/iOS/spc_ios_hd_lease_2048.json \
localhost:8080
```

Use the `parse_fps` utility to verify the server's output and ensure that it returns the expected CKC. See Verify the key server module output for instructions.

After testing is complete, you can clean the build artifacts by using the following command:

```
cargo clean
```

2. Build your server for testing in macOS

This process builds a FairPlay Streaming key server module as a Rust library that exposes a C-compatible API, and uses a Rust HTTPD server. This process includes building libraries, running test cases, and verifying outputs.

Before starting, ensure you install Rust. See https://www.rust-lang.org/tools/install for instructions.

Build the library

In Terminal, use the following command to build the library for testing:

```
cd Development/Key_Server_Module/Rust

cargo build --features test_credentials --lib
```

To allow the cryptographic library to run, use the following command:

```
xattr -d com.apple.quarantine prebuilt/macOS/libfpscrypto.dylib
```

Next, run test cases to ensure the library behaves as you expect. You can only use the provided test cases with the provided development credentials. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command runs all test inputs:

```
cargo test --test regression_tests --features test_credentials
```

Optionally, you can run a test on a single input, as in the following example:

```
cargo run --bin fpssdk_local --features test_credentials \
../Test_Inputs/iOS/spc_ios_hd_lease_2048.json
```

Build and run the server

Use the following command to build the server:

```
cargo build --features test_credentials --bin fpssdk_server
```

Then use the following command to run the server:

```
cargo run --features test_credentials --bin fpssdk_server
```

Test your server

After building and running the Rust server, send a test SPC to the test server to ensure the server behaves as you expect. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command sends a test SPC to the test server:

```
// In a new terminal window:

curl -d @../Test_Inputs/iOS/spc_ios_hd_lease_2048.json \
localhost:8080
```

Use the `parse_fps` utility to verify the server's output and ensure that it returns the expected CKC. See Verify the key server module output for instructions on verifying the server output.

After testing is complete, you can clean the build artifacts by using the following command:

```
cargo clean
```

3. Obtain production credentials

To use FPS in production, you generate private keys and certificate signing requests (CSRs), submit the CSRs to the Apple Developer website, and download the resulting credentials. This process involves generating both 1024-bit and 2048-bit RSA keys, creating CSRs, and then obtaining a certificate bundle and provisioning data through the Apple Developer website. You need to integrate these files into your SDK before using the server in production. This guide explains how to copy the private keys and provisioning data into the appropriate locations within the SDK for Rust.

Generate a private key and a certificate signing request

Before you can submit a request to obtain production credentials, you need to obtain private keys and CSRs. Use the commands below to create new 1024-bit and 2048-bit private keys. If you already have an existing 1024-bit private key used with FairPlay Streaming, you may reuse it and only create the 2048-bit one.

The following Terminal input is an example of how to create a 1024-bit RSA private key and a certificate signing request:

```
   openssl req -out csr_1024.csr -new -newkey rsa:1024 \
-keyout priv_key_1024.pem \
-subj /CN=SubjectName/OU=OrganizationalUnit/O=Organization/C=US
```

The following Terminal input is an example of how to create a 2048-bit RSA private key and a certificate signing request:

```
   openssl req -out csr_2048.csr -new -newkey rsa:2048 \
-keyout priv_key_2048.pem \
-subj /CN=SubjectName/OU=OrganizationalUnit/O=Organization/C=US
```

Keep your private keys in a safe and secure location. You will need them to deploy your FairPlay Streaming key server.

Submit the request

After you create your CSRs, you need to use them to create your certificate bundle and provisioning data through the Apple Developer website.

1. Go to the Apple Developer website at: https://developer.apple.com/.
2. Click Certificates, IDs, & Profiles.
3. Log in.
4. Click the Certificates tab.
5. Click the Add button (+).
6. Select FairPlay Streaming Certificate, and then click the Continue button.
7. Select SDK 5.x.
8. Under 2048-bit Certificate, click Choose File and select your 2048-bit certificate signing request file.

9. Under 1024-bit Certificate, either select to reuse your previous certificate, or click Choose File and select your 1024-bit certificate signing request file.
10. Click Continue.
11. Click the Download button to download the `fps_bundle.zip` file.

Receive certificate bundle and provisioning data

The `fps_bundle.zip` file contains:

- `fps_certificate.bin`: this file should be hosted on your servers. Clients will need to fetch this certificate and use it when making license requests for your key server.
- `provisioning_data.bin`: this file is used by the FairPlay Streaming key server at runtime. See the integration instructions below.

Integrate credentials

The SDK uses three credentials at runtime. The FairPlay Streaming key server reads the data inside the following files at runtime:

- `priv_key_1024.pem`:

  Copy this file to `Rust/src/extension/credentials/priv_key_1024.pem`.

- `priv_key_2048.pem`:

  Copy this file to `Rust/src/extension/credentials/priv_key_2048.pem`.

- `provisioning_data.bin`:

  Copy this file to `Rust/src/extension/credentials/provisioning_data.bin`.

Note: If you prefer to load your credentials another way, you can edit the `getPrivateKey()` and `getProvisioningData()` functions in `Rust/src/extension/structures/extension.rs`.

4. Build your server for production on Linux

Before starting, ensure the following:

1. You obtain production credentials and integrate the credentials into your library.
2. You install Rust. See https://www.rust-lang.org/tools/install for instructions.
3. You install perl using the following command:

   ```
   apt-get install perl
   ```

   or

   ```
   yum install perl
   ```

4. You install gcc using the following command:

   ```
   apt-get install gcc
   ```

   or

   ```
   yum install gcc
   ```

Build the library

Use the following command to build the library for production:

```
cargo build --release --lib
```

Build and run your server

After building the library, use the following command to build the server:

```
cargo build --release --bin fpssdk_server
```

Then use the following command to run the server:

```
cargo run --release --bin fpssdk_server
```

5. Build your server for production in macOS

Before starting, ensure the following:

1. You obtain production credentials and integrate the credentials into your library.
2. You install Rust. See https://www.rust-lang.org/tools/install for instructions on installation.

Build the library

Use the following command to build the Rust library for production:

```
cargo build --release --lib
```

Run and run your server

After building the library, use the following command to build the server:

```
cargo build --release --bin fpssdk_server
```

Then use the following command to run the server:

```
cargo run --release --bin fpssdk_server
```

6. Verify the key server module output

Before starting, ensure you install Python 3. See https://www.python.org/downloads/ for instructions. The FPS Server SDK package includes pre-generated SPC test vectors and a verification utility called `parse_fps` to test your KSM implementation. This utility takes in the Base64 or binary-encoded SPC and, optionally, the corresponding CKC field (Base64 or binary-encoded format), and then decrypts and prints the data within them.

1. Install the crypto package for python:

   ```
   pip3 install –U PyCryptodome
   ```

2. Copy one `libfpscrypto` for whichever architecture `parse_fps` is running on into the `parse_fps/` directory.

   ```
   cp Rust/prebuilt/macOS/libfpscrypto.dylib parse_fps/

   cp Rust/prebuilt/x86_64–unknown–linux–gnu/libfpscrypto.so \
   parse_fps/

   cp Rust/prebuilt/aarch64–unknown–linux–gnu/libfpscrypto.so \
   parse_fps/
   ```

3. Run the following command:

   ```
   cd Development/Key_Server_Module/parse_fps

   python3 –m fps.parse_fps ––spc ./samples/sample_spc.b64 \
   ––ckc ./samples/sample_ckc.b64
   ```

You can find a sample output using the SPC and CKC in `parse_fps/samples:`.

Add production credentials

You can use production credentials to verify input and output using `parse_fps` as well. To add production credentials, add extra fields to the `fps/cfg/credentials_sdk.py` file in the following format:

```
{

    'pkey_1024': 'credentials/priv_key_1024.pem',

    'pkey_2048': 'credentials/priv_key_2048.pem',

    'cert': 'credentials/fps_certificate.bin',

    'provisioning_data': 'credentials/provisioning_data.bin',

}
```

After adding the credentials, you can use `parse_fps` to test and verify production input and output. The format for using the tool is the same as when using test credentials. You don’t need to remove the fields for test credentials when adding production credentials.

7. Customize the key server module

The key server module has an extension component in the SDK that you can modify for additional functionality and customization. The extension is where you add any custom code, structures, variables, and logic.

Many of the Base functions call a corresponding `Custom` function to allow for customization in the extension. These extension functions are located in `Rust/src/extension/structures/extension.rs`. The file contains functions that allow you to change the behavior at many points of the server code, such as handling additional fields in the input JSON file, or adding additional fields to the output JSON file.

Logging

Calls to the key server module result in the creation of an output JSON file. This output file may contain a successful status code and the resulting license, or it may contain an error code. To help with debugging error scenarios, by default, the KSM also prints log messages. Throughout the code, there are two main types of logging: production and debug. Both allow you to customize the format.

Production logs:

- Function name: `fpsLogError!()`
- Enabled in debug and release builds.
- Print to `stderr`.
- Print with a prefix (timestamp, name, version, file, function, line number), which you can customize in `logInitCustom()`.

Debug logs:

- Function name: `Log::Debug!()`
- Use the Rust `env_logger` crate https://docs.rs/env_logger/latest/env_logger/
- Disabled in release builds by enabling the `release_max_level_off` feature of the Log crate in `Cargo.toml`
- Print to stderr.
- You can customize the format in `logInitCustom()`.

Determine your business rules

The `checkBusinessRules()` function provides a suggested set of business rules that includes the following:

- UHD content requires security level Main and HDCP Type 1.
- HD content requires security level Baseline or higher, and HDCP Type 0.
- SD content requires security level Baseline or higher.
- Audio content has no special requirements.

You can edit or add rules to match your business requirements. For example, if your rules need additional input types, or you have different requirements for 720 p and 1080 p, or SDR and HDR, you can add types to the `content-type` field in the input JSON file, which the `ContentType` enumeration describes and the `parseAssetInfoCustom()` function parses.

Customize using extension structures

You can find custom structures in the `extension_structures` file. These structures are all originally empty, so you add fields for custom implementations. The structures are:

Extension	Description
`SKDExtension`	A custom extension to modify how to deliver, validate, and store keys on the client side.
`FPSOperationExtension`	A custom extension to modify the set of operations that the SDK performs, such as playback or key requests.
`AssetInfoExtension`	A custom extension to modify information about the asset that's streaming, such as its encryption status, content ID, or custom attributes that the streaming app uses.
`ServerCtxExtension`	A custom extension to modify the server-side operations, such as license requests or validation of the server's identity.
`FPSResultsExtension`	A custom extension to modify the result of an operation, such as a license or playback request.
`FPSResultExtension`	A custom extension to modify individual operation results.
`SPCDataExtension`	A custom extension to modify the server playback context data by customizing its validation for specific content playback scenarios.
`CKCDataExtension`	A custom extension to modify the content key context data by customizing how the key server processes and delivers key responses to the client.
`SPCContainerExtension`	A custom extension to modify the SPC container by defining additional encapsulation logic for secure key request data before sending it to the key server.
`CKCContainerExtension`	A custom extension to modify the CKC container by customizing content key response data for enhanced security.
`KeyDurationExtension`	A custom extension to modify key duration settings by defining specific playback duration rules.

8. API reference

The key server module consists of two main components: `Base` and `Extension`. The `Base` structure contains all parts of the FairPlay Streaming logic that are necessary for the protocol to function and isn't intended to be modified. `Extension` allows for added functionality and customization and is intended to be modified. The Rust folder contains a `Sources` folder with Rust source files and a manifest file (`Cargo.toml`) that defines the package and its contents, and contains details about the build targets and their dependencies. The `Base` class contains the basic functionality to parse SPCs and create CKCs. Each part of this process is divided into a separate directory for better readability.

Base structures

The following code is part of the `Base` structures in `base_fps_structures.rs`:

`FPSOperations`

Rust	

`pub struct FPSOperations {`
`  pub operationsPtr: Vec<FPSOperation>,`
`}`	

API	Description
`pub struct FPSOperations {  }`	Contains a vector of `FPSOperation`. This is necessary if receiving multiple requests in the same JSON file.
`pub operationsPtr:` `Vec<FPSOperation>`	A list of `FPSOperation` objects.

`FPSOperation`

Rust	

`pub struct FPSOperation {`
`  pub id: u64,`
`  pub spc: Vec<u8>,`
`  pub isCheckIn: bool,`
`  pub assetInfo: AssetInfo,`
 	
`  pub extension:`
`  extension_structures::FPSOperationExtension`
`}`	

API	Description

`pub struct FPSOperation { }`	The basic structure of a FairPlay Streaming key request after parsing the JSON file. Includes the ID of the request, the SPC, and the `AssetInfo`.
`pub id: u64`	A unique ID for the operation request.
`pub spc: Vec<u8>`	An `Array` of the server playback context data.
`pub isCheckIn: bool`	A Boolean value that indicates whether this is a check-in request.
`pub assetInfo: AssetInfo`	Information about the asset.
`pub extension: extension_structures::FPSOperationExtension`	A structure for extending `FPSOperation` with custom fields.

`AssetInfo`

Rust

```
pub struct AssetInfo {
    pub key: Vec<u8>,
    pub iv: Vec<u8>,
    pub isCKProvided: bool,
    pub leaseDuration: u32,
    pub rentalDuration: u32,
    pub playbackDuration: u32,

    pub hdcpReq: u64,

    pub licenseType: u32,
    pub streamId: Vec<u8>,
    pub isStreamIdSet: bool,
    pub titleId: Vec<u8>,
    pub isTitleIdSet: bool,

    pub extension:
        extension_structures::AssetInfoExtension
}
```

API	Description
`pub struct AssetInfo { }`	Contains information about the requested asset.
`pub key: Vec<u8>`	The encryption key.
`pub iv: Vec<u8>`	The initialization vector.
`pub isCKProvided: bool`	A Boolean value that indicates whether the content key is provided.

`pub leaseDuration: u32`	The license duration, in seconds, starting from SPC creation time. Required for lease requests. Mutually exclusive with `offline-hls`.
`pub rentalDuration: u32`	The rental duration, in seconds, starting from asset download time.
`pub playbackDuration: u32`	The lease duration, in seconds, starting from asset first playback time.
`pub hdcpReq: u64`	Specifies the High-bandwidth Digital Content Protection(HDCP) for the asset.
`pub licenseType: u32`	The license associated with the asset streaming or offline.
`pub streamId: Vec<u8>`	The stream ID associated with the asset.
`pub titleId: Vec<u8>`	The title ID associated with the asset.
`pub isStreamIdSet: bool`	A Boolean value that indicates whether the stream ID is set.
`pub isTitleIdSet: bool`	A Boolean value that indicates whether the title ID is set.
`pub extension: extension_structures::AssetInfoExtension`	A structure for extending `AssetInfo` with custom fields.

`FPSResults`

Rust	

`pub struct FPSResults {`
`  pub resultPtr: Vec<FPSResult>,`
 	
`  pub extension:`
`  extension_structures::FPSResultsExtension`
`}`	

API	Description
`pub struct FPSResults { }`	Contains a vector of `FPSResult`. This is necessary when sending multiple requests at once (much like `FPSOperations`).

`ub resultPtr: Vec<FPSResult>`	A list of `FPSResult` objects.

`pub extension:`
 	
`extension_structures::FPSResultsExtension`
	A structure for extending `FPSResults` with custom fields.

`FPSResult`

Rust	

`pub struct FPSResult {`
`  pub id: u64,`
`  pub status: FPSStatus,`
`  pub hu: Vec<u8>,`
`  pub ckc: Vec<u8>,`
 	
`  pub sessionId: u64,`
 	
`  pub isCheckIn: bool,`
`  pub syncServerChallenge: u64,`
`  pub syncFlags: u64,`
`  pub syncTitleId: Vec<u8>,`
`  pub durationToRentalExpiry: u32,`
`  pub recordsDeleted: usize,`
`  pub deletedContentIDs: Vec<u8>,`
 	
`  pub deviceIdentitySet: bool,`
`  pub fpdiVersion: u32,`
`  pub deviceClass: u32,`
`  pub vendorHash: Vec<u8>,`
`  pub productHash: Vec<u8>,`
`  pub fpVersionREE: u32,`
`  pub fpVersionTEE: u32,`
`  pub osVersion: u32,`
`  pub vmDeviceInfo: Option<VMDeviceInfo>,`
 	
`  pub extension:`
`  extension_structures::FPSResultExtension`
`}`	

API	Description
`pub struct FPSResult { }`	The structure containing the response information before the key server serializes it into a JSON file.
`pub id: u64,`	A unique ID.
`pub status: FPSStatus,`	The status of the operation.
`pub hu: Vec<u8>,`	An `Array` representing the HU identifier.
`pub ckc: Vec<u8>,`	An `Array` representing the content key context.

`pub sessionId: u64`	The session identifier.
`pub isCheckIn: bool`	A Boolean value that indicates whether this is a check-in response.
`pub syncServerChallenge: u64`	A server challenge for synchronization.
`pub syncFlags: u64`	A flag that holds synchronization status.
`pub syncTitleId: Vec<u8>`	An `Array` representing the synchronization title ID.
`pub durationToRentalExpiry: u32`	The duration remaining until the rental content expires, in seconds.
`pub recordsDeleted: usize`	The number of deleted records.
`pub deletedContentIDs: Vec<u8>`	A list of deleted content IDs.
`pub deviceIdentitySet: bool`	A Boolean value that indicates whether the device identity is successfully set.
`pub fpdiVersion: u32`	The current version of the FPDI.
`pub deviceClass: u32`	The type of device for streaming.
`pub vendorHash: Vec<u8>`	A hash value representing the device vendor.
`pub productHash: Vec<u8>`	A hash value representing the product.
`pub fpVersionREE: u32`	The FairPlay version in the Rich Execution Environment (REE).
`pub fpVersionTEE: u32`	The FairPlay version in the Trusted Execution Environment (TEE).
`pub osVersion: u32`	The operating system version running on the device.
`pub vmDeviceInfo: Option<VMDeviceInfo>`	Host and guest device information, if running in a virtual machine.
`pub extension: extension_structures::FPSResultExtension`	A structure for extending `FPSResult` with custom fields.

KSMKeyPayload

Rust

```
pub struct KSMKeyPayload {
    pub version: u64,
    pub contentKey: *const u8,
    pub contentKeyLength: u64,
    pub contentIV: *const u8,
    pub contentIVLength: u64,
    pub contentType: u64,
    pub SK_R1: *const u8,
    pub SK_R1Length: u64,
    pub R2: *const u8,
    pub R2Length: u64,
    pub R1Integrity: *const u8,
    pub R1IntegrityLength: u64,
    pub supportedKeyFormats: *const u64,
    pub numberOfSupportedKeyFormats: u64,
    pub cryptoVersionUsed: u64,
    pub provisioningData: *const u8,
    pub provisioningDataLength: u64,
    pub certHash: *const u8,
    pub certHashLength: u64,
    pub clientHU: *const u8,
    pub clientHULength: u64,
    pub contentKeyTLLVTag: u64,
    pub contentKeyTLLVPayload: *mut u8,
    pub contentKeyTLLVPayloadLength: u64,
    pub R1: *mut u8,
    pub R1Length: u64,
}
```

API	Description
`pub struct KSMKeyPayload { }`	A structure containing all the information to send to the cryptographic library to create the content key TLLV.
`pub version: u64,`	The version number.
`pub contentKey: *const u8`	A pointer to the content key.
`pub contentKeyLength: u64`	The length of the content key.
`pub contentIV: *const u8`	A pointer to the content IV.
`pub contentIVLength: u64`	The length of the content IV.
`pub contentType: u64`	The type of content.
`pub SK_R1: *const u8`	A pointer to session key R1.
`pub SK_R1Length: u64`	The length of session key R1.
`pub R2: *const u8`	A pointer to R2.
`pub R2Length: u64`	The length of R2.
`pub R1Integrity: *const u8`	A pointer to the R1 integrity value.

`pub R1IntegrityLength: u64`	The length of the R1 integrity value.
`pub supportedKeyFormats: *const u64`	A pointer to the list of supported key formats.
`pub numberOfSupportedKeyFormats: u64`	The number of supported key formats.
`pub cryptoVersionUsed: u64`	The version of the cryptographic library.
`pub provisioningData: *const u8`	A pointer to the provisioning data.
`pub provisioningDataLength: u64`	The length of the provisioning data.
`pub certHash: *const u8`	A pointer to the certificate hash.
`pub certHashLength: u64`	The length of the certificate hash.
`pub clientHU: *const u8`	A pointer to the client HU.
`pub clientHULength: u64`	The length of the client HU.
`pub contentKeyTLLVTag: u64`	The tag associated with the content key TLLV.
`pub contentKeyTLLVPayload: *mut u8`	A pointer to the payload of the content key TLLV.
`pub contentKeyTLLVPayloadLength: u64`	The length of the content key TLLV payload.
`pub R1: *mut u8`	A pointer to R1.
`pub R1Length: u64`	The length of R1.

The following code is part of the `Base` structures in `base_server_structures.rs`:

`FPSServerCtx`

Rust	

`pub struct FPSServerCtx {`
`  pub spcContainer: FPSServerSPCContainer,`
`  pub ckcContainer: FPSServerCKCContainer,`
`  pub isStreamIdSet: bool,`
`  pub streamId: Vec<u8>,`
`  pub isTitleIdSet: bool,`
`  pub titleId: Vec<u8>,`
 	
`  pub extension:`
`  extension_structures::ServerCtxExtension,`
`}`	

API	Description

`pub struct FPSServerCtx { }`	Contains the SPC and CKC containers, along with stream and title IDs. This is the base structure that the key server most commonly sends to functions.
`pub spcContainer: FPSServerSPCContainer`	A container that holds the SPC data.
`pub ckcContainer: FPSServerCKCContainer`	A container that holds the CKC data.
`pub isStreamIdSet: bool`	A Boolean value that indicates whether the stream ID is set.
`pub streamId: Vec<u8>`	The stream ID.
`pub isTitleIdSet: bool`	A Boolean value that indicates whether the title ID is set.
`pub titleId: Vec<u8>`	The title ID.
`pub extension: extension_structures::ServerCtxExtension`	A structure for extending `FPSServerCtx` with custom fields.

`FPSServerSPCContainer`

Rust	
`pub struct FPSServerSPCContainer { pub version: u32, pub reservedValue: u32, pub aesKeyIV: Vec<u8>, pub aesWrappedKey: Vec<u8>, pub aesWrappedKeySize: usize, pub certificateHash: Vec<u8>, pub spcDecryptedData: Vec<u8>, pub spcDataSize: usize, pub spcDataOffset: usize, pub spcData: FPSServerSPCData, pub extension: extension_structures::SPCContainerExtension, }`	
API	Description
`pub struct FPSServerSPCContainer { }`	Contains all the parsed information from the SPC, including version, SPC encryption, AES key and IV, and TLLV data.

`pub version: u32`	Stores the current SPC version.
`pub reservedValue: u32`	Stores a reserved value.
`pub aesKeyIV: Vec<u8>`	The AES key IV value.
`pub aesWrappedKey: Vec<u8>`	The AES wrapped key value.
`pub aesWrappedKeySize: usize`	Stores the size of the AES wrapped key.
`pub certificateHash: Vec<u8>`	The hash of the certificate value.
`pub spcDecryptedData: Vec<u8>`	Stores the decrypted SPC data.
`pub spcDataSize: usize`	Stores the size of the SPC data.
`pub spcDataOffset: usize`	The SPC offset data.
`pub spcData: FPSServerSPCData`	The parsed SPC data.
`pub extension: extension_structures::SPCContainerExtension`	The additional extension data for the SPC container.

`FPSServerCKCContainer`

Rust	

`pub struct FPSServerCKCContainer {`
`  pub version: u32,`
`  pub aesKeyIV: Vec<u8>,`
`  pub ckc: Vec<u8>,`
`  pub ckcDataPtr: Vec<u8>,`
`  pub ckcData: FPSServerCKCData,`
`}`	

API	Description
`pub struct FPSServerCKCContainer { }`	Contains information to add to the CKC, including version, CKC encryption, AES key and IV, and the CKC payload.
`pub version: u32`	Stores the version of the CKC.
`pub aesKeyIV: Vec<u8>`	The key IV value.
`pub ckc: Vec<u8>`	The CKC data as an `Array`.
`pub ckcDataPtr: Vec<u8>`	Points to the CKC data as an `Array`.
`pub ckcData: FPSServerCKCData`	Contains the CKC data.

FPSServerSPCData

Rust

```
pub struct FPSServerSPCData {
    pub antiReplay: Vec<u8>,
    pub sk: Vec<u8>,
    pub hu: Vec<u8>,
    pub r2: Vec<u8>,
    pub r1: Vec<u8>,
    pub skR1IntegrityTag: Vec<u8>,
    pub skR1Integrity: Vec<u8>,
    pub skR1: Vec<u8>,
    pub assetId: Vec<u8>,
    pub versionUsed: u32,
    pub versionsSupported: Vec<u32>,
    pub returnTLLVs: Vec<FPSServerTLLV>,
    pub returnRequest: FPSServerTLLV,
    pub clientFeatures: FPSServerClientFeatures,
    pub spcDataParser: FPSServerSPCDataParser,
    pub playInfo: FPSServerMediaPlaybackState,
    pub streamingIndicator: u64,
    pub transactionId: u64,

    pub syncServerChallenge: u64,
    pub syncFlags: u64,
    pub syncTitleId: Vec<u8>,
    pub durationToRentalExpiry: u32,
    pub recordsDeleted: usize,
    pub deletedContentIDs: Vec<u8>,

    pub clientCapabilities: Vec<u8>,

    pub isSecurityLevelTLLVValid: bool,
    pub supportedSecurityLevel: u64,
    pub clientKextDenyListVersion: u32,

    pub deviceIdentity: FPSDeviceIndentity,
    pub deviceInfo: FPSDeviceInfo,

    pub numberOfSupportedKeyFormats: u32,
    pub supportedKeyFormats:
        [u64; FPS_MAX_KEY_FORMATS],

    pub vmDeviceInfo: Option<VMDeviceInfo>,

    pub extension:
        extension_structures::SPCDataExtension,
}
```

API	Description
`pub struct FPSServerSPCData { }`	Contains parsed information from the SPC TLLVs after decryption.
`pub antiReplay: Vec<u8>`	The antireplay value.
`pub sk: Vec<u8>`	An `Array` that represents the session key for decryption.

`pub hu: Vec<u8>`	An `Array` that represents the HU.
`pub r2: Vec<u8>`	An `Array` that represents the r2.
`pub r1: Vec<u8>`	An `Array` that represents the r1.
`pub skR1IntegrityTag: Vec<u8>`	An `Array` that represents the integrity tag for session key R1.
`pub skR1Integrity: Vec<u8>`	An `Array` that represents the integrity value for session key R1.
`pub skR1: Vec<u8>`	An `Array` that represents the session key r1.
`pub assetId: Vec<u8>`	The asset ID that identifies the asset.
`pub versionUsed: u32`	The version number.
`pub versionsSupported: Vec<u32>`	A list of supported versions.
`pub returnTLLVs: Vec<FPSServerTLLV>`	A list of TLLVs that the server returns.
`pub returnRequest: FPSServerTLLV`	A specific TLLV that the server returns as part of the request.
`pub clientFeatures: FPSServerClientFeatures`	The client features.
`pub spcDataParser: FPSServerSPCDataParser`	An instance for parsing the SPC data.
`pub playInfo: FPSServerMediaPlaybackState`	The current state of media playback.
`pub streamingIndicator: u64,`	A streaming indicator for content.
`pub transactionId: u64`	A unique transaction identifier.
`pub syncServerChallenge: u64,`	A challenge number for synchronizing the server.
`pub syncFlags: u64,`	Flags for controlling synchronization settings.
`pub syncTitleId: Vec<u8>,`	An `Array` representing the synchronization title ID.
`pub durationToRentalExpiry: u32,`	The duration remaining until rental expiration.
`pub recordsDeleted: usize,`	The number of deleted records.
`pub deletedContentIDs: Vec<u8>`	A list of deleted content IDs.

`pub clientCapabilities: Vec<u8>`	An `Array` representing the client's capabilities.
`pub isSecurityLevelTLLVValid: bool`	A Boolean value that indicates whether the security level TLLV is valid.
`pub supportedSecurityLevel: u64`	The supported security level value.
`pub clientKextDenyListVersion: u32`	The version of the client's Kext deny list.
`pub deviceIdentity: FPSDeviceIndentity`	The identity information for the device interacting with the server.
`pub deviceInfo: FPSDeviceInfo`	The information about the device interacting with the server.
`pub numberOfSupportedKeyFormats: u32`	The number of key formats that the client supports.
`pub supportedKeyFormats: [u64; FPS_MAX_KEY_FORMATS]`	A list of supported key formats.
`pub vmDeviceInfo: Option<VMDeviceInfo>`	Host and guest device information, if running in a virtual machine.
`pub extension: extension_structures::SPCDataExtension`	The additional extension data.

FPSServerCKCData

Rust

```
pub struct FPSServerCKCData {
    pub ck: Vec<u8>,
    pub iv: Vec<u8>,
    pub r1: Vec<u8>,
    pub keyDuration: FPSServerKeyDuration,
    pub hdcpTypeTLLVValue: u64,

    pub contentKeyTLLVTag: u64,
    pub contentKeyTLLVPayload: Vec<u8>,

    pub extension:
        extension_structures::CKCDataExtension,
}
```

API	Description
`pub struct FPSServerCKCData { }`	Data to add to the CKC TLLVs.
`pub ck: Vec<u8>`	An `Array` of the content key that encrypts or decrypts the content.

`pub iv: Vec<u8>`	An `Array` of the initialization vector that initializes encryption.
`pub r1: Vec<u8>`	An `Array` of the r1 values.
`pub keyDuration: FPSServerKeyDuration`	An instance of `FPSServerKeyDuration` and the duration that the key is valid.
`pub hdcpTypeTLLVValue: u64`	A value indicating the HDCP requirements.
`pub contentKeyTLLVTag: u64`	The tag associated with the content key.
`pub contentKeyTLLVPayload: Vec<u8>`	The payload of the content key TLLV.
`pub extension: extension_structures::CKCDataExtension`	An extension that holds additional data.

`FPSDeviceIdentity`

Rust	

`pub struct FPSDeviceIndentity {`
`  pub isDeviceIdentitySet: bool,`
`  pub fpdiVersion: u32,`
`  pub deviceClass: u32,`
`  pub vendorHash: Vec<u8>,`
`  pub productHash: Vec<u8>,`
`  pub fpVersionREE: u32,`
`  pub fpVersionTEE: u32,`
`  pub osVersion: u32,`
`}`	

API	**Description**
`pub struct FPSDeviceIndentity { }`	Information to help identify the client device type, including vendor and product hashes, REE and TEE versions (only for third-party devices), and OS version (only for Apple products). **Note:** Only devices running FairPlay client software released in 2021 or later send this TLLV. Prioritize using it over `FPSDeviceInfo` for client device type information.
`pub isDeviceIdentitySet: bool`	A Boolean value that indicates whether the device identity is set.

`pub fpdiVersion: u32`	The current FPDI version.
`pub deviceClass: u32`	The device class for categorizing the type of device.
`pub vendorHash: Vec<u8>`	A hash representing the vendor of the device.
`pub productHash: Vec<u8>`	A hash representing the product of the device.
`pub fpVersionREE: u32`	The current version of the REE on the device, only applicable for third-party devices.
`pub fpVersionTEE: u32`	The current version of the TEE on the device, only applicable for third-party devices.
`pub osVersion: u32`	The current operating system version, only applicable for Apple products.

`FPSDeviceInfo`

Rust

```
pub struct FPSDeviceInfo {
    pub isDeviceInfoSet: bool,
    pub deviceType: u64,
    pub osVersion: u32,
}
```

API	Description
`pub struct FPSDeviceInfo { }`	Basic information about the client device, including device type and OS version. **Note:** This is a legacy TLLV. Use `FPSDeviceIdentity` instead, if available.
`pub isDeviceInfoSet: bool`	A Boolean value that indicates whether the device info is set.
`pub deviceType: u64`	The client device type.
`pub osVersion: u32`	The current operating system version.

FPSServerTLLV

Rust	

`pub struct FPSServerTLLV {`
`  pub tag: u64,`
`  pub value: Vec<u8>,`
`}`	

API	Description
`pub struct FPSServerTLLV { }`	Contains the tag and value fields for a TLLV.
`pub tag: u64`	The tag identifying the specific field.
`pub value: Vec<u8>`	An `Array` that stores the value associated with the tag.

FPSServerClientFeatures

Rust	

`pub struct FPSServerClientFeatures {`
`  pub supportsOfflineKeyTLLV: bool,`
`  pub supportsOfflineKeyTLLVV2: bool,`
`  pub supportsSecurityLevelBaseline: bool,`
`  pub supportsSecurityLevelMain: bool,`
`  pub supportsHDCPTypeOne: bool,`
 	
`  pub extension:`
`  extension_structures::ClientFeaturesExtension`
`}`	

API	Description
`pub struct FPSServerClientFeatures { }`	Contains fields that indicate whether the client supports certain features, including offline key V1 vs V2, Baseline vs Main security levels, and HDCP Type 1.
`pub supportsOfflineKeyTLLV: bool`	A Boolean value that indicates whether the client supports the offline key TLLV format for version 1.
`pub supportsOfflineKeyTLLVV2: bool`	A Boolean value that indicates whether the client supports the offline key TLLV format for version 2.
`pub supportsSecurityLevelBaseline: bool`	A Boolean value that indicates whether the client supports the baseline security level.

`pub supportsSecurityLevelMain: bool`	A Boolean value that indicates whether the client supports the main security level.
`pub supportsHDCPTypeOne: bool`	A Boolean value that indicates whether the client supports HDCP Type 1.
`pub extension: extension_structures::ClientFeaturesExtension`	An extension that holds additional client features.

FPSServerSPCDataParser

Rust	
`pub struct FPSServerSPCDataParser { pub currentOffset: usize, pub TLLVs: Vec<FPSServerTLLV>, pub parsedTagValues: Vec<u64>, }`	
API	Description
`pub struct FPSServerSPCDataParser { }`	An intermediary data structure for parsing the SPC. It holds the current offset within the SPC data and parsed tags, along with the parsed TLLVs.
`pub currentOffset: usize,`	The current offset within the SPC.
`pub TLLVs: Vec<FPSServerTLLV>,`	A current list of parsed TLLVs.
`pub parsedTagValues: Vec<u64>`	A list of parsed tag values, where each value corresponds to a specific tag that the key server reads from the SPC data.

FPSServerMediaPlaybackState

Rust	
`pub struct FPSServerMediaPlaybackState { pub date: u32, pub playbackState: u32, pub playbackId: u64, }`	
API	Description
`pub struct FPSServerMediaPlaybackState { }`	Contains information such as the date, playback state, and playback ID.

`pub date: u32`	The date associated with the playback state.
`pub playbackState: u32`	The current playback state of the media.
`pub playbackId: u64`	A unique identifier for the playback session.

FPSServerKeyDuration

Rust

```
pub struct FPSServerKeyDuration {
    pub leaseDuration: u32,
    pub rentalDuration: u32,
    pub playbackDuration: u32,
    pub keyType: u32,

    pub extension:
        extension_structures::KeyDurationExtension,
}
```

API	Description
`pub struct FPSServerKeyDuration {`	Contains information about different key durations, including lease, rental, and playback duration, along with the key type.
`pub leaseDuration: u32`	The duration of the lease for the key, in seconds, before it expires.
`pub rentalDuration: u32`	The rental duration of the key.
`pub playbackDuration: u32`	The duration of content playback for the key.
`pub keyType: u32`	The type of key — whether it's for leasing, rental, or playback.
`pub extension: extension_structures::KeyDurationExtension`	Additional extended information related to key durations.

`VMDeviceInfo`

Rust	

`pub struct VMDeviceInfo {`
`  pub hostDeviceClass: FPSDeviceClass,`
`  pub hostOSVersion: u32,`
`  pub hostVMProtocolVersion: u32,`
`  pub guestDeviceClass: FPSDeviceClass,`
`  pub guestOSVersion: u32,`
`  pub guestVMProtocolVersion: u32,`
`}`	

API	Description
`pub struct VMDeviceInfo {`	Contains information about the VM host and guest, if the requesting device is running in a VM.
`pub hostDeviceClass: FPSDeviceClass`	Device class of the VM host.
`pub hostOSVersion: u32`	OS version of the VM host.
`pub hostVMProtocolVersion: u32`	FairPlay virtualization protocol version used by the VM host.
`pub guestDeviceClass: FPSDeviceClass`	Device class of the VM guest.
`pub guestOSVersion: u32`	OS version of the VM guest.
`pub guestVMProtocolVersion: u32`	FairPlay virtualization protocol version used by the VM guest.

Parse input JSON files

JSON parsing begins by calling `parseRootFromString()` or `parseRootFromJson()`. These functions take in a string or a file, respectively, and convert it into the Rust internal JSON data structure. Then the program parses the JSON files into usable data structures. After ingesting the JSON files into the structure, the `processOperations()` and `parseOperations()` functions complete the parsing. These functions convert the data into a usable `FPSOperations` data structure, and decrypt and parse the SPC and its TLLVs. The following table describes the purpose of the functions for parsing input JSON files:

Function	Description
`processOperations()`	The main function for handling a request. It takes in the ingested JSON file and returns the output JSON file.
`parseOperations()`	Handles parsing the input JSON file into the usable `FPSOperations` data structure.

`parseCreateCKCOperation()`	Parses a single `create-ckc` request into a usable `FPSOperation`. Parses ID, SPC, check-in, and any asset info.
`parseAssetInfo()`	If there is asset info available, this function parses elements such as content key and IV, lease duration, and offline HLS.

Decrypt and parse SPC TLLVs

The `readNextTLLV()` function reads the TLLV data, and `parseTLLV()` parses it. First, `parseTLLV()` uses the TLLV tag to identify the specific TLLV to parse and then calls the parsing function for that TLLV (parsing functions each have their own file and are in the `parse_spc_TLLVs/` directory). If `parseTLLV()` doesn't find a matching TLLV tag, it drops the TLLV. There's a full list of TLLVs under the `FPSTLLVTagValue` enumeration. The following table describes the functions you use to decrypt and parse SPC TLLVs:

Function	Description
`createResults()`	Sets the result ID and calls `genCKCWithCKAndIV()`.
`genCKCWithCKAndIV()`	Checks the SPC version, and calls functions to parse the SPC, query a database if needed, populate the `fpsResult` data structure, create the key payload, and generate the CKC.
`parseSPCV1()`	Calls functions to parse the SPC container, decrypt the SPC, parse the decrypted SPC, and check supported features.
`parseSPCContainer()`	Parses the unencrypted section of the SPC: • Version • 16-byte IV • Encrypted AES wrapped key • Certificate hash • 4-byte SPC size
`decryptSPCData()`	Decrypts the AES wrapped key using RSA OAEP, and decrypts the SPC using the decrypted AES wrapped key.
`parseSPCData()`	Initializes play info, then loops through SPC, reading each TLLV and parsing it. It then saves the TLLVs in case the CKC needs to return them, and extracts TLLVs that the return tags specify. **Note:** TLLV parsing exists under `parseTLLV()`.

`checkSupportedFeatures()`	Sets feature flags based on the parsed SPC information: • `supportsLease` • `supportsOfflineKeyTLLV` • `supportsOfflineKeyTLLVV2` • `supportsSecurityLevelBaseline` • `supportsSecurityLevelMain` • `supportsHDCPTypeOne`
`populateServerCtxResult()`	Copies needed SPC information into the results structure.

Create CKC TLLVs

The key server creates the content key within the `createContentkeyPayloadCustomImpl()` function in the extension. However, you need to call the `KSMCreateKeyPayload()` function from the object files in the prebuilt directory. After generating the content key payload, the program calls `generateCKCV1()` to create the CKC TLLVs. After parsing all of the `create-ckc` requests and creating the resulting CKCs, the key server serializes the final result JSON file with the `serializeResults()` function. The following table describes the functions you use to create CKC TLLVs:

Function	Description
`serializeCKCData()`	Serializes TLLVs to add to the CKC: • Content key • R1 • Server return tags • HDCP requirement • Security level
`deriveAntiReplayKey()`	Derives the encryption key from the antireplay seed and
`encryptCKCData()`	Encrypts the CKC using the antireplay key.
`serializeCKCContainer()`	Serializes the encrypted CKC, along with additional unencrypted fields: • 4-byte version • 4-byte reserved field • 16-byte IV • CKC data size • CKC data

Input/Output JSON format

By default, the key server module requires asset information (content key, content IV, content type, HDCP requirement, and so forth.) to pass as part of the input JSON file. Alternatively, if asset information is unknown at the time of input JSON creation, you can implement the `queryDatabaseCustom()` function to look up asset information based on the `asset-id` inside the SPC. The following table provides the input JSON fields:

Field name	Type	Description
`fairplay-streaming-request`	Object	Contains the information for the FairPlay streaming request.
`version`	Integer	The input format version. This is always 1.
`create-ckc`	Object	Links inputs and outputs in the same request.
`id`	Integer	The value for correlating inputs and outputs if there are multiple in the same request.

Field name	Type	Description
`spc`	String	A server playback context that the FairPlay library generates on the client device.
`check-in`	Boolean	`True` for sync operations.
`asset-info`	Object Array	Information about the requested asset.
`content-key`	String	Content key as a hex string. Not required for lease renewals.
`content-iv`	String	Content IV as a hex string. Not required for lease renewals.
`content-type`	String	The type of content, such as `audio`, `hd`, and `uhd`. If not present, defaults to `unknown`.
`lease-duration`	Integer	The license duration, in seconds, starting from SPC creation time. Required for lease requests. Mutually exclusive with `offline-hls`.
`hdcp-type`	Integer	HDCP requirement: –1 for no HDCP `0` for HDCP Type 0 1 for HDCP Type 1. If not present, the default is `0`.
`offline-hls`	Object	Required for persistent license requests. Mutually exclusive with `lease-duration`.
`rental-duration`	Integer	The rental duration, in seconds, starting from asset download time.
`playback-duration`	Integer	The lease duration, in seconds, starting from asset first playback time.
`stream-id`	String	The unique ID of each HLS substream.
`title-id`	String	The ID of a title (program). Same for all HLS substreams of a given title.

Below is a sample input JSON file for a streaming request:

```
{
    "fairplay-streaming-request": {
        "version": 1,
        "create-ckc": [
            {
                "id": 1,
                "spc": “AAAAAgAAAACIiMWpQhMDI6pMnx2nfIiIMoaz9xQol...”,
                "asset-info": [
                    {
                        "content-key": "0102030405060708090A0B0C0D0E0F10",
                        "content-iv": "F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF",
                        "content-type": "hd",
                        "hdcp-type": 0,
                        "lease-duration": 1200
                    }
                ]
            }
        ]
    }
}
```

Below is a sample input JSON file for an offline request:

```
{
    "fairplay-streaming-request": {
        "version": 1,
        "create-ckc": [
            {
                "id": 1,
                "spc": “AAAAAQAAAACIiMWpQhMDI6pMnx2nfIiITwKbcTan4UepHXxB...”,
                "asset-info": [
                    {
                        "content-key": "ab07634237ab000fad0d2f29797c8f74",
                        "content-iv": "9a52030a2eb83b14f2e7989b8869c894",
                        "content-type": "uhd",
                        "hdcp-type": 1,
                        "offline-hls": {
                            "stream-id": "171062176140000000000000000000000",
                            "title-id": "5E785A7C000000000000000000000000",
                            "rental-duration": 2592000,
                            "playback-duration": 172800
                        }
                    }
                ]
            }
        ]
    }
}
```

The following table provides the output JSON fields:

Field name	Type	Description
`fairplay-streaming-response`	Object	Contains the information for the FairPlay streaming request.
`create-ckc`	Object	Links inputs and outputs in the same request.
`id`	Integer	The value for correlating inputs and outputs if there are multiple in the same request.
`status`	Integer	Returns a status. A value of `0` is success; other values are errors.
`hu`	String	An anonymized unique ID of the playback device as a hex string.
`ckc`	String	The output content key context as a Base64 string. It’s not present for sync operations.

Field name	Type	Description
`check-in-server-challenge`	String	A unique challenge that the server generates. This is only for sync requests.
`check-in-flags`	String	Specifies a sync TLLV flag setting. This is only for sync requests.
`check-in-title-id`	String	The title ID in the `offline-hls` parameters of the input request as a hex string. This is only for sync requests.
`duration-left`	String	The duration until the expiration for rentals. This is only for sync requests.
`check-in-stream-id`	String Array	An array of content IDs that the key server checks in as hex strings. The is only for sync requests.
`fpdi-version`	Integer	The device identity TLLV version. This is only present when receiving a device identity TLLV.
`device-class`	Integer	Specifies the device class, such as Apple Mobile. This is only present when receiving a device identity TLLV.
`vendor-hash`	String	A unique identifier for the device vendor as a hex string. This is only present when receiving a device identity TLLV.
`product-hash`	String	A unique identifier for a product as a hex string. This is only present when receiving a device identity TLLV.
`fps-ree-version`	String	The current version of FairPlay software running in `REE/userland` as a hex string. This is only present when receiving a device identity TLLV.
`fps-tee-version`	String	The current version of FairPlay software running in `TEE/kernel` as a hex string. This is only present when receiving a device identity TLLV.
`os-version`	String	The OS version as a hex string. This is only present when receiving a device identity TLLV.

Below is an example of an output JSON file:

```
{
    "fairplay-streaming-response": {
        "create-ckc": [
            {
                "id": 1,
                "status": 0,
                "hu": "DB27C96B93D9218D50943F1498A8055E69993C18",
                "ckc": “AAAAAQAAAAA83lRbbGmLWMgKQJnAtLi8AAABoKqMgPh8l6CWq...”
                "fpdi-version": 1,
                "device-class": 2,
                "vendor-hash": "CA0D91584DE3468C",
                "product-hash": "9A77725DAE435607",
                "fps-ree-version": "00000000",
                "fps-tee-version": "00000000",
                "os-version": "00100000"
            }
        ]
    }
}
```

For additional examples, see the `Test_Inputs` folder.

Revision history

SDK Version	Date	Notes
5.0	2024-10-08	New document that describes how to build, set up, and customize FairPlay Streaming Server SDK 5.
5.1	2025-05-05	Split Rust and Swift into separate documents. Minor edits and formatting changes. Added support for Linux ARM64. Added VM Device information.

Apple Inc.
One Apple Park Way
Cupertino, CA 95014