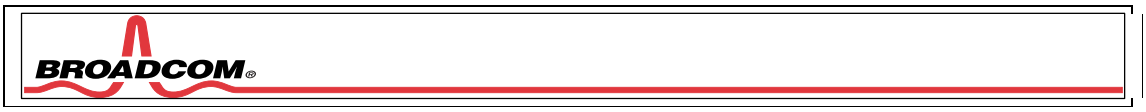

Broadcom DTCP-IP software stack
Version 3.1

User Guide

5300 California Ave. Irvine, CA 92617-USA • Phone: (949) 450-8700 • Fax: (949) 450-8710

TABLE OF CONTENTS

REVISION HISTORY

Revision Number	*Date*	*By*	*Change Description*
1.0	03/27/09	L. Sun	Initial Created from Reference Software User Guide
1.1	07/09/09	L. Sun	Major revision, Feature enhancement/bug fixes, added Brutus as DTCP client, Removed bcrypt library dependency, removed DTCP constants from source code.
1.1	08/16/10	L. Sun	Updated API description.
2.0	12/07/10	L. Sun	Updated API description, build flag , production key handling, etc.
3.0	02/11/11	L. Sun	Updated library to conform to V1SE 1.31
3.1	09/08/12	L. Sun	Updated API description, build procedure/flags, consolidated production key and test key binaries, etc.

1 Introduction

The Broadcom DTCP-IP library implements the DTCP-IP stack based on DTCP spec volume 1, rev 1.4 and supplement E rev 1.31. For DTCP-IP command set that this library currently supported, please refer to section 5.

1 References

Available in broadcom reference SW release documents

- Reference Software User Guide
- Nexus Development Guide
- Nexus Usage Guide

External Documents:

- DTCP Specification Volume 1 Revision 1.6 (Informational Version). (http://www.dtcp.com)
- DTCP Volume 1 Supplement E Mapping DTCP to IP, Revision 1.31 (Informational Version). (http://www.dtcp.com)

2 Prerequisites and dependencies

The user of Broadcom's DTCP-IP library must have:

- Broadcom's set-top reference software source or binary package.
- Broadcom set-top development environment (Linux kernel and Toolchain)
- Broadcom set-top reference platform.

It is also assumed that the user is familiar with building/installing Broadcom set-top reference software package.

The DTCP-IP library has dependencies on following libraries:

- Boradcom libraries:
  - Nexus lib (nexus.so).
  - OS lib (lib_os.so).
  - Playback_ip lib (libb_playback_ip.so).
  - Common DRM lib (libcmndr.so, libdrmrootfs.so)
  - Common crypto lib (libbcrypt.so)
  - Nexus security lib (libnexus_security.so)
- Open source libraries:
  - Openssl lib (libcrypto.so, libssl.so).

NOTE1: DTCP-IP lib doesn't depends on playback_ip directly, but the sample DTCP-IP application invoking DTCP-IP API through playback_ip lib.

3 Platform, Linux kernel and toolchains

The platform, kernel and tool-chain are the same as nexus package, except that some additional environment variables are needed as described in 3.1

3.1 Environment variables related to DTCP-IP Build

- **DTCP_IP_SUPPORT=y**

This will flag the build system to build playback_ip and settop API with DTCP-IP support.

- **DTCP_IP_HARDWARE_ENCRYPTION=y/n**

If this variable is set to "y", then the library will use hardware M2M DMA for DTCP-IP content ***encryption***, otherwise it will use pure software for the operation. The default is using software encryption.

- **DTCP_IP_HARDWARE_DECRYPTION=y/n**

If this variable is set to "y", then the library will use hardware M2M DMA for DTCP-IP content ***decryption***, otherwise it will use software for the operation. The default is to use software decryption.

- **NEXUS_COMMON_CRYPTO_SUPPORT=y**
- **BHSM_KEYLADDER=ON**
- **DTCP_IP_DATA_BRCM=y**
- **HSM_SOURCE_AVAILABLE=n**
- **NEXUS_SECURITY_KEYLADDER_EXTENSION_INC=$(path_to_nexus)/extentions/security/keyladder/$(CHIP)/keyladder_ext.inc**
- **NEXUS_SECURITY_USERCMD_EXTENSION_INC=$(path_to_nexus)/extensions/security/usercmd/$(CHIP)/usercmd_ext.inc**
- **NEXUS_EXTRALIBS=$(path_to_nexus)/modules/security/$(CHIP)/lib/libnexus_security.so**

These variables are security related build flags, some of the modules need special permission go gain access. If you don't have permission, please contact with STB security team.

4 Building and Running example applications

Sample server/client programs are provided along with the playback_ip lib, which is located in "nexus/lib/playback_ip/apps".
"ip_streamer" is a server application with DTCP-IP source functionality, "ip_client" is a client application. Both server and client support DTCP-IP over HTTP streaming ONLY.

4.1 Building the example applications

Although the release provided the pre-compiled example application, re-compilation might be needed for testing/evaluation purpose.

NOTE2: to compile the example application, you must have broadcom's set-top reference software source package installed on your build machine. The example application need the reference software package's header files to compile.

In addition to DTCP-IP related environment variables specified in 3.1, the following environment variables are also required for building example client/server application with DTCP-IP.

- Export the environment variables:
  - PLATFORM= your platform (e.g. 97425)
  - ARCH=mipsel-linux
  - BCHP_VER=B0 (your chip revision)
  - LINUX=/your brcm-linux kernel installation path
  - BUILD_SYSTEM=nexus
  - LIVEMEDIA_SUPPORT=y
  - MEDIA_AVI_SUPPORT=y
  - MEDIA_ASF_SUPPORT=y
  - NETACCEL_SUPPORT=n
  - PLAYBACK_IP_SUPPORT=y
  - RAP_AC3_SUPPORT=y
  - RAP_DDP_SUPPORT=y
  - RAP_DDP_TO_AC3_SUPPORT=y
  - RAP_MPEG_SUPPORT=y
  - SSL_SUPPORT=y

- cd into your nexus top directory.
- Run "***make –C lib/playback_ip/apps install***"
- When build complete, the sample apps and runtime shared library will be located in "***nexus/bin***" directory.

4.2. Customize the Makefile

- **DTCP_DEMO_MODE**

This symbol is for debugging only, turning on this symbol will result in lots of debugging message showing up in the Linux console, if on the STB, the "msg_modules" environment variable included "b_dtcp_ip", e.g. "export msg_modules=b_dtcp_ip"

- **CLOSE_SOCKET_ON_AKE_OK**

This symbol is for server stack only and is turned ON by default. The purpose of having this symbol is that for today's DTCP-IP enabled client (sink) devices, some vendors choose to close the DTCP-IP socket after successful AKE procedure, while some vendors choose to keep the socket open. So this symbol is provided for more flexibility.

4.2 Running the example applications (Testing key mode)

The following steps are required to run the example DTCP-IP client/server applications:

- Boot up Linux on set-top box.
- Copy the apps and shared libraries from your build machine's "***nexus/bin***" directory to set-top box. If you are using NFS mount, you may choose to copy the apps and shared libraries to NFS mount directory and export the LD_LIBRARY_PATH by typing "export LD_LIBRARY_PATH=/mnt/nfs" (assuming your NFS is mounted to /mnt/nfs directory of your set-top box).
- Copy the DTCP constants, RNG seeds, and dummy SRM message files to STB (or NFS mount directory), to the same path as where your sample apps are located. All data files required are located under *"nexus/lib/dtcp_ip//data"* directory.
- Start the server program by following command:

nexus ip_streamer –p5000 –d8000 –K0 –m /data/videos

Where :

-d 8000: DTCP-IP port number (default 80000).
-p 1234: HTTP streaming port number (default 5000)
-K0: specify to use DTCP-IP test key.
-m /data/videos : media file directory.

You can use "nexus ip_streamer –help" to obtain detailed command line option/description.
You should see following message prompt after successfully launched the server application:

**** 00:00:00.167 ip_streamer: Starting Streaming Thread*
**** 00:00:00.167 ip_streamer: Starting Streaming Thread*
**** 00:00:00.168 ip_streamer: Starting Streaming Thread*

The server then block-waiting for client connection request.

For client example, perform same steps as describe above for all apps, shared libraries, key data, RNG seeds, etc to STB(or NFS mount directory).

- Launch client application by following command:

nexus ip_client –d 192.168.1.10 –p1234 –t3 –K0 –u/sample2.mpg –S2 –n8000

Where:

-d 192.168.1.10 : The DTCP-IP server's ip address.
-p 1234: the HTTP streaming port number.
-t3: streaming protocol. (For DTCP, we only support HTTP streaming, which is #3)
-K0: specify to use DTCP-IP test key.

-u/sample2.mpg: URL for media file the client is requesting, so you must first copy this media file to your server side's media directory.
-S2: Security protocol, 2 is for DTCP-IP
-n 8000: is the DTCP-IP port number, ***it must be as same as the port number that*** **the server is using.**

NOTE3: the sample media file is not included in the release bundle, you should use your own media file for testing.

After successfully launched client application, it will try to connect with the server and perform AKE (Authentication and Key Exchange procedure). If A KE successfully finished, the client is authenticated with the server and the server then begins to encrypt media file and transfer it to client through HTTP protocol. The client side will decrypt and playback the stream. You should be able to watch the video playback if you have TV connected on the client set-top box.

4.3 Running the example applications (Production key mode)

To run the example application with production key, you need to request two sets of DRM bin files from Broadcom security team.
A DRM bin files securely handles the DRM keys. There are two ways to use these files:

- Flash the bin file into the board. Follow the instructions from Reference Security Software page.
  - This is the default use case.

- Copy the bin file under a folder accessible during application runtime.
  - This is non-default case, you need to modify the source file *BSEAV/lib/security/common_drm/third_party/dtcp_ip/dtcp_ip_vendor.c*
  - Pass the path of this DRM bin file to ***DRM_DtcpIp_Initialize()*** function.
  - Rebuild the library.

NOTE: At runtime, the application (ip_client/ip_streamer) parses the bin file to extract the DTCP-IP keys in a secure manner using common DRM.

After flashing the bin file, start the server application with following command:
nexus ip_streamer –p5000 –d8000 –K1 –m /data/videos

Where:
-K1 specify the server to use DRM based production key.
Other arguments are the same with running in testing key mode.
Start the client application with following command:
nexus ip_client –d 192.168.1.10 –p1234 –t3 –K1 –u/sample2.mpg –S2 –n8000
Where:
-K1 specify the client to use DRM based production key.

Other arguments are the same with running in testing key mode.

5. DTCP-IP API

Broadcom's DTCP-IP library was implemented based on DTCP specification Volume 1, rev 1.4 and Supplement E, Rev 1.31. The following AKE commands are supported:

- CHALLENGE
- RESPONSE
- RESPONSE2
- EXCHANGE _KEY
- SRM
- AKE_CANCEL
- CONTENT_KEY_REQ
- CAPABILITY_REQ
- CAPABILITY_EXCHANGE
- RTT_READY
- RTT_SETUP
- RTT_TEST
- RTT_VERIFY
- CONT_KEY_CONF

The following commands are not supported in release 3.0

- SET_DTCP_MODE
- BG-RTT_INITIATE

The library doesn't support all MOVE mode operation.

5.1 High level API

Application incorporates the DTCP functionalities by calling the high level API. The top level header file is *"$(dtcp_ip_release_dir)/include/b_dtcp_applib.h"*. The top level header file defines the API prototypes and data type that are required for calling DTCP-IP API. Please refer to the top level header file for more detail description for API function parameters, enumeration and function return code.

5.1.1 Initialize/cleanup the library

5.1.1.1. Library startup.

Both client (sink) and server(source) device must initialize the library before calling any other DTCP-IP functions. The function to initialize the library is:

void * DtcpAppLib_Startup(B_DeviceMode_T mode, bool use_pcp_ur, B_DTCP_KeyFormat_T key_format, bool ckc_check);

mode: The mode of the device, available enumeration includes:

```
B_DeviceMode_eSource,   /* Format-non-cognizant source function. */
```

```
B_DeviceMode_eSink,     /* Format-non-cognizant sink function. */
B_DeviceMode_eFCSource, /* Format-cognizant source function */
B_DeviceMode_eFCSink,   /* Format-cognizant sink function. */
B_DeviceMode_eRecord,   /* Format-non-cognizant recording function. */
B_DeviceMode_eFCRecord, /* Format-cognizant recording function. */
B_DeviceMode_eAudioFCSource,    /* Audio-Format-cognizant source function. */
B_DeviceMode_eAudioFCSink,      /* Audio-format-cognizant sink function.*/
B_DeviceMode_eAudioFCRecord,    /* Audio-format-cognizant recording function.*/
```

Depending on what type of device are you implementing, you should choose a proper "mode" value to pass it in.

use_pcp_ur: a boolean flag to indicate if your device supports PCP_UR. PCP_UR capability was added for V1SE 1.3. Broadcom's DTCP-IP lib version 2.0 or older doesn't support PCP_UR, This parameter is for backward compatibility.

If the library has been successfully initialized, a void type pointer to DTCP-IP context will be returned otherwise NULL will be returned. Application needs to save the returned context pointer, so that before existing from the application, it needs to pass the context pointer to following function to clean up the DTCP-IP lib:

key_format: an enum value to indicate the library which key to use, the enum is defined as:

```
typedef enum B_DTCP_KeyFormat
{
   B_DTCP_KeyFormat_eTest = 0,   /* Testing Key format */
   B_DTCP_KeyFormat_eCommonDRM = 1, /* DRM based production key format.*/
   B_DTCP_KeyFormat_eProduction  /* Obsoleted, legacy production key format*/
}B_DTCP_KeyFormat_T;
```

ckc_check: a Boolean indicating if the sink device will perform "Content Key Confirmation procedure". This flag is for compatibility purpose, since earlier DTCP-IP enabled source device might doesn't support CKC.

5.1.1.2. Library shutdown.

void DtcpAppLib_Shutdown(void * ctx);

ctx: The DTCP-IP context pointer obtained from DtcpAppLib_Startup() function call.

5.1.1.3. Initialize Hardware security parameters

If you wish to use Broadcom HW m2m DMA for encryption/decryption(The library was built with *DTCP_IP_HARDWARE_ENCRYPTION=y and DTCP_IP_HARDWARE_DECRYPTION=y).*

Then you must also call following function to initialize the hardware security module:

BERR_Code DtcpInitHWSecurityParams(void * nexusDmaHandle);

nexusDmaHandle: a Handle to nexus DMA object, if it's NULL, then the DTCP-IP lib will create a handle internally.
The following cleanup function need to be called before exiting from the application to release resources:
void DtcpCleanupHwSecurityParams(void);

5.1.2 Server (source) Functions

After library startup, application invokes the server stack listening function by calling:

int DtcpAppLib_Listen(void * ctx, const char * aSourceIP, unsigned short aSourcePort);

ctx: The DTCP context pointer obtained from ***DtcpAppLib_Startup().***
aSourceIP: The IP address server listening on, if it's NULL, it will listen to all address.
aSourcePort: The DTCP-IP AKE port number.

The server stack listening function is a non-blocking function, application might continue to perform other initialization task after above function call. For streaming server, for example, after above function call, the server might wait for incoming streaming connection. The DTCP-IP server stack will spawn a new thread whenever a new connection request is received from sink device. The new thread will perform AKE function.

For streaming server example, if the server received a media streaming request, the server should first try to retrieve the AKE handle by calling the function:

Void DtcpAppLib_GetSinkAkeSession(void * ctx, const char * aRemoteIP, void **aAkeHandle);
ctx: DTCP context pointer obtained from DtcpAppLib_Startup() function.
aRemoteIp: sink device's IP address.
aAkeHandle: pointer to the AKE session handle.

If the sink device has already passed AKE procedure before calling this function, then dereferencing *aAkeHandle* should points to the sink AKE session handle. If the sink device failed the AKE procedure, dereferencing *aAkeHandle* should be a NULL pointer, the server then should reject the streaming request.

If the server wishes to stop listening to AKE request, it should call following function:

int DtcpAppLib_CancelListen(void * ctx);

5.1.3 Client (Sink) Functions

The client application invokes the client stack by calling the function:

int DtcpAppLib_DoAke(void * ctx, const char *aRemoteIp, unsigned short aRemotePort, void ** AkeHandle);

ctx: DTCP context pointer obtained from DtcpAppLib_Startup() function.
aRemoteIp: Source device's IP address.
aRemotePort: DTCP-IP AKE port number.
AkeHandle: pointer to AKE session handle.

This function is a blocking function, it tries to connect to the specified DTCP-IP server device and perform AKE procedure. Application should check the return code, if it is "BERR_SUCCESS", then the client successfully authenticated with the server, and dereferencing "*AkeHandle*" should be non-NULL. The AkeHandle should be saved for later used by streaming interface function.

If the application wish to terminate the AKE session, it needs to call following function, and pass in an AKE handle obtained from DtcpAppLib_DoAke() function:

Int DtcpAppLib_CloseAke(void * ctx, void *aAkeHandle);
ctx: DTCP context pointer obtained from DtcpAppLib_Startup() function.
aAkeHandle: DTCP-IP AKE session handle obtained from DtcpAppLib_DoAke() function.

After the client device has been authenticated(AKE succeeded) with server, it will remain authenticated, however, if the client device didn't perform any DTCP-IP streaming functionality for 2 hours, the server will expire this AKE session's exchange key, therefore, the following function is provided for client device to verify it's current exchange key is still valid:

bool DtcpAppLib_VerifyExchKey(void *ctx, void * aAkeHandle);

ctx: DTCP context pointer obtained from DtcpAppLib_Startup() function.
aAkeHandle: DTCP-IP AKE session handle obtained from DtcpAppLib_DoAke() function.
If the function returned "false", client device must update its exchange key, if the client wish to remain authenticated and before requesting any DTCP protected stream from server:

int DtcpAppLib_GetNewExchKey(void *ctx, void *aAkeHandle);

ctx: DTCP context pointer obtained from DtcpAppLib_Startup() function.
aAkeHandle: DTCP-IP AKE session handle obtained from DtcpAppLib_DoAke() function.
Above function will perform AKE again with the server, but unlike the "***DtcpAppLib_DoAke***" function, it doesn't create new AKE session.

5.1.4 Streaming interface.

The stack provides a streaming interface, apart from AKE functions, enable applications to packetize/depcketize media data for DTCP-IP streaming transmitting/receiving. The library's streaming interface only supports HTTP protocol.

After AKE succeeded and server received HTTP streaming request form client, the server call following function to open a source stream:

void * DtcpAppLib_OpenSourceStream(void *aAkeHandle, B_StreamTransport_T TransportType, int contentLength, int cci, int content_type, int max_packet_size);

aAkeHandle: DTCP-IP AKE session handle obtained from DtcpAppLib_GetSinkAkeSession() function.
TransportType: Streaming protocol, the only supported type is B_StreamTransport_eHTTP.
contentLength: The length of the media content, if live streaming use DTCP_CONTENT_LENGTH_UNLIMITED which is defined as -1.
cci: embedded CCI (Copy Control Information) of the content. The application is responsible for parsing the source content to extract CCI and pass it to DTCP-IP layer. The DTCP-IP library recognize following CCI:

- B_CCI_eCopyFree,
- B_CCI_eNoMoreCopy,
- B_CCI_eCopyOneGeneration,
- B_CCI_eCopyNever

Based on the content CCI, the device mode and the type of the content, the library will determine the EMI (Encryption Mode Indicator) value used for DTCP-IP transmission. Please refer to V1SE1.31 for more detailed information about the relationship between EMI and CCI.
Content_type: type of the content, available value includes:

- B_Content_eAudioVisual,
- B_Content_eType1Audio,
- B_Content_eReserved1,
- B_Content_eReserved2,
- B_Content_eMPEG_TS,
- B_Content_eType2Audio,
- B_Content_eMultiplex

max_packet_size: maximum PCP packet size.

void DtcpAppLib_SetSourceStreamAttribute(void *hStreamHandle, bool content_has_cci, int content_type, int aps, int ict, int ast);

Stream: Source stream handle.
content_has_cci: flag to indicate if the content has CCI or not.

content_type: type of the content.
Aps*:* analog copy right information.
Ict: image constraint token.
ast :analog sunset token.

The DTCP-IP library treat the media content transparently, it doesn't have capability to probe the content, to obtain CCI (Copy Control Information), and other associated media copy right information(APS, ICT, or AST). Therefore, this API is provided for upper layer application to set the source stream's attributes.
If PCP_UR is to be used for DTCP-IP transmission, then this function must be called after opening the source stream, before starting the media transmission (Before calling "DtcpAppLib_StreamPacketizeData" function).

void DtcpAppLib_SetSourceStreamEmi(void *hStreamHandle, int emi);
hStreamHandle: handle to the sink stream.
emi: value to set.

Set the stream's emi value, if caller wish to override the emi value obtained internally by DTCP lib.
The DTCP-IP lib will choose a proper EMI value based on device mode, content type, and CCI value of the content, if it fails to get a the suitable EMI value, the most strict value "CopyNever" will be used for content transmission. If the caller has determined the proper EMI value to be used, it can then call above function, after DtcpAppLib_OpenSourceStream() function, to override the EMI value assigned by DTCP-IP lib.

int DtcpAppLib_GetSinkStreamEmi(void *hStreamHandle);
hStreamHandle: Sink stream handle.

This function is used by sink device, to check the current stream's EMI value sent from source device. For example, if you are implementing a DVR, and the received stream's EMI value is set to "No-More-Copy", then you may not recording this stream.

Similarly, sink device needs to call following function to open a sink stream:

void * DtcpAppLib_OpenSinkStream(void * aAkeHandle, B_StreamTransport_T Transport_Type);

aAkeHandle: DTCP-IP AKE session handle obtained from DtcpAppLib_DoAke().
Type: Sink stream transport type, currently only B_StreamTransport_eHTTP is supported.

The functions will return NULL if it failed, otherwise a stream handle is returned, application needs to save the handle for later when it tries to packetize/de-packetize data.

After streaming finished, application need to call following function to close the stream:

void DtcpAppLib_CloseStream(void * hStreamHandle);

hStreamHandle: Stream handle obtained from DtcpAppLib_OpenSourceStream() or DtcpAppLib_OpenSinkStream() call.

The server (Source device) application calls packetize data function to packetize the media data into DTCP packet and transmit it out.

BERR_Code DtcpAppLib_StreamPacketizeData(void * hStreamHandle,
void * hAkeHandle,
unsigned char * clear_buf,
unsigned int clear_buf_size,
unsigned char * encrypted_buf,
unsigned int * encrypted_buf_size,
unsigned int * total);

hStreamHandle: Source stream handle obtained from DtcpAppLib_OpenSourceStream().
hAkeHandle: DTCP-IP AKE session handle.
clear_buf: pointer to the buffer where the clear contents are stored.
clear_buf_size: size of the clear buffer.
encrypted_buf: pointer to the buffer where the encrypted content will be stored.
encrypted_buf_size: maximum size of the encrypted buffer.
total: Total bytes of clear data that has been processed(encrypte).

NOTE4: DTCP-IP content encryption algorithm requires the clear data to be multiple of 16 bytes, if you passed in clear data that is not 16 bytes multiple, instead of padding, the DTCP-IP lib will trim it down to 16B aligned. So the returned "total" will be less then clear_buf_size. You will have some "left-over" bytes, that need to be taken care of by the caller.
The function could also add "PCP" header into encrypted buffer, depending on the PCP packet size, and how many clear buffer has been processed. The PCP header size is 14 bytes, so you should allocate extra space to accommodate for this.

Please refer to "***ip_streamer.c***" file (in /nexus/lib/playback_ip/apps/) for the pacing logic for packetizing DTCP data.

Client device (Sink device) call the de-packetize data function to decrypt the content and feed into playback component:

BERR_Code DtcpAppLib_StreamDepacketizeData(void * hStreamHandle,
void * hAkeHandle,
unsigned char * encrypted_buf,
unsigned int encrypted_buf_size,
unsigned char * clear_buf,
unsigned int * clear_buf_size,
unsigned int * total,

bool *pcp_header_found);

hStreamHandle: Sink stream handle.
hAkeHandle: DTCP-IP AKE session handle.
encrypted_buf: pointer to the buffer where the encrypted content are stored(input data).
encrypted_buf_size: size of the input buffer.
clear_buf: pointer to the buffer where the decrypted data will store (output data).
clear_buf_size: maximum size of the clear buffer. Upon return, the value indicates the actual size of the clear media content that has been decrypted.
total: Total size of the encrypted buffer that has been processed in this function call.
pcp_header_fund: A Boolean value indicates if a PCP header is fund in this function call.

NOTE5: If there is a PCP header in the input data, this function will strip out the PCP header, therefore, the output data only contains the decrypted media content.

NOTE6: For client example application, the depacketize data function is not used in the example client source file, instead it was called inside the playback_ip library. However, user may choose to implement their own streaming function by calling the DTCP-IP de-packetize data function.

6 Third party production key support

The DTCP-IP library default build supports Broadcom specific key file format (DTCP_IP_DATA_BRCM=y is exported when building library). However, a customer with its proprietary DTCP-IP production key handling scheme can modify **DRM_Dtcp_Ip_Initialize()** API, defined in "BSEAV/lib/security/third_party/dtcp_ip/drm_dtcp_ip_vendor.c" to map the DTCP-IP keys to corresponding data structure. If third party key handling scheme will be used, please export DTCP_IP_DATA_BRCM=n when building the library.