Xilinx Standalone Library Documentation

XilSecure Library v3.2

UG1189 (2018.3) November 14, 2018

Table of Contents

Chapter 1

Overview

The XilSecure library provides APIs to access secure hardware on the Zynq® UltraScale+™ MPSoC devices and also provides a software implementation for SHA-2 hash generation.
The XilSecure library includes:

- SHA-3/384 engine for 384 bit hash calculation
- AES engine for symmetric key encryption and decryption
- RSA engine for signature generation, signature verification, encryption and decryption.

  Note

  The above libraries are grouped into the Configuration and Security Unit (CSU) on the Zynq UltraScale+ MPSoC device.

- SHA-2/256 algorithm for calculating 256 bit hash

  Note

  The SHA-2 hash generation is a software algorithm which generates SHA2 hash on provided data. SHA-2 support is deprecated. Use SHA-3 instead of SHA-2.

Source Files

The following is a list of source files shipped as a part of the XilSecure library:

- `xsecure_hw.h`: This file contains the hardware interface for all the three modules.
- `xsecure_sha.h`: This file contains the driver interface for SHA-3 module.
- `xsecure_sha.c`: This file contains the implementation of the driver interface for SHA-3 module.
- `xsecure_rsa.h`: This file contains the driver interface for RSA module.
- `xsecure_rsa.c`: This file contains the implementation of the driver interface for RSA module.
- `xsecure_aes.h`: This file contains the driver interface for AES module.
- `xsecure_aes.c` : This file contains the implementation of the driver interface for AES module
- `xsecure_sha2.h`: This file contains the interface for SHA2 hash algorithm.

- xsecure_sha2_a53_32b.a: Pre-compiled file which has SHA2 implementation for A53 32bit.
- xsecure_sha2_a53_64b.a: Pre-compiled file which has SHA2 implementation for A53 64 bit.
- xsecure_sha2_a53_r5.a: Pre-compiled file which has SHA2 implementation for r5.
- xsecure_sha2_pmu.a: Pre-compiled file which has SHA2 implementation for PMU.

XILINX

Chapter 2

AES-GCM

Overview

This block uses AES-GCM algorithm to encrypt or decrypt the provided data. It requires a key of size 256 bits and initialization vector(IV) of size 96 bits.
XilSecure library supports the following features:

- Encryption of data with provided key and IV
- Decryption of data with provided key and IV
- Decryption of Zynq® Ultrascale+™ MPSoC boot image partition, where boot image is generated using `bootgen`.
    - Support for Key rolling
    - Operational key support
- Authentication using GCM tag.
- Key loading based on key selection, key can be either the user provided key loaded into the KUP key or the device key used in the boot.

For either encryption or decryption AES should be initialized first, the XSecure_AesInitialize() API initializes the AES's instance with provided parameters as described.

AES Encryption Function Usage

When all the data to be encrypted is available, the `XSecure_AesEncryptData()` can be used with appropriate parameters as described. When all the data is not available use the following functions in following order.

1. `XSecure_AesEncryptInit()`
2. `XSecure_AesEncryptUpdate()` - This API can be called multiple times till input data is completed.

AES Decryption Function Usage

When all the data to be decrypted is available, the XSecure_AesDecryptData() can be used with appropriate parameters as described. When all the data is not available use the following functions in following order.

1. `XSecure_AesDecryptInit()`

2. `XSecure_AesDecryptUpdate()` - This API can be called multiple times till input data is completed.

The GCM-TAG matching will also be verified and appropriate status will be returned.

> **WARNING:** *When using the KUP key for encryption/decryption of the data, you should take care of the key storage. Key should be placed in a secure memory region for example, internal memory. Not doing so may result in security vulnerability.*

Modules

- AES-GCM API Example Usage

Functions

- s32 XSecure_AesInitialize (XSecure_Aes ∗InstancePtr, XCsuDma ∗CsuDmaPtr, u32 KeySel, u32 ∗Iv, u32 ∗Key)
- void XSecure_AesDecryptInit (XSecure_Aes ∗InstancePtr, u8 ∗DecData, u32 Size, u8 ∗GcmTagAddr)
- s32 XSecure_AesDecryptUpdate (XSecure_Aes ∗InstancePtr, u8 ∗EncData, u32 Size)
- s32 XSecure_AesDecryptData (XSecure_Aes ∗InstancePtr, u8 ∗DecData, u8 ∗EncData, u32 Size, u8 ∗GcmTagAddr)
- s32 XSecure_AesDecrypt (XSecure_Aes ∗InstancePtr, u8 ∗Dst, const u8 ∗Src, u32 Length)
- void XSecure_AesEncryptInit (XSecure_Aes ∗InstancePtr, u8 ∗EncData, u32 Size)
- void XSecure_AesEncryptUpdate (XSecure_Aes ∗InstancePtr, const u8 ∗Data, u32 Size)
- void XSecure_AesEncryptData (XSecure_Aes ∗InstancePtr, u8 ∗Dst, const u8 ∗Src, u32 Len)
- void XSecure_AesReset (XSecure_Aes ∗InstancePtr)
- void XSecure_AesWaitForDone (XSecure_Aes ∗InstancePtr)

Function Documentation

s32 XSecure_AesInitialize (XSecure_Aes ∗ *InstancePtr*, XCsuDma ∗ *CsuDmaPtr*, u32 *KeySel*, u32 ∗ *Iv*, u32 ∗ *Key*)

This function initializes the instance pointer.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
CsuDmaPtr	Pointer to the XCsuDma instance.
KeySel	Key source for decryption, can be KUP/device key • XSECURE_CSU_AES_KEY_SRC_KUP :For KUP key • XSECURE_CSU_AES_KEY_SRC_DEV :For Device Key
Iv	Pointer to the Initialization Vector for decryption
Key	Pointer to Aes decryption key in case KUP key is used. Passes `Null` if device key is to be used.

Returns

XST_SUCCESS if initialization was successful.

Note

All the inputs are accepted in little endian format, but AES engine accepts the data in big endianess, this will be taken care while passing data to AES engine.

void XSecure_AesDecryptInit (XSecure_Aes * *InstancePtr*, u8 * *DecData*, u32 *Size*, u8 * *GcmTagAddr*)

This function initializes the AES engine for decryption.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
DecData	Pointer in which decrypted data will be stored.
Size	Expected size of the data in bytes.
GcmTagAddr	Pointer to the GCM tag which needs to be verified during decryption of the data.

Returns

None

Note

If data is encrypted using XSecure_AesEncrypt API then GCM tag address will be at the end of encrypted data. EncData + Size will be the GCM tag address. Chunking will not be handled over here.

s32 XSecure_AesDecryptUpdate (XSecure_Aes * *InstancePtr*, u8 * *EncData*, u32 *Size*)

This function is used to update the AES engine for decryption with provided data.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
EncData	Pointer to the encrypted data which needs to be decrypted.
Size	Expected size of data to be decrypted in bytes.

Returns

Final call of this API returns the status of GCM tag matching.

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag is mismatched
- XST_SUCCESS: If GCM tag is matching.

Note

When Size of the data equals to size of the remaining data that data will be treated as final data. This API can be called multpile times but sum of all Sizes should be equal to Size mention in init. Return of the final call of this API tells whether GCM tag is matching or not.

s32 XSecure_AesDecryptData (XSecure_Aes * *InstancePtr*, u8 * *DecData*, u8 * *EncData*, u32 *Size*, u8 * *GcmTagAddr*)

This function decrypts the encrypted data provided and updates the DecData buffer with decrypted data.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
DecData	Pointer to a buffer in which decrypted data will be stored.
EncData	Pointer to the encrypted data which needs to be decrypted.
Size	Size of data to be decrypted in bytes.

Returns

This API returns the status of GCM tag matching.

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag was mismatched
- XST_SUCCESS: If GCM tag was matched.

Note

When XSecure_AesEncryptData() API is used for encryption In same buffer GCM tag also be stored, but Size should be mentioned only for data.

s32 XSecure_AesDecrypt (XSecure_Aes * *InstancePtr,* u8 * *Dst,* const u8 * *Src,* u32 *Length*)

This function will handle the AES-GCM Decryption.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
Src	Pointer to encrypted data source location
Dst	Pointer to location where decrypted data will be written.
Length	Expected total length of decrypted image expected.

Returns

returns XST_SUCCESS if successful, or the relevant errorcode.

Note

This function is used for decrypting the Image's partition encrypted by Bootgen

void XSecure_AesEncryptInit (XSecure_Aes * *InstancePtr,* u8 * *EncData,* u32 *Size*)

This funcion is used to initialize the AES engine for encryption.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
EncData	Pointer of a buffer in which encrypted data along with GCM TAG will be stored. Buffer size should be Size of data plus 16 bytes.
Size	A 32 bit variable, which holds the size of the input data to be encrypted.

Returns

None

Note

If all the data to be encrypted is available at single location One can use XSecure_AesEncryptData() directly.

void XSecure_AesEncryptUpdate (XSecure_Aes * *InstancePtr,* const u8 * *Data,* u32 *Size*)

This function is used to update the AES engine with provided data for encryption.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.
Data	Pointer to the data for which encryption should be performed.
Size	A 32 bit variable, which holds the size of the input data in bytes.

Returns

None

Note

When Size of the data equals to size of the remaining data to be processed that data will be treated as final data. This API can be called multpile times but sum of all Sizes should be equal to Size mentioned at encryption initialization (XSecure_AesEncryptInit()). If all the data to be encrypted is available at single location Please call XSecure_AesEncryptData() directly.

void XSecure_AesEncryptData (XSecure_Aes * *InstancePtr*, u8 * *Dst*, const u8 * *Src*, u32 *Len*)

This Function encrypts the data provided by using hardware AES engine.

Parameters

InstancePtr	A pointer to the XSecure_Aes instance.
Dst	A pointer to a buffer where encrypted data along with GCM tag will be stored. The Size of buffer provided should be Size of the data plus 16 bytes
Src	A pointer to input data for encryption.
Len	Size of input data in bytes

Returns

None

Note

If data to be encrypted is not available at one place one can call XSecure_AesEncryptInit() and update the AES engine with data to be encrypted by calling XSecure_AesEncryptUpdate() API multiple times as required.

void XSecure_AesReset (XSecure_Aes * *InstancePtr*)

This function resets the AES engine.

Parameters

InstancePtr	is a pointer to the XSecure_Aes instance.

Returns

None

void XSecure_AesWaitForDone (XSecure_Aes * *InstancePtr*)

This function waits for AES completion.

Parameters

InstancePtr	Pointer to the XSecure_Aes instance.

Returns

None

AES-GCM API Example Usage

The `xilsecure_aes_example.c` file illustrates AES usage with decryption of a Zynq® UltraScale+™ MPSoC boot image placed at a predefined location in memory. You can select the key type (device key or user-selected KUP key). The example assumes that the boot image is present at `0x0400000` (DDR); consequently, the image must be loaded at that address through JTAG. The example decrypts the boot image and returns `XST_SUCCESS` or `XST_FAILURE` based on whether the GCM tag was successfully matched.
The Multiple key(Key Rolling) or Single key encrypted images will have the same format. The images include:

- Secure header - This includes the Dummy AES Key of 32byte + Block 0 IV of 12byte + DLC for Block 0 of 4byte + GCM tag of 16byte(Un-Enc).
- Block N - This includes the Boot Image Data for Block N of n size + Block N+1 AES key of 32byte + Block N+1 IV of 12byte + GCM tag for Block N of 16byte(Un-Enc).

The Secure header and Block 0 will be decrypted using Device key or user provided key. If more than one block is found then the key and IV obtained from previous block will be used for decryption.
Following are the instructions to decrypt an image:

1. Read the first 64 bytes and decrypt 48 bytes using the selected Device key.
2. Decrypt Block 0 using the IV + Size and the selected Device key.
3. After decryption, you will get the decrypted data+KEY+IV+Block Size. Store the KEY/IV into KUP/IV registers.
4. Using Block size, IV and the next Block key information, start decrypting the next block.

5. If the current image size is greater than the total image length, perform the next step. Else, go back to the previous step.

6. If there are failures, an error code is returned. Else, the decryption is successful.

The following is a snippet from the `xilsecure_aes_example.c` file.

```
int SecureAesExample(void)
{
  u8 *Dst = (u8 *)(UINTPTR)DestinationAddr;
  XCsuDma_Config *Config;

  int Status;
  u32 PhOffset;
  u32 PartitionOffset;
  u32 PartitionLen;

  Config = XCsuDma_LookupConfig(0);
  if (NULL == Config) {
    xil_printf("config  failed \n\r");
    return XST_FAILURE;
  }

  Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  /* Read partition offset and length from partition header */
  PhOffset = XSecure_In32((UINTPTR)(ImageOffset +
        XSECURE_BHDR_PH_OFFSET));
  PartitionOffset = (XSecure_In32((UINTPTR)(ImageOffset + PhOffset +
      XSECURE_PH_PARTITION_OFFSET)) * XSECURE_WORD_MUL);
  PartitionLen = (XSecure_In32((UINTPTR)(ImageOffset + PhOffset +
      XSECURE_PH_PARTITION_LEN_OFFSET)) * XSECURE_WORD_MUL);

  *(csu_iv + XSECURE_IV_INDEX) =  (*(csu_iv + XSECURE_IV_INDEX)) +
      (XSecure_In32((UINTPTR)(ImageOffset + PhOffset +
        XSECURE_PH_PARTITION_IV_OFFSET)) &
        XSECURE_PH_PARTITION_IV_MASK);
  /*
   * Initialize the Aes driver so that it's ready to use
   */
  XSecure_AesInitialize(&Secure_Aes, &CsuDma, XSECURE_CSU_AES_KEY_SRC_KUP,
                                (u32 *)csu_iv, (u32 *)csu_key);

  Status = XSecure_AesDecrypt(&Secure_Aes, Dst,
      (u8 *)(UINTPTR)(ImageOffset + PartitionOffset),
            PartitionLen);


  if(Status != XST_SUCCESS)
  {
    return XST_FAILURE;
  }

  return XST_SUCCESS;
}
```

Note

The `xilsecure_aes_example.c` example file is available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

The following example illustrates the usage of AES encryption and decryption APIs.

```
static s32 SecureAesExample(void)
{
  XCsuDma_Config *Config;
  s32 Status;
  u32 Index;
  u8 DecData[XSECURE_DATA_SIZE]__attribute__ ((aligned (64)));
  u8 EncData[XSECURE_DATA_SIZE + XSECURE_SECURE_GCM_TAG_SIZE]
          __attribute__ ((aligned (64)));

  /* Initialize CSU DMA driver */
  Config = XCsuDma_LookupConfig(XSECURE_CSUDMA_DEVICEID);
  if (NULL == Config) {
    return XST_FAILURE;
  }

  Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  /* Initialize the Aes driver so that it's ready to use */
  XSecure_AesInitialize(&Secure_Aes, &CsuDma,
        XSECURE_CSU_AES_KEY_SRC_KUP,
        (u32 *)Iv, (u32 *)Key);

  xil_printf("Data to be encrypted: \n\r");
  for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
    xil_printf("%02x", Data[Index]);
  }
  xil_printf( "\r\n\n");

  /* Encryption of Data */
  /*
   * If all the data to be encrypted is contiguous one can call
   * XSecure_AesEncryptData API directly.
   */
  XSecure_AesEncryptInit(&Secure_Aes, EncData, XSECURE_DATA_SIZE);
  XSecure_AesEncryptUpdate(&Secure_Aes, Data, XSECURE_DATA_SIZE);

  xil_printf("Encrypted data: \n\r");
  for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
    xil_printf("%02x", EncData[Index]);
  }
  xil_printf( "\r\n");

  xil_printf("GCM tag: \n\r");
  for (Index = 0; Index < XSECURE_SECURE_GCM_TAG_SIZE; Index++) {
    xil_printf("%02x", EncData[XSECURE_DATA_SIZE + Index]);
  }
  xil_printf( "\r\n\n");

  /* Decrypt's the encrypted data */
  /*
   * If data to be decrypted is contiguous one can also call
   * single API XSecure_AesDecryptData
   */
  XSecure_AesDecryptInit(&Secure_Aes, DecData, XSECURE_DATA_SIZE,
          EncData + XSECURE_DATA_SIZE);
  /* Only the last update will return the GCM TAG matching status */
  Status = XSecure_AesDecryptUpdate(&Secure_Aes, EncData,
             XSECURE_DATA_SIZE);
  if (Status != XST_SUCCESS) {
    xil_printf("Decryption failure- GCM tag was not matched\n\r");
    return Status;
  }

  xil_printf("Decrypted data\n\r");
```

```
  for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
    xil_printf("%02x", DecData[Index]);
  }
  xil_printf( "\r\n");

  /* Comparison of Decrypted Data with original data */
  for(Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
    if (Data[Index] != DecData[Index]) {
      xil_printf("Failure during comparison of the data\n\r");
      return XST_FAILURE;
    }
  }

  return XST_SUCCESS;
}
```

Chapter 3

RSA

Overview

The xsecure_rsa.h file contains hardware interface related information for RSA hardware accelerator. This block performs the modulus math based on Rivest-Shamir-Adelman (RSA)-4096 algorithm. It is an asymmetric algorithm.

Initialization & Configuration

The Rsa driver instance can be initialized by using the `XSecure_RsaInitialize()` function.
The method used for RSA needs pre-calculated value of `R^2 mod N`, which is generated by bootgen and is present in the signature along with modulus and exponent. If you do not have the pre-calculated exponential value pass NULL, the controller will take care of exponential value.

Note

- From RSA key modulus, exponent should be extracted. If image is created using bootgen all the fields are available in the boot image.
- For matching, PKCS v1.5 padding scheme has to be applied in the manner while comparing the data hash with decrypted hash.

Modules

- RSA API Example Usage

Functions

- s32 XSecure_RsaInitialize (XSecure_Rsa *InstancePtr, u8 *Mod, u8 *ModExt, u8 *ModExpo)
- s32 XSecure_RsaDecrypt (XSecure_Rsa *InstancePtr, u8 *EncText, u8 *Result)
- u32 XSecure_RsaSignVerification (u8 *Signature, u8 *Hash, u32 HashLen)
- s32 XSecure_RsaPublicEncrypt (XSecure_Rsa *InstancePtr, u8 *Input, u32 Size, u8 *Result)
- s32 XSecure_RsaPrivateDecrypt (XSecure_Rsa *InstancePtr, u8 *Input, u32 Size, u8 *Result)

Function Documentation

s32 XSecure_RsaInitialize (XSecure_Rsa * *InstancePtr,* u8 * *Mod,* u8 * *ModExt,* u8 * *ModExpo*)

This function initializes a specific Xsecure_Rsa instance so that it is ready to be used.

Parameters

InstancePtr	Pointer to the XSecure_Rsa instance.
Mod	A character Pointer which contains the key Modulus of key size.
ModExt	A Pointer to the pre-calculated exponential (R^2 Mod N) value. • NULL - if user doesn't have pre-calculated R^2 Mod N value, control will take care of this calculation internally.
ModExpo	Pointer to the buffer which contains key exponent.

Returns

XST_SUCCESS if initialization was successful.

Note

`Modulus`, `ModExt` and `ModExpo` are part of prtition signature when authenticated boot image is generated by bootgen, else the all of them should be extracted from the key.

s32 XSecure_RsaDecrypt (XSecure_Rsa * *InstancePtr,* u8 * *EncText,* u8 * *Result*)

This function handles the RSA decryption from end to end.

Parameters

InstancePtr	Pointer to the XSecure_Rsa instance.
EncText	Pointer to the buffer which contains the input data to be decrypted.
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

XST_SUCCESS if decryption was successful.

Note

This API will be deprecated soon. Instead of this please use XSecure_RsaPublicEncrypt() API. This API can only support 4096 key Size.

u32 XSecure_RsaSignVerification (u8 * *Signature*, u8 * *Hash*, u32 *HashLen*)

This function verifies the RSA decrypted data provided is either matching with the provided expected hash by taking care of PKCS padding.

Parameters

Signature	Pointer to the buffer which holds the decrypted RSA signature
Hash	Pointer to the buffer which has hash calculated on the data to be authenticated.
HashLen	Length of Hash used. • For SHA3 it should be 48 bytes • For SHA2 it should be 32 bytes

Returns

XST_SUCCESS if decryption was successful.

s32 XSecure_RsaPublicEncrypt (XSecure_Rsa * *InstancePtr*, u8 * *Input*, u32 *Size*, u8 * *Result*)

This function handles the RSA signature encryption with public key components provide at XSecure_RsaInitialize() API.

Parameters

InstancePtr	Pointer to the XSecure_Rsa instance.
Input	Pointer to the buffer which contains the input data to be decrypted.
Size	Key size in bytes, Input size also should be same as Key size mentioned.Inputs supported are • XSECURE_RSA_4096_KEY_SIZE and • XSECURE_RSA_2048_KEY_SIZE
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

XST_SUCCESS if encryption was successful.

Note

Modulus of API XSecure_RsaInitialize() should also be same size of key size mentioned in this API and exponent should be 32 bit size.

s32 XSecure_RsaPrivateDecrypt (XSecure_Rsa * *InstancePtr*, u8 * *Input*, u32 *Size*, u8 * *Result*)

This function handles the RSA signature decryption with private key components provide at XSecure_RsaInitialize() API.

Parameters

InstancePtr	Pointer to the XSecure_Rsa instance.
Input	Pointer to the buffer which contains the input data to be decrypted.
Size	Key size in bytes, Input size also should be same as Key size mentioned. Inputs supported are XSECURE_RSA_4096_KEY_SIZE and XSECURE_RSA_2048_KEY_SIZE*
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

XST_SUCCESS if decryption was successful. else returns an error code

- XSECURE_RSA_DATA_VALUE_ERROR - if input data is greater than modulus
    - XST_FAILURE - on RSA operation failure

Note

Modulus and Exponent in XSecure_RsaInitialize() API should also be same as key size mentioned in this API.

RSA API Example Usage

The `xilsecure_rsa_example.c` file illustrates the usage of XilSecure APIs by authenticating the FSBL partition of the Zynq® UltraScale+™ MPSoC boot image. The boot image signature is encrypted using RSA-4096 algorithm. Resulting digest is matched with SHA3 Hash calculated on the FSBL.
The authenticated boot image should be loaded in memory through JTAG and address of the boot image should be passed to the function. By default, the example assumes that the authenticated image is present at location 0x04000000 (DDR), which can be changed as required.
The following is a snippet from the `xilsecure_rsa_example.c` file.

```
u32 SecureRsaExample(void)
{
  u32 Status;

  /*
   * Download the boot image elf at a DDR location, Read the boot header
   * assign Src pointer to the location of FSBL image in it. Ensure
   * that linker script does not map the example elf to the same
   * location as this standalone example
   */
  u32 FsblOffset = XSecure_In32((UINTPTR)(ImageOffset + HeaderSrcOffset));

  xil_printf(" Fsbl Offset in the image is  %0x ",FsblOffset);
  xil_printf(" \r\n ");
```

```
u32 FsblLocation = ImageOffset + FsblOffset;

xil_printf(" Fsbl Location is %0x ",FsblLocation);
xil_printf(" \r\n ");

u32 TotalFsblLength = XSecure_In32((UINTPTR)(ImageOffset +
        HeaderFsblTotalLenOffset));

u32 AcLocation = FsblLocation + TotalFsblLength - XSECURE_AUTH_CERT_MIN_SIZE;

xil_printf(" Authentication Certificate Location is %0x ",AcLocation);
xil_printf(" \r\n ");

u8 BIHash[XSECURE_HASH_TYPE_SHA3] __attribute__ ((aligned (4)));
u8 * SpkModular = (u8 *)XNULL;
u8 * SpkModularEx = (u8 *)XNULL;
u32 SpkExp = 0;
u8 * AcPtr = (u8 *)(UINTPTR)AcLocation;
u32 ErrorCode = XST_SUCCESS;
u32 FsblTotalLen = TotalFsblLength - XSECURE_FSBL_SIG_SIZE;

xil_printf(" Fsbl Total Length(Total - BI Signature) %0x ",
    (u32)FsblTotalLen);
xil_printf(" \r\n ");

AcPtr += (XSECURE_RSA_AC_ALIGN + XSECURE_PPK_SIZE);
SpkModular = (u8 *)AcPtr;
AcPtr += XSECURE_FSBL_SIG_SIZE;
SpkModularEx = (u8 *)AcPtr;
AcPtr += XSECURE_FSBL_SIG_SIZE;
SpkExp = *((u32 *)AcPtr);
AcPtr += XSECURE_RSA_AC_ALIGN;

AcPtr += (XSECURE_SPK_SIG_SIZE + XSECURE_BHDR_SIG_SIZE);
xil_printf(" Boot Image Signature Location is %0x ",(u32)(UINTPTR)AcPtr);
xil_printf(" \r\n ");

/*
 * Set up CSU DMA instance for SHA-3 transfers
 */
XCsuDma_Config *Config;

Config = XCsuDma_LookupConfig(0);
if (NULL == Config) {
  xil_printf("config  failed\n\r");
  return XST_FAILURE;
}

Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
if (Status != XST_SUCCESS) {
  return XST_FAILURE;
}

/*
 * Initialize the SHA-3 driver so that it's ready to use
 * Look up the configuration in the config table and then initialize it.
 */

XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);
/* As we are authenticating FSBL here SHA3 KECCAK should be used */
Status = XSecure_Sha3PadSelection(&Secure_Sha3, XSECURE_CSU_KECCAK_SHA3);
if (Status != XST_SUCCESS) {
  goto ENDF;
}
XSecure_Sha3Start(&Secure_Sha3);

XSecure_Sha3Update(&Secure_Sha3, (u8 *)(UINTPTR)FsblLocation,
        FsblTotalLen);
```

```
  XSecure_Sha3Finish(&Secure_Sha3, (u8 *)BIHash);

  /*
   * Initialize the Rsa driver so that it's ready to use
   * Look up the configuration in the config table and then initialize it.
   */
  XSecure_RsaInitialize(&Secure_Rsa, SpkModular, SpkModularEx,
          (u8 *)&SpkExp);

  /*
   * Decrypt Boot Image Signature.
   */
  if(XST_SUCCESS != XSecure_RsaDecrypt(&Secure_Rsa, AcPtr,
            XSecure_RsaSha3Array))
  {
    ErrorCode = XSECURE_IMAGE_VERIF_ERROR;
    goto ENDF;
  }

  xil_printf("\r\n Calculated Boot image Hash \r\n ");
  int i= 0;
  for(i=0; i < 384/8; i++)
  {
    xil_printf(" %0x ", BIHash[i]);
  }
  xil_printf(" \r\n ");

  xil_printf("\r\n Hash From Signature \r\n ");
  int ii= 128;
  for(ii = 464; ii < 512; ii++)
  {
    xil_printf(" %0x ", XSecure_RsaSha3Array[ii]);
  }
  xil_printf(" \r\n ");

  /*
   * Authenticate FSBL Signature.
   */
  if(XSecure_RsaSignVerification(XSecure_RsaSha3Array, BIHash,
                         XSECURE_HASH_TYPE_SHA3) != 0)
  {
    ErrorCode = XSECURE_IMAGE_VERIF_ERROR;
  }

ENDF:
  return ErrorCode;
}
```

Note

The xilsecure_rsa_example.c example file is available in the <library-install-path>\examples folder. Where <library-install-path> is the XilSecure library installation path.

The following example illustrates the usage of RSA APIs to encrypt the data using public key and to decrypt the data using private key.

Note

Application should take care of the padding.

```
u32 SecureRsaExample(void)
{
  u32 Index;
```

```
  /* RSA signature decrypt with private key */
  /*
   * Initialize the Rsa driver with private key components
   * so that it's ready to use
   */
  XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, PrivateExp);


  if(XST_SUCCESS != XSecure_RsaPrivateDecrypt(&Secure_Rsa, Data,
            Size, Signature)) {
    xil_printf("Failed at RSA signature decryption\n\r");
    return XST_FAILURE;
  }

  xil_printf("\r\n Decrypted Signature with private key\r\n ");

  for(Index = 0; Index < Size; Index++) {
    xil_printf(" %02x ", Signature[Index]);
  }
  xil_printf(" \r\n ");

  /* Verification if Data is expected */
  for(Index = 0; Index < Size; Index++) {
    if (Signature[Index] != ExpectedSign[Index]) {
      xil_printf("\r\nError at verification of RSA signature"
          " Decryption\n\r");
      return XST_FAILURE;
    }
  }

  /* RSA signature encrypt with Public key components */

  /*
   * Initialize the Rsa driver with public key components
   * so that it's ready to use
   */

  XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, (u8 *)&PublicExp);

  if(XST_SUCCESS != XSecure_RsaPublicEncrypt(&Secure_Rsa, Signature,
            Size, EncryptSignatureOut)) {
    xil_printf("\r\nFailed at RSA signature encryption\n\r");
    return XST_FAILURE;
  }
  xil_printf("\r\n Encrypted Signature with public key\r\n ");

  for(Index = 0; Index < Size; Index++) {
    xil_printf(" %02x ", EncryptSignatureOut[Index]);
  }

  /* Verification if Data is expected */
  for(Index = 0; Index < Size; Index++) {
    if (EncryptSignatureOut[Index] != Data[Index]) {
      xil_printf("\r\nError at verification of RSA signature"
          " encryption\n\r");
      return XST_FAILURE;
    }
  }


  return XST_SUCCESS;
}
```

Chapter 4

SHA-3

Overview

This block uses the NIST-approved SHA-3 algorithm to generate 384 bit hash on the input data. Because the SHA-3 hardware only accepts 104 byte blocks as minimum input size, the input data is padded with a 10∗1 sequence to complete the final byte block. The padding is handled internally by the driver API.

Note

By default, the library calculates hash using NIST SHA3 padding, but supports choosing KECCAK SHA3 padding.

Initialization & Configuration

The SHA-3 driver instance can be initialized using the `XSecure_Sha3Initialize()` function.
A pointer to CsuDma instance has to be passed in initialization as CSU DMA will be used for data transfers to SHA module.

SHA-3 Functions Usage

When all the data is available on which sha3 hash must be calculated, the `XSecure_Sha3Digest()` can be used with appropriate parameters, as described. When all the data is not available on which sha3 hash must be calculated, use the sha3 functions in the following order:

1. `XSecure_Sha3Start()`
2. `XSecure_Sha3Update()` - This API can be called multiple times till input data is completed.
3. `XSecure_Sha3Finish()` - Provides the final hash of the data. To get intermediate hash values after each `XSecure_Sha3Update()`, you can call `XSecure_Sha3_ReadHash()` after the `XSecure_Sha3Update()` call.

Modules

- SHA-3 API Example Usage

Functions

- s32 XSecure_Sha3Initialize (XSecure_Sha3 *InstancePtr, XCsuDma *CsuDmaPtr)
- void XSecure_Sha3Start (XSecure_Sha3 *InstancePtr)
- void XSecure_Sha3Update (XSecure_Sha3 *InstancePtr, const u8 *Data, const u32 Size)
- void XSecure_Sha3Finish (XSecure_Sha3 *InstancePtr, u8 *Hash)
- void XSecure_Sha3Digest (XSecure_Sha3 *InstancePtr, const u8 *In, const u32 Size, u8 *Out)
- void XSecure_Sha3_ReadHash (XSecure_Sha3 *InstancePtr, u8 *Hash)
- s32 XSecure_Sha3PadSelection (XSecure_Sha3 *InstancePtr, XSecure_Sha3PadType Sha3Type)

Function Documentation

s32 XSecure_Sha3Initialize (XSecure_Sha3 * *InstancePtr*, XCsuDma * *CsuDmaPtr*)

This function initializes a specific Xsecure_Sha3 instance so that it is ready to be used.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
CsuDmaPtr	Pointer to the XCsuDma instance.

Returns

XST_SUCCESS if initialization was successful

Note

The base address is initialized directly with value from xsecure_hw.h By default uses NIST SHA3 padding, to change to KECCAK padding call XSecure_Sha3PadSelection() after XSecure_Sha3Initialize().

void XSecure_Sha3Start (XSecure_Sha3 * *InstancePtr*)

This function configures the SSS and starts the SHA-3 engine.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

None

void XSecure_Sha3Update (XSecure_Sha3 * *InstancePtr*, const u8 * *Data*, const u32 *Size*)

This function updates hash for new input data block.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Data	Pointer to the input data for hashing.
Size	Size of the input data in bytes.

Returns

None

void XSecure_Sha3Finish (XSecure_Sha3 * *InstancePtr*, u8 * *Hash*)

This function sends the last data and padding when blocksize is not multiple of 104 bytes.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to location where resulting hash will be written

Returns

None

void XSecure_Sha3Digest (XSecure_Sha3 * *InstancePtr*, const u8 * *In*, const u32 *Size*, u8 * *Out*)

This function calculates the SHA-3 digest on the given input data.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
In	Pointer to the input data for hashing
Size	Size of the input data
Out	Pointer to location where resulting hash will be written.

Returns

None

void XSecure_Sha3_ReadHash (XSecure_Sha3 * *InstancePtr*, u8 * *Hash*)

Reads the SHA3 hash of the data. It can be called intermediately of updates also to read hashs.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to a buffer in which read hash will be stored.

Returns

None

Note

None

s32 XSecure_Sha3PadSelection (XSecure_Sha3 * *InstancePtr*, XSecure_Sha3PadType *Sha3Type*)

This function provides an option to select the SHA-3 padding type to be used while calculating the hash.

Parameters

InstancePtr	Pointer to the XSecure_Sha3 instance.
Sha3Type	Type of the sha3 padding to be used. • For NIST SHA-3 padding - XSECURE_CSU_NIST_SHA3 • For KECCAK SHA-3 padding - XSECURE_CSU_KECCAK_SHA3

Returns

By default provides support for NIST SHA-3, if wants to change for Keccak SHA-3 this function should be called after XSecure_Sha3Initialize()

SHA-3 API Example Usage

The `xilsecure_sha_example.c` file is a simple example application that demonstrates the usage of SHA-3 device to calculate 384 bit hash on Hello World string. A more typical use case of calculating the hash of boot image as a step in authentication process using the SHA-3 device has been illustrated in the `xilsecure_rsa_example.c`.
The contents of the `xilsecure_sha_example.c` file are shown below:

```
int SecureHelloWorldExample()
{
  u8 HelloWorld[4] = {'h','e','l','l'};
  u32 Size = sizeof(HelloWorld);
  u8 Out[384/8];
  XCsuDma_Config *Config;

  int Status;

  Config = XCsuDma_LookupConfig(0);
  if (NULL == Config) {
    xil_printf("config  failed\n\r");
    return XST_FAILURE;
  }

  Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
  if (Status != XST_SUCCESS) {
    return XST_FAILURE;
  }

  /*
   * Initialize the SHA-3 driver so that it's ready to use
   */
  XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);

  XSecure_Sha3Digest(&Secure_Sha3, HelloWorld, Size, Out);

  xil_printf(" Calculated Digest \r\n ");
  int i= 0;
  for(i=0; i< (384/8); i++)
  {
    xil_printf(" %0x ", Out[i]);
  }
  xil_printf(" \r\n ");

  return XST_SUCCESS;
}
```

Note

The `xilsecure_sha_example.c` and `xilsecure_rsa_example.c` example files are available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

Chapter 5

SHA-2

Overview

This is an algorithm which generates 256 bit hash on the input data.

Note

SHA-2 will be deprecated from release 2019.1 onwards. Xilinx recommends you to use SHA-3.

SHA-2 Function Usage

When all the data is available on which sha2 hash must be calculated, the `sha_256()` can be used with appropriate parameters, as described. When all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

1. `sha2_starts()`
2. `sha2_update()` - This API can be called multiple times till input data is completed.
3. `sha2_finish()` - Provides the final hash of the data.

To get intermediate hash values after each `sha2_update()`, you can call `sha2_hash()` after the `sha2_update()` call.

Modules

- SHA-2 Example Usage

Functions

- void sha_256 (const unsigned char *in, const unsigned int size, unsigned char *out)
- void sha2_starts (sha2_context *ctx)
- void sha2_update (sha2_context *ctx, unsigned char *input, unsigned int ilen)
- void sha2_finish (sha2_context *ctx, unsigned char *output)
- void sha2_hash (sha2_context *ctx, unsigned char *output)

Function Documentation

void sha_256 (const unsigned char * *in*, const unsigned int *size*, unsigned char * *out*)

This function calculates the hash for the input data using SHA-256 algorithm. This function internally calls the sha2_init, updates and finishes functions and updates the result.

Parameters

In	Char pointer which contains the input data.
Size	Length of the input data
Out	Pointer to location where resulting hash will be written.

Returns

None

void sha2_starts (sha2_context * *ctx*)

This function initializes the SHA2 context.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.

Returns

None

void sha2_update (sha2_context * *ctx*, unsigned char * *input*, unsigned int *ilen*)

This function adds the input data to SHA256 calculation.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
input	Pointer to the data to add.
Out	Length of the input data.

Returns

None

void sha2_finish (sha2_context * *ctx*, unsigned char * *output*)

This function finishes the SHA calculation.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
output	Pointer to the calculated hash data.

Returns

None

void sha2_hash (sha2_context * *ctx*, unsigned char * *output*)

This function reads the SHA2 hash, it can be called intermediately of updates to read the SHA2 hash.

Parameters

ctx	Pointer to sha2_context structure that stores status and buffer.
output	Pointer to the calculated hash data.

Returns

None

SHA-2 Example Usage

The `xilsecure_sha2_example.c` file contains the implementation of the interface functions for SHA driver. When all the data is available on which sha2 must be calculated, the sha_256() function can be used with appropriate parameters, as described. But, when all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

- sha2_update() can be called multiple times till input data is completed.
- sha2_context is updated by the library only; do not change the values of the context.

The contents of the `xilsecure_sha2_example.c` file are shown below:

```
u32 XSecure_Sha2_Hash_Gn()
{
  sha2_context Sha2;
  u8 Output_Hash[32];
  u8 IntermediateHash[32];
  u8 Cal_Hash[32];
  u32 Index;
  u32 Size = XSECURE_DATA_SIZE;
  u32 Status;
```

```
  /* Generating SHA2 hash */
  sha2_starts(&Sha2);
  sha2_update(&Sha2, (u8 *)Data, Size - 1);

  /* If required we can read intermediate hash */
  sha2_hash(&Sha2, IntermediateHash);
  xil_printf("Intermediate SHA2 Hash is: ");
  for (Index = 0; Index < 32; Index++) {
    xil_printf("%02x", IntermediateHash[Index]);
  }
  xil_printf("\n");

  sha2_finish(&Sha2, Output_Hash);

  xil_printf("Generated SHA2 Hash is: ");
  for (Index = 0; Index < 32; Index++) {
    xil_printf("%02x", Output_Hash[Index]);
  }
  xil_printf("\n");

  /* Convert expected Hash value into hexa */
  Status = XSecure_ConvertStringToHexBE(XSECURE_EXPECTED_SHA2_HASH,
              Cal_Hash, 64);
  if (Status != XST_SUCCESS) {
    xil_printf("Error: While converting expected "
        "string of SHA2 hash to hexa\n\r");
    return XST_FAILURE;
  }

  /* Compare generated hash with expected hash value */
  for (Index = 0; Index < 32; Index++) {
    if (Cal_Hash[Index] != Output_Hash[Index]) {
      xil_printf("Error: SHA2 Hash generated through "
      "XilSecure library does not match with "
      "expected hash value\n\r");
      return XST_FAILURE;
    }
  }

  return XST_SUCCESS;
}
```

Note

The `xilsecure_sha2_example.c` example file is available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support .

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Please Read: Important Legal Notices

Automotive Applications Disclaimer

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.