IoT Embedded

Software Development Kit

SDK Version 1.0.2

Document Rev: November 7, 2019

SecureRF Corporation
100 Beard Sawmill Road, Suite 350
Shelton, CT 06484
203-227-3151
support@SecureRF.com

Contents

1. Introduction

IMPORTANT NOTE Refer to the readme.txt file in the SDK route for quick-start directions for your particular SDK.

The IoT Embedded SDK is a collection of pre-compiled C and assembly language routines that you can call from your own code in order to implement SecureRF's Group Theoretic-based cryptographic functions. This document provides instructions and function references that will help expedite your development.

The SDK is delivered as one or more object files and is available for the MSP430, ARM Cortex-M0/M3/M4/A9 and 8051. Compilers supported include TI Code Composer Studio, GCC, IAR, and Keil. Please contact SecureRF if you have other requirements.

Underpinning all SecureRF cryptographic methods is the selection of a hard mathematical problem and a strong one-way function that is easy to compute in one direction but computationally infeasible to reverse. For SecureRF's methods, the selected hard problem is known as the Simultaneous Conjugacy Search Problem that requires solving a set of simultaneous equations over the Braid group. Braids were initially described by E. Artin in 1925, along with a set of generators that described an infinite group. Braids have a rich set of properties making them ideal candidates for energy-saving Group Theoretic Cryptography (GTC).

Any given braid can be used to describe a sequence of mathematical operations (known as E-Multiplication, SecureRF's strong one-way function) to be applied on a set of parallel operands such as a matrix and a permutation. A single braid operation simply describes a twist (either clockwise or anti-clockwise) on an adjacent pair of operands. The SDK, by necessity, contains some related terminology and concepts that may be unfamiliar to you. Good primers on Public-Key cryptosystems, braid theory and SecureRF's GTC-based methods may be found here: http://www.securerf.com/technology/papers/

The SDK provides APIs for the two most widely used cryptographic functions 1. Ironwood™ Key Agreement Protocol, and 2. WalnutDSA™ Digital Signature Algorithm.

Important Note: This document is general for all supported target processors and IDEs. Please see the readme.txt file in your SDK deliverable for instructions specific to your processor.

2. Ironwood: Ironwood Key Agreement Protocol

2.1 Ironwood Theory of Operation

Introduction

This section introduces SecureRF's "Ironwood" Meta Key Agreement and Authentication Protocol. Ironwood is considered a "Meta" key agreement protocol because it has some characteristics of a public key solution and some of a shared-key solution, specifically it does not require a secure key database in order to authenticate endpoints. In a real-world application, two parties are involved in running Ironwood. One side is called the "Home Device" and the other is called "Other Device". Ironwood enables these two endpoints to generate a shared secret over an open channel without any prior communication.

The Home Device is provisioned with a public key, a private key and "T-values" (a collection of non-zero field elements). The Other Device is provisioned with an Ironwood public authentication token and a private authentication token. In general, the Home Device should be that device that is best able to safeguard the private key and T-values. For example, if the two devices consist of a mobile device and a hardened RFID tag, you will likely want to make the tag the Home Device. If run-time is of paramount concern, it may be helpful to know that the Other Device requires 3x less computations than the Home Device.

Either party may be configured to authenticate the other side, or they may be configured to authenticate each other. In most cases, you'll want the public key for the side being challenged to be included in a digital certificate that is signed by a trusted party. The challenger will validate the certificate before moving on to shared secret calculation-portion of the protocol. The Host Device and the Other Device may authenticate each other offline i.e. without any connection to a supporting server.

For most IoT Embedded SDKs, the sample code that ships with it contains two directories: "home_device" and "other_device". In each directory is a main.c file that runs one side or the other. However, for GCC projects, there is only a single makefile, and the method compiled—"Home" or "Other"—is determined by the CONFIG variable in the makefile. The Home version computes a shared secret from static keys. The Other version uses the S-Column generated by the Home side to generate the same shared secret. In real life applications, this S-Column would be transmitted from Home to Other, but in this scenario, we've precomputed the S-Column and hard-coded it into the keys files. Both versions compute a CMAC using the shared secret and a pre-defined nonce. Note, when running the examples, you should run the "Home Device" first, then perform a "make clean" then run the "Other" device last. Also note that for certain SDKs, there may only be a single "main.c" file that contains both Home Device code and Other Device code (with one or the other ifdef'd out). Please see the README notes that accompany every SDK to ascertain which version you have.

Example protocol

Below in Fig. 1. is an example of how Ironwood can be used to authenticate one of the two parties, in this case, the Home Device authenticates the Other Device. This example is included in the SDK sample code. This is just one example of how Ironwood can be used to authenticate another device—feel free to modify the protocol for your own requirements. For example, you can expand the protocol to have both devices mutually authenticate each other. The example provides for replay protection by its use of a nonce (a random number) that is generated for each authentication session (note: in the SDK sample code, the nonce is pre-generated, but in a production applications, you would use a TRNG to generate the nonce on the fly). Once the Other device is authenticated, a derivative of the shared secret may be used to encrypt/decrypt messages that are subsequently passed between the two devices.

The example protocol also employs the use of SecureRF's WalnutDSA digital signature algorithm that is discussed in Section 3. below.

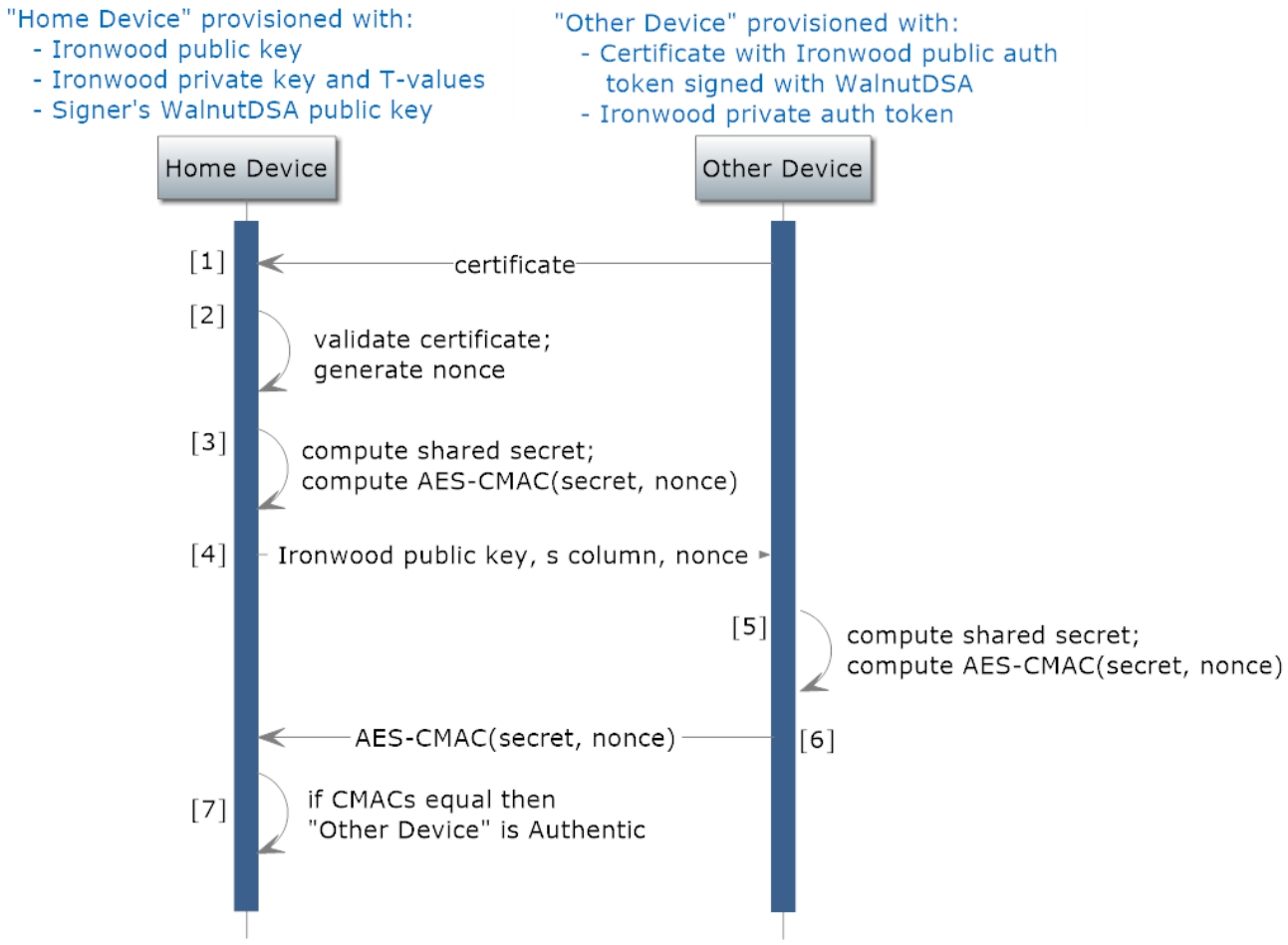

Fig. 1. Sequence diagram -- "Home Device" authenticates "Other Device"

Step	Description
[1]	"Home Device" reads certificate from "Other Device".
[2]	"Home Device" verifies certificate using Signer's WalnutDSA verification key then generates a nonce ("number-once" random number).
[3]	"Home Device" computes a shared secret using its private key and the "Other Device's" public auth token. The "Home Device" uses the shared secret to key an AES CMAC of the nonce.
[4]	"Home Device" sends the public portion of its authentication token to the "Other Device" together with a 16 byte randomly-generated nonce.
[5]	"Other Device" computes its own shared secret using the private portion of its authentication token, the "Home Device" public key and its s column.
[6]	"Other Device" sends its AES-CMAC to the "Home Device".
[7]	If the "Home Device" CMAC matches the "Other Device" CMAC, the "Home Device" considers the "Other Device" to be authentic.

Table 1. Protocol sequence for authenticating the "Home Device"

2.2 Ironwood Constants

IRONWOOD_BRAID:	Given parameters BnFq, IRONWOOD_BRAID = n.
IRONWOOD_FIELD:	Given parameters BnFq, IRONWOOD_FIELD = q.
IRONWOOD_TVALUES_BYTES:	Number of bytes in the t-values
IRONWOOD_PUBLIC_KEY_BYTES:	Byte length of Ironwood public key
IRONWOOD_AUTH_TOKEN_BYTES:	Byte length of Ironwood authentication token
IRONWOOD_SHARED_SECRET_BYTES:	Byte length of the Ironwood shared secret
IRONWOOD_S_COLUMN_BYTES:	Byte length of the Ironwood S column

2.3 Ironwood Functions

```
int ironwood_kap(const void *authtoken, size_t tokenlen,
            const void *tvalues, size_t tvalueslen,
            const void *privkey, size_t privkeylen,
            void *sharedsecret, size_t secretlen,
            void *s_column, size_t scolumnlen,
            ironwood_ctx *c)
```

Generate a shared secret and public key material for a second party.

authtoken – other party's authentication token, must be at least IRONWOOD_AUTH_TOKEN_BYTES.
tokenlen – byte length of authtoken buffer
tvalues – private data used for generating a shared secret, must be at least IRONWOOD _TVALUES_BYTES.
tvalueslen – byte length of tvalues buffer
privkey – the private key used to calculate the shared secret
privkeylen – byte length of privkey buffer
sharedsecret – output buffer for shared secret, must be at least IRONWOOD_SHARED_SECRET_BYTES.
secretlen – byte length of sharedsecret buffer available for writing
s_column – output buffer for s-column to be sent to other party, must be at least IRONWOOD_S_COLUMN_BYTES
scolumnlen – byte length of s_column available for writing
c – temporary buffer used when computing shared secret. Will be cleared on exit.

Establish a shared secret between two parties. Both parties in the key agreement require different data to compute the shared secret. The party calling this function (Alice) requires a public authentication token (authtoken) from the other party (Bob). Bob requires both a public key that matches the private key and tvalues and a public intermediate value called an s-column (s_column) from Alice. Bob's public authentication token is required before Alice can compute either the s-column or the shared secret (sharedsecret). Bob requires the s-column and Alice's public key before he can compute the same shared secret.

This function computes a shared secret sharedsecret and s-column s_column with a public authentication token authtoken and private data tvalues and privkey.

The buffer c is cleared both at function entry and function exit. This removes the need for this function to allocate data on the stack.

This method returns 1 on success, 0 on failure.

3. Walnut DSA: Walnut Digital Signature Algorithm

3.1 WalnutDSA Theory of Operation

Introduction

This section introduces SecureRF's "WalnutDSA" Digital Signature Algorithm. WalnutDSA allows one device to generate a message that may be verified by another device. More specifically, the algorithm allows a signer with a fixed private/public key pair to create a digital signature associated to a given message which can be validated by anyone who knows the public key of the signer and the WalnutDSA verification protocol.

One common application of WalnutDSA for the security solutions that SecureRF provides involves the exchange of public keys for SecureRF's Ironwood™ Key Agreement Protocol. In that protocol, two parties exchange their Ironwood public keys in order to each compute a shared secret. But how does the challenger party know that the public key it receives is really from the party it claims to be? The answer is to require the party who is challenged provide an Ironwood public key that is signed by a third-party that the challenger trusts, and to ensure that the signer's WalnutDSA public key is available to the challenger. If the signature verifies correctly using the signer's WalnutDSA public key, the challenger knows that the message could only have come from the signer since only the signer possesses the associated WalnutDSA private key. This process is depicted in Fig. 2. below where for our example, "Message" is an Ironwood public key.

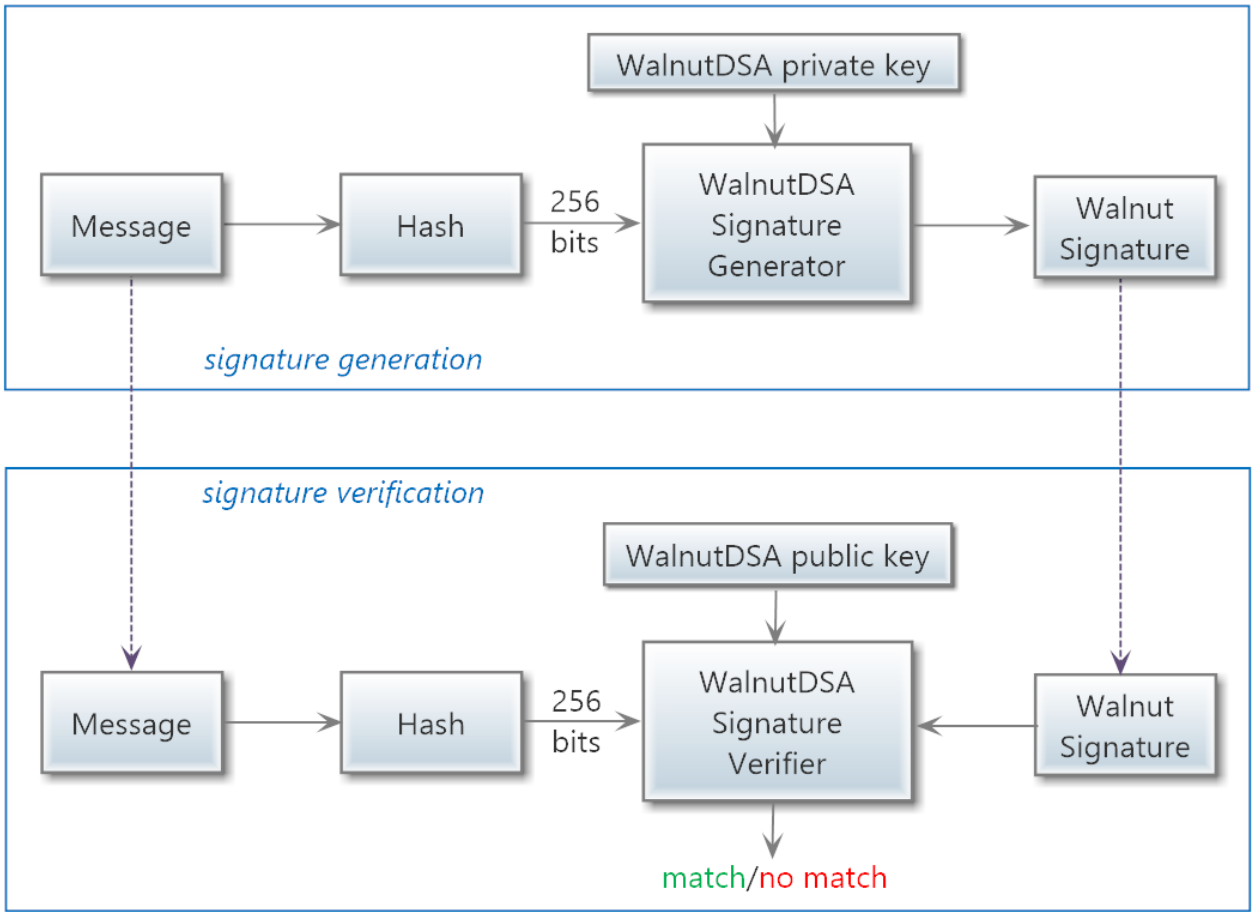

Fig. 2. Block diagram for WalnutDSA "signing" and "verification" process

SecureRF's IoT Embedded SDK contains a WalnutDSA signature verification routine. The WalnutDSA verifier requires a 256-bit input, so the message is hashed to a constant length prior to being processed. SecureRF

provides a hash function in the SDK but SDK users can substitute their own so long as the hash function for the verifier is the same as the hash function for the signer. A separate SDK will be available from SecureRF to create WalnutDSA signatures.

3.2 WalnutDSA Constants:

WALNUT_BRAID:	Given parameters BnFq, WALNUT_BRAID = n.
WALNUT_FIELD:	Given parameters BnFq, WALNUT_FIELD = q.
WALNUT_PUBLIC_KEY_BYTES:	byte length of the signer's public key
WALNUT_DGST_SIZ:	number of bytes expected in the digest

3.3 WalnutDSA Functions

```
int walnut_verify_init(const void *pubkey, const void *dgst,
      walnut_verify_ctx *c)
```

Initialize a walnut_verify_ctx.

pubkey – WALNUT_PUBLIC_KEY_BYTES byte public key from signer
dgst – WALNUT_DGST_SIZ byte digest that is to be verified.
c – walnut_verify_ctx to initialize

Walnut DSA verifies that a message has been signed with the private signer's key that is associated with a public signer's key. This is done by hashing the message to a constant length and verifying that the message digest and signature can be verified against the public signer's key.

These methods use a context to store state between calls and avoid allocating data on the stack. This allows the signature to be added in arbitrary-sized chunks.

pubkey contains the public key of the signer. dgst is a WALNUT_DGST_SIZ message. By default, this is 256 bits. It is expected to be the hash of the signed data. Both the signer and the verifier must use the same hash algorithm.

Returns 1 on success, 0 on failure.

```
int walnut_verify_update(walnut_verify_ctx *c, const void *sig,
      size_t len)
```

Update a walnut_verify_ctx with the next portion of the signature

c – walnut_verify_ctx to update
sig – pointer to the next len bytes of the signature to process
len – byte length of the signature portion stored in sig

The context can be updated in chunks or all at once. Initialize the context once with `walnut_verify_init` and update the context repeatedly with some portion of the signature until the context has been updated with the entire signature.

Returns 1 on success, 0 on failure.

```
int walnut_verify_final(walnut_verify_ctx *c)
```

Check the signature provided to the walnut_verify_ctx against the signer's public key and message digest.

c – walnut_verify_ctx context with complete signature added.

Returns 1 on success (if the message has been verified as authentic) and 0 on failure.

4. CMAC – Cipher-based Message Authentication Code

4.1 Constants:

CIPHER_KEY_SIZE: Size in bytes of the cipher key. For AES-128 this is 16 bytes.
CIPHER_BLOCK_BYTES: Size in bytes of a single block of the block cipher. For AES-128 this is 16 bytes.

4.2 Functions

```
int cmac_verify_auth(const void *key, const void *in, size_t datalen, const void
*mac, cmac_ctx *c)
```

Verify that a cmac with input in and key matches the claimed cmac mac in constant time.

key – CIPHER_KEY_SIZE byte key used by underlying block cipher
in – input message whose message will be verified
datalen – length of message in in bytes
mac – BLOCK_BYTES claimed mac
c – buffer used for temporary storage. Zeroed on function entrance and exit.

cmac_verify_auth will check to see if the mac passed in matches the data and key. Data verification takes place in constant time.

Returns 1 on success (if the mac matches with input parameters in and key) and 0 on failure.

5. Examples

See accompanying main.c source files in the “Home” and “Other” folders for sample code.

6. Note about WalnutDSA sample signatures

WalnutDSA verification run-time is affected by the length of the signature (in bytes) to be verified. The length of signatures fall within certain bounds but their size is random (within those bounds).
The SDK includes three sample signature sizes to enable you to obtain test run-time over varying signature lengths: “large”, “medium”, and “small” as shown in Fig. 3.

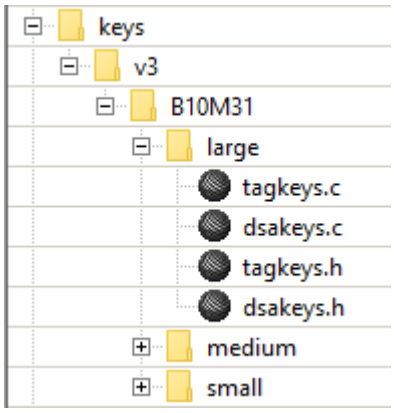

Fig. 3 Sample signature folders

6.1 Changing sample signature

Depending on your development environment, you can copy the “dsa” and “tag” keys from the large, medium, or small folder to your active project location. For example, for the IAR Embedded Workbench IDE, if you wanted to use the small sample signature, you would do two things:

1. Copy “tagkeys.c” and “dsakeys.c” from the “small” folder to your project’s file list as shown in Fig. 4.

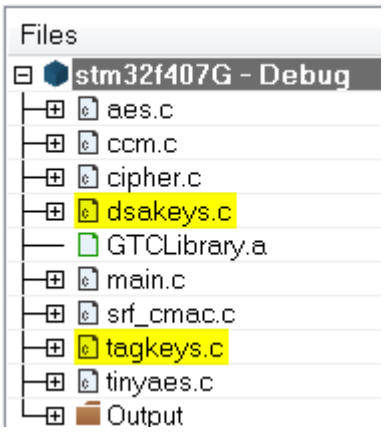

Fig. 4 Project files in IAR Workbench

2. Include the header file directory as shown in Fig. 5.

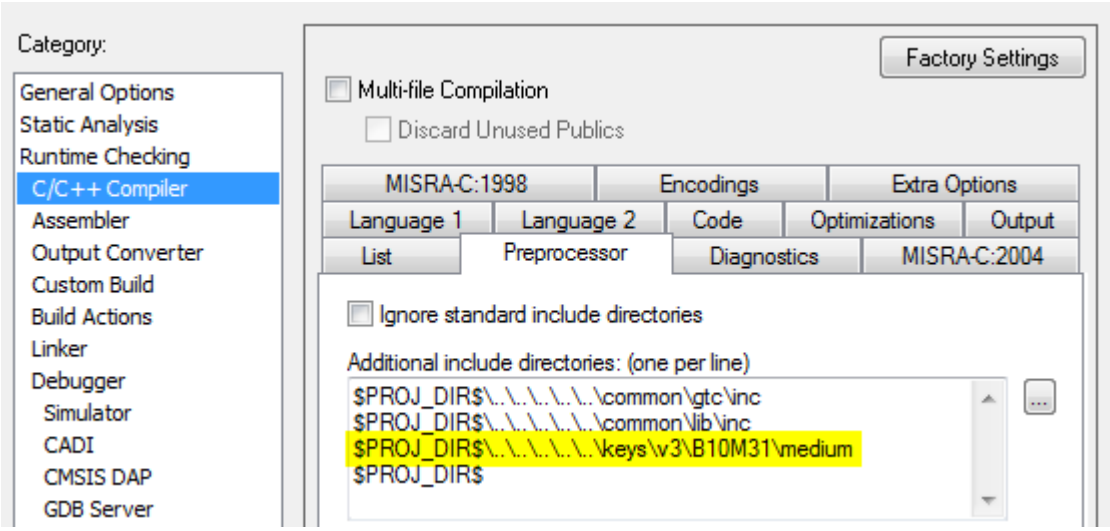

Fig. 5. Including header file directory in IAR Embedded Workbench