HASHLIB

Industry-Standard Cryptography on the TI-84+ CE

Version 9

Quick Reference

by Anthony Cagliano

Contents

Installation

`HASHLIB` is a library, not a program or an application. It is meant to be used in accordance with the same usage guidelines as the other libraries in the CE C toolchain (by MateoC). If you are a developer looking to use the library within your own project, you must follow the steps below. *NOTE: This assumes you have defined the shell variable `$CEDEV` as directed by the CE C toolchain documentuation. If you have not done this (or familiarized yourself with the toolchain documentation), I recommend doing so first before attempting to use this (or any other) library.*

- Move the library's C header and .lib files to the correct directories. This is `$CEDEV/include` for `hashlib.h` and `$CEDEV/lib/libload/` for `hashlib.lib`. You can do this manually, or by running `make install` in a Terminal window from within the directory containing this documentation file.
- Include the C header for HASHLIB in any C source file where you use anything from the library. Do this like so: `#include <hashlib.h>`.
- Use defines or functions from the library freely within any C source file where the library's header is included.

If you are an end user who needs the library present, either for testing or for using a program that requires HASHLIB, install it by sending the library file, the TI application variable `hashlib.8xv` to your device. If you do not do this, programs that use it will fail to start, instead returning to the homescreen and yelling at you about a missing library.

1 Enumerations

Enumeration	Values	Meaning
hash_algorithms	SHA256	SHA-256 hash algorithm
aes_cipher_modes	AES_MODE_CBC	AES CBC cipher mode
	AES_MODE_CTR	AES CTR cipher mode
aes_padding_schemes	SCHM_PKCS7	PKCS#7 padding mode
	SCHM_ISO2	ISO-9797 M2 padding mode
	SCHM_DEFAULT	selects default padding mode (PKCS#7)
aes_error_t	AES_OK	AES operation succeeded
	AES_INVALID_ARG	one or more invalid inputs
	AES_INVALID_MSG	message invalid
	AES_INVALID_CIPHERMODE	unrecognized cipher mode
	AES_INVALID_PADDINGMODE	unrecognized padding mode
	AES_INVALID_CIPHERTEXT	invalid ciphertext or bad hash
rsa_error_t	RSA_OK	RSA operation succeeded
	RSA_INVALID_ARG	one or more invalid inputs
	RSA_INVALID_MSG	message too long
	RSA_INVALID_MODULUS	modulus not odd
	RSA_ENCODING_ERROR	OAEP encoding error

2 Defines

Define	Purpose
fastRam_Safe	Defines a region of fast memory generally safe to use for short-term computations.
fastRam_Unsafe	Defines a region of fast memory that generally gets used by the SRNG and SHA-256 algorithms.
SHA256_DIGEST_LEN	Defines the binary-length of a SHA-256 hash digest.
AES_BLOCKSIZE	Defines the block size of the of the AES cipher, in bytes.
AES_IVSIZE	Defines the size of the AES initialization vector.

3 Macros

Macro	Function
aes_outsize(len)	Outputs the required buffer size for a properly padded ciphertext given an input unpadded plaintext length **len**.
aes_extoutsize(len)	Outputs the required buffer size for the above with an additional block for an initialization vector.

4 Implementations

4.1 Cryptographically-Secure Random Number Generator (CSRNG)

Many of the psuedorandom number generators (PRNGs) you find in computers, even the one within the C toolchain for the CE, are insecure for cryptographic purposes. They produce statistical randomness, but the state is generally seeded using a value such as rtc_Time(). If an adversary reconstructs the seed, every output of the PRNG becomes computable with little effort. These types of PRNGs are called deterministic algorithms–given the input, the output is predictable. These PRNGs work for games and other applications where the illusion of randomness is sufficient, but they are not safe for cryptography.

A secure RNG is a random number generator that is not only statistically random, but also passes the next-bit and state compromise tests. The next-bit test is defined like so:

> Given the prior output of the RNG (bits 0 -> i), the next bit (i+1) of the output cannot be predicted by a polynomial-time statistical test with a probability non-negligibly greater than 50%.

In simpler terms, a secure RNG must be unpredictable, or "entropic". The state compromise test is defined like so:

> An adversary gaining knowledge of the initial state of the RNG does not gain any information about its output.

The CSRNG provided by HASHLIB solves both tests like so:
(next-bit)

- The CSRNG's output is derived from a 119-byte entropy pool created by reading data from the most entropic byte located within floating memory on the device.
- The "entropic byte" is a byte containing a bit that that, out of 1024 test reads, has the closest to a 50/50 split between 1's and 0's.
- The byte containing that bit is read in xor mode seven times per byte to offset any hardware-based correlation between subsequent reads from the entropic byte.
- The entropy pool is then run through a cryptographic hash to spread that entropy evenly through the returned random value.
- The PRNG produces 96.51 bits of entropy per 32 bit number returned. (See *HASHLIB Cryptanalysis, Section 1.3* for proof of entropy.)
- **Assertion**: *A source of randomness with sufficient entropy passed through a cryptographic hash will produce output that passes all statistical randomness tests as well as the next-bit test.*

(state compromise)

- The entropy pool is discarded after it is used once, and a new pool is generated for the next random number. This means that the prior state has no bearing on the next output.
- The CSRNG destroys its own state after the random number is generated so that the state used to generate it does not persist in memory.

bool csrand_init(void)

Initializes the CSRNG by polling the 512 bytes from 0xD65800 to 0xD66000 looking for a floating bit within unmapped memory with the least bias. Returns **True** if the source selection succeeded and **False** if it failed. Be sure to intercept and handle a return value of False from this function.

uint32_t csrand_get(void)

Returns a securely psuedorandom 32-bit (4 byte) unsigned integer.

bool csrand_fill(void* buffer, size_t size)

buffer Pointer to an arbitrary buffer to fill with random bytes.
size Number of bytes to write.

4.2 Cryptographic Hashing

A cryptographic hash is used to validate that data is unchanged between two endpoints. It is similar to a checksum, but checksums can be easily fooled; cryptographic hashes are a lot harder to fool due to how they distribute the bits in a data stream. The general use of a hash is as follows:

(a) The party sending a message hashes it and includes that hash as part of the message.

(b) The recipient hashes the message (excluding the hash) themselves and then compares that hash to the one included in the message.

(c) If the hashes match, the message is complete and unaltered. If not, the message is incomplete or has been tampered with and should be discarded.

void hash_init(hash_ctx* ctx, uint8_t hash_alg)

Initializes the hash-state context for use.
ctx A pointer to an instance of *hash_ctx*.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

void hash_update(hash_ctx* ctx, const void* data, size_t len)

Updates the hash-state with new data. Be sure to initialize it first!
ctx A pointer to an instance of *hash_ctx*.
data A pointer to arbitrary data to hash.
len The size, in bytes, of the data to hash.

void hash_final(hash_ctx* ctx, void* digest)

Performs final transformations on the context and returns a digest from the current hash-state. Does not destroy the context. It can still be used with the same data stream if needed.
ctx A pointer to an instance of *hash_ctx*.
digest A pointer to a buffer to write the digest to.

4.3 Hash-Based Message Authentication Code (HMAC)

An HMAC generates a more secure hash by using a key known only to authorized parties as part of the hash initialization. Thus, while normal hashes can be generated and verified by anyone, only the parties with the key can generate and validate using a HMAC hash.

void hmac_init(hmac_ctx* ctx, const void* key, size_t keylen, uint8_t hash_alg)

Initializes the HMAC hash-state context for use.
ctx A pointer to an instance of *hmac_ctx*.
key A pointer to the key to use in the HMAC initilaization.
keylen The length of the key, in bytes.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

> NIST recommends a minimum **key** length of 128 bits, or 16 bytes.

void hmac_update(hmac_ctx* ctx, const void* data, size_t len)

Updates the HMAC hash-state with new data. Be sure to initialize it first!
ctx A pointer to an instance of *hmac_ctx*.
data A pointer to arbitrary data to hash.
len The size, in bytes, of the data to hash.

void hmac_final(hmac_ctx* ctx, void* digest)

Performs final transformations on the context and returns a digest from the current hash-state. Does not destroy the context. It can still be used with the same data stream if needed.
ctx A pointer to an instance of *hmac_ctx*.
digest A pointer to a buffer to write the digest to.

4.4 Mask and Key Generation

Sometimes in cryptography you need to generate hashes or keys of an arbitrary size. Two related, but different, functions exist to fill this role. The first of the two is a **mask generation function (MGF)**. A MGF generates a mask of arbitrary length by passing the data with a counter appended to it to a cryptographic primative such as SHA-256. The second of the two is a **password-based key derivation function**. A PBKDF works by using the supplied password as the key for an HMAC and then hashing the salt for the given number of rounds for each block of output.

void hash_mgf1(const void* data, size_t datalen,
void* outbuf, size_t outlen, uint8_t hash_alg)

Generates a mask of a given length from the given data.
data A pointer to data to generate the mask with.
datalen The length, in bytes, of the data.
outbuf A pointer to a buffer to write the mask to.
outlen The number of bytes of the mask to output.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

void hmac_pbkdf2(const char* password, size_t passlen,
void* key, size_t keylen
const void* salt, size_t saltlen,
size_t rounds, uint8_t hash_alg)

Generates a key of given length from a password, salt, and a given number of rounds.
password A pointer to a string containing the password.
passlen The length of the password string.
key A pointer to a buffer to write the key to.
keylen The number of bytes of the key to output.
salt A pointer to a buffer containing psuedorandom bytes.
saltlen The length of the salt. NIST recommends at least 128 bits (16 bytes).
rounds The number of times to iterate the HMAC function per block in the output.
hash_alg The hashing algorithm to use. See `hash_algorithms` (Enumerations).

> NIST recommends a minimum **salt** length of 128 bits, or 16 bytes.

4.5 Advanced Encryption Standard (AES)

The **Advanced Encryption Standard (AES)** is a symmetric encryption system (the same key works for encryption and decryption) and a block cipher (the encryption occurs in blocks of a certain size [16 bytes for AES]). AES is one of the most secure encryption systems in use today, expected to remain secure even through the advent of quantum computing. It is also fast and more secure than asymmetric encryption for smaller key sizes.

bool aes_loadkey(const void* key, const aes_ctx* ks, size_t keylen)

Populates an AES key schedule context given an AES key.
key Pointer to a buffer containing the AES key.
ks Pointer to an AES key schedule to output.
keylen The length, in bytes, of the AES key.

> It is recommended that you cycle (regenerate) your AES key after it is used to encrypt between 2^{48} and 2^{64} blocks of data (depending on cipher mode). You will never feasibly achieve this on a calculator.

aes_error_t aes_encrypt(const void* plaintext, size_t len, void* ciphertext, const aes_ctx* ks, const void* iv, uint8_t ciphermode, uint8_t paddingmode)

Encrypts the given message using the AES cipher.
plaintext A pointer to a buffer containing data to encrypt.
len The length of the data to encrypt.
ciphertext A pointer to a buffer to write the encrypted output to.
ks A pointer to an AES key schedule output by aes_loadkey().
iv A pointer to an initialization vector, a buffer equal to the block size in length containing psuedo-random bytes.
ciphermode The cipher mode to use for encryption. See enumerations.
paddingmode The padding mode to use prior to encryption, if needed. See enumerations.

Generate a new **IV** for each message.
Ciphertext should be equal in size to the smallest multiple of the blocksize that can hold the encrypted data and any necessary padding. Helper macros provided.

aes_error_t aes_decrypt(const void* ciphertext, size_t len, void* plaintext, const aes_ctx* ks, const void* iv, uint8_t ciphermode, uint8_t paddingmode)

Decrypts the given message using the AES cipher.
ciphertext A pointer to a buffer containing data to decrypt.
len The length of the data to decrypt.
plaintext A pointer to a buffer to write the decrypted output to.
ks A pointer to an AES key schedule output by aes_loadkey().
iv A pointer to an initialization vector, a buffer equal to the block size in length containing psuedo-random bytes.
ciphermode The cipher mode to use for decryption. See enumerations.
paddingmode The padding mode to use after decryption, if needed. See enumerations.

Decrypt using the same **IV** that the message was encrypted with.

4.6 RSA Public Key Encryption

Public key encryption is a form of asymmetric encryption generally used to share a secret key for AES or another symmetric encryption system. To communicate between two parties, both need a public key and a private key. The public key is (hence the term "public") common knowledge and is sent to other parties in the clear. The private key is known only to the host. The public key is used to encrypt messages for the host, and the private key is used by the host to decrypt those messages. The public key and private key are inverses of each other such that:
$encrypted = message^{\,public\ exponent} \% public\ modulus$
$message = encrypted^{\,private\ exponent} \% private\ modulus$
RSA is very slow, especially on the TI-84+ CE. Encrypting with just a 1024-bit modulus will take several seconds. For this reason, do not use RSA for sustained encrypted communication. Use RSA once to share a key with a remote host, then use AES. Also the RSA implementation in this library is encryption only. This means you will need to handshake with a server to create a secure session, like so:

(a) Connect to remote host. Let that server generate a public and private key pair. Send the public key to the calculator.

(b) Use hashlib to generate an AES secret. Encrypt that secret using RSA and send the encrypted message to the remote host.

(c) Decrypt the message on the server, and set up an AES session using the secret just shared with the remote host.

rsa_error_t rsa_encrypt(const void* msg, size_t msglen, void* ciphertext, const void* pubkey, size_t keylen, uint8_t oaep_hash_alg)

Encrypts the given message using the given public key and the public exponent 65537. Applies the OAEP v2.2 encoding scheme prior to encryption.
msg A pointer to a buffer containing data to encrypt.
msglen The length of the data to encrypt.
ciphertext A pointer to a buffer to write the encrypted output to.
pubkey A pointer to an RSA public modulus.
keylen The length of the RSA public modulus, in bytes.
oaep_hash_alg The hashing algorithm to use for OAEP. See `hash_algorithms` (Enumerations).

4.7 Miscellaneous Functions

void digest_tostring(const void* digest, size_t len, const char* hexstr)

Outputs a textual representation of the hex encoding of a binary digest.
Ex: 0xfe, 0xa4, 0xc1, 0xf2 => "FEA4C1F2"
digest A pointer to a digest to convert to a string.
len The length of the digest, in bytes, to convert.
hexstr A pointer to a buffer to write the string. Must be equal to twice the digest length + 1.

void digest_compare(const void* digest1, const void* digest2, size_t len)

Compares the given number of bytes at *digest*1 with *digest*2 in a manner that is resistant to timing analysis.
digest1 A pointer to the first buffer to compare.
digest2 A pointer to the second buffer to compare.
len The number of bytes to compare.

4.8 Addendum: Authenticated Encryption with HASHLIB

While HASHLIB provides no implemented authenticated encryption schemes, it is possible to construct one using the provided API. This section will explain how to do so, review some rules for constructing one properly (as well as some pitfalls that can arise) and give an example that uses the existing API.

Authenticated encryption is an encryption scheme that combines a cipher with a cryptographic hash, producing a data stream that is not only obfuscated but also has its integrity and authenticity verifiable. The aforementioned hash is usually appended to the end of the data stream and is the output of passing the entire data stream (encrypted and unencrypted portions) to the hashing algorithm. Using a cryptographic hash like SHA-256 will ensure ciphertext integrity but not authenticity. An HMAC can ensure both integrity and authenticity since it incorporates a key that you can exchange with authorized parties, allowing only those parties to authenticate the message.

Rules for implementing authenticated encryption properly:

(a) **Use different keys for encryption and authentication. Do not interchange them.**

(b) **Generate new keys for each secure session. Never reuse keys.**

(c) **Encrypt, then authenticate. Run your message through the cipher first, then hash the output. This method is more secure.**

(d) **Hash the entire message, encrypted and unencrypted portions.** Just because a segment of a message doesn't need to be hidden doesn't mean we shouldn't check for tampering.

Sample authenticated encryption construction using HASHLIB API

```
// assume data in 'message', header in 'header'
char packet[200];  // this can be smaller or larger, as needed
uint8_t cipher_key[32];  // this can be 16, 24, or 32 for AES
uint8_t hmac_key[32];   // this is arbitrary, can be any size >=16
aes_ctx ks;
hmac_ctx hmac;

// generate AES key, HMAC key, and IV
if(!csrand_init()) return 1; // handle this. Always handle this.
csrand_fill(cipher_key, sizeof cipher_key);
csrand_fill(hmac_key, sizeof hmac_key);
csrand_fill(&packet[sizeof header], AES_BLOCKSIZE);

// copy message and headers into place. Packet format [headers][IV][message]
memcpy(packet, header, sizeof header);
memcpy(&packet[AES_BLOCKSIZE + sizeof header], message, sizeof message);

// initialize AES key schedule and encrypt message
aes_loadkey(key, &ks, 32);
aes_encrypt(&packet[AES_BLOCKSIZE + sizeof header], sizeof message, &packet[
AES_BLOCKSIZE + sizeof header], ks, &packet[sizeof header], AES_MODE_CTR, SCHM_DEFAULT);

// append HMAC to message. Packet format [headers][IV][message][hmac]
hmac_init(&hmac, hmac_key, sizeof hmac_key, SHA256);
hmac_update(&hmac, packet, sizeof headers + sizeof message + AES_BLOCKSIZE);
hmac_final(&hmac, &packet[sizeof headers + sizeof message + AES_BLOCKSIZE]);
```

5 Contributors

- Anthony Cagliano [cryptographer, lead developer]
- beckadamtheinventor [contributing developer, assembly conversions]
- commandblockguy [contributing developer]
- Zeroko [information on entropy on TI-84+ CE]
- jacobly [ez80 implementation of digest_compare and _powmod for RSA]

6 Disclaimer

HASHLIB is a work-in-progress and has seen very little time as the forerunning cryptography library for the TI-84+ CE calculator. This means that it has not had much time to be thoroughly analyzed, and due to some hardware constraints may never offer total security against every possible attack. For this reason, I heavily advise that however secure HASHLIB may be, you never use it for encrypting truly sensitive data like online banking and other accounts, credit card information, and the like over an insecure network. It is likely safe enough to be used to encrypt data transfers and account login for TI-84+ CE game servers and package managers like the ones currently under development. By using this software you release and save harmless its developer(s) from liability for data compromise that may arise should you use this software.