LibTomCrypt

Developer Manual

Tom St Denis
LibTom Projects

This document is part of the LibTomCrypt package and is hereby released into the public domain.

Open Source. Open Academia. Open Minds.

Tom St Denis
Ottawa, Ontario
Canada

Contents

List of Figures

Chapter 1

Introduction

1.1 What is the LibTomCrypt?

LibTomCrypt is a portable ISO C cryptographic library meant to be a tool set for cryptographers who are designing cryptosystems. It supports symmetric ciphers, one-way hashes, pseudo-random number generators, public key cryptography (via PKCS #1 RSA, DH or ECCDH), and a plethora of support routines.

The library was designed such that new ciphers/hashes/PRNGs can be added at run-time and the existing API (and helper API functions) are able to use the new designs automatically. There exists self-check functions for each block cipher and hash function to ensure that they compile and execute to the published design specifications. The library also performs extensive parameter error checking to prevent any number of run-time exploits or errors.

1.1.1 What the library IS for?

The library serves as a toolkit for developers who have to solve cryptographic problems. Out of the box LibTomCrypt does not process SSL or OpenPGP messages, it doesn't read X.509 certificates, or write PEM encoded data. It does, however, provide all of the tools required to build such functionality. LibTomCrypt was designed to be a flexible library that was not tied to any particular cryptographic problem.

1.2 Why did I write it?

You may be wondering, *Tom, why did you write a crypto library. I already have one.* Well the reason falls into two categories:

1. I am too lazy to figure out someone else's API. I'd rather invent my own simpler API and use that.
2. It was (still is) good coding practice.

The idea is that I am not striving to replace OpenSSL or Crypto++ or Cryptlib or etc. I'm trying to write my **own** crypto library and hopefully along the way others will appreciate the work.

With this library all core functions (ciphers, hashes, prngs, and bignum) have the same prototype definition. They all load and store data in a format independent of the platform. This means if you encrypt with Blowfish on a PPC it should decrypt on an x86 with zero problems. The consistent API also means that if you learn how to use Blowfish with the library you know how to use Safer+, RC6, or Serpent as well. With all of the core functions there are central descriptor tables that can be used to make a program automatically pick between ciphers, hashes and PRNGs at run-time. That means your application can support all ciphers/hashes/prngs/bignum without changing the source code.

Not only did I strive to make a consistent and simple API to work with but I also attempted to make the library configurable in terms of its build options. Out of the box the library will build with any modern version of GCC without having to use configure scripts. This means that the library will work with platforms where development tools may be limited (e.g. no autoconf).

On top of making the build simple and the API approachable I've also attempted for a reasonably high level of robustness and efficiency. LibTomCrypt traps and returns a series of errors ranging from invalid arguments to buffer overflows/overruns. It is mostly thread safe and has been clocked on various platforms with *cycles per byte* timings that are comparable (and often favourable) to other libraries such as OpenSSL and Crypto++.

1.2.1 Modular

The LibTomCrypt package has also been written to be very modular. The block ciphers, one–way hashes, pseudo–random number generators (PRNG), and bignum math routines are all used within the API through *descriptor* tables which are essentially structures with pointers to functions. While you can still call particular functions directly (*e.g. sha256_process()*) this descriptor interface allows the developer to customize their usage of the library.

For example, consider a hardware platform with a specialized RNG device. Obviously one would like to tap that for the PRNG needs within the library (*e.g. making a RSA key*). All the developer has to do is write a descriptor and the few support routines required for the device. After that the rest of the API can make use of it without change. Similarly imagine a few years down the road when AES2 (*or whatever they call it*) has been invented. It can be added to the library and used within applications with zero modifications to the end applications provided they are written properly.

This flexibility within the library means it can be used with any combination of primitive algorithms and unlike libraries like OpenSSL is not tied to direct routines. For instance, in OpenSSL there are CBC block mode routines for every single cipher. That means every time you add or remove a cipher from the library you have to update the associated support code as well. In LibTomCrypt the associated code (*chaining modes in this case*) are not directly tied to the ciphers. That is a new cipher can be added to the library by simply providing the key setup, ECB decrypt and encrypt and test vector routines. After that all five chaining mode routines can make use of the cipher right away.

1.3 License

1.4 Patent Disclosure

The author (Tom St Denis) is not a patent lawyer so this section is not to be treated as legal advice. To the best of the author's knowledge the only patent related issues within the library are the RC5 and RC6 symmetric block ciphers. They can be removed from a build by simply commenting out the two appropriate lines in *tomcrypt_custom.h*. The rest of the ciphers and hashes are patent free or under patents that have since expired.

The RC2 and RC4 symmetric ciphers are not under patents but are under trademark regulations. This means you can use the ciphers you just can't advertise that you are doing so.

1.5 Thanks

I would like to give thanks to the following people (in no particular order) for helping me develop this project from early on:

1. Richard van de Laarschot
2. Richard Heathfield
3. Ajay K. Agrawal
4. Brian Gladman
5. Svante Seleborg
6. Clay Culver
7. Jason Klapste
8. Dobes Vandermeer
9. Daniel Richards
10. Wayne Scott
11. Andrew Tyler
12. Sky Schulz
13. Christopher Imes

There have been quite a few other people as well. Please check the change log to see who else has contributed from time to time.

Chapter 2

The Application Programming Interface (API)

2.1 Introduction

In general the API is very simple to memorize and use. Most of the functions return either **void** or **int**. Functions that return **int** will return **CRYPT_OK** if the function was successful, or one of the many error codes if it failed. Certain functions that return int will return -1 to indicate an error. These functions will be explicitly commented upon. When a function does return a CRYPT error code it can be translated into a string with

```
const char *error_to_string(int err);
```

An example of handling an error is:

```
void somefunc(void)
{
   int err;

   /* call a cryptographic function */
   if ((err = some_crypto_function(...)) != CRYPT_OK) {
      printf("A crypto error occurred, %s\n", error_to_string(err));
      /* perform error handling */
   }
   /* continue on if no error occurred */
}
```

There is no initialization routine for the library and for the most part the code is thread safe. The only thread related issue is if you use the same symmetric cipher, hash or public key state data in multiple threads. Normally that is not an issue.

To include the prototypes for *LibTomCrypt.a* into your own program simply include *tomcrypt.h* like so:

```
#include <tomcrypt.h>
int main(void) {
```

```
    return 0;
}
```

The header file *tomcrypt.h* also includes *stdio.h*, *string.h*, *stdlib.h*, *time.h* and *ctype.h*.

2.2 Macros

There are a few helper macros to make the coding process a bit easier. The first set are related to loading and storing 32/64-bit words in little/big endian format. The macros are:

STORE32L(x, y)	**unsigned long** x, **unsigned char** *y	$x \to y[0 \ldots 3]$
STORE64L(x, y)	**unsigned long long** x, **unsigned char** *y	$x \to y[0 \ldots 7]$
LOAD32L(x, y)	**unsigned long** x, **unsigned char** *y	$y[0 \ldots 3] \to x$
LOAD64L(x, y)	**unsigned long long** x, **unsigned char** *y	$y[0 \ldots 7] \to x$
STORE32H(x, y)	**unsigned long** x, **unsigned char** *y	$x \to y[3 \ldots 0]$
STORE64H(x, y)	**unsigned long long** x, **unsigned char** *y	$x \to y[7 \ldots 0]$
LOAD32H(x, y)	**unsigned long** x, **unsigned char** *y	$y[3 \ldots 0] \to x$
LOAD64H(x, y)	**unsigned long long** x, **unsigned char** *y	$y[7 \ldots 0] \to x$
BSWAP(x)	**unsigned long** x	Swap bytes

Figure 2.1: Load And Store Macros

There are 32 and 64-bit cyclic rotations as well:

ROL(x, y)	**unsigned long** x, **unsigned long** y	$x << y, 0 \le y \le 31$
ROLc(x, y)	**unsigned long** x, **const unsigned long** y	$x << y, 0 \le y \le 31$
ROR(x, y)	**unsigned long** x, **unsigned long** y	$x >> y, 0 \le y \le 31$
RORc(x, y)	**unsigned long** x, **const unsigned long** y	$x >> y, 0 \le y \le 31$
ROL64(x, y)	**unsigned long** x, **unsigned long** y	$x << y, 0 \le y \le 63$
ROL64c(x, y)	**unsigned long** x, **const unsigned long** y	$x << y, 0 \le y \le 63$
ROR64(x, y)	**unsigned long** x, **unsigned long** y	$x >> y, 0 \le y \le 63$
ROR64c(x, y)	**unsigned long** x, **const unsigned long** y	$x >> y, 0 \le y \le 63$

Figure 2.2: Rotate Macros

2.3 Functions with Variable Length Output

Certain functions such as (for example) *rsa_export()* give an output that is variable length. To prevent buffer overflows you must pass it the length of the buffer where the output will be stored. For example:

```
#include <tomcrypt.h>
int main(void) {
    rsa_key key;
    unsigned char buffer[1024];
    unsigned long x;
    int err;

    /* ... Make up the RSA key somehow ... */

    /* lets export the key, set x to the size of the
     * output buffer */
    x = sizeof(buffer);
    if ((err = rsa_export(buffer, &x, PK_PUBLIC, &key)) != CRYPT_OK) {
       printf("Export error: %s\n", error_to_string(err));
       return -1;
```

```
    }

    /* if rsa_export() was successful then x will have
     * the size of the output */
    printf("RSA exported key takes %d bytes\n", x);

    /* ... do something with the buffer */

    return 0;
}
```

In the above example if the size of the RSA public key was more than 1024 bytes this function would return an error code indicating a buffer overflow would have occurred. If the function succeeds, it stores the length of the output back into x so that the calling application will know how many bytes were used.

As of v1.13, most functions will update your length on failure to indicate the size required by the function. Not all functions support this so please check the source before you rely on it doing that.

2.4 Functions that need a PRNG

Certain functions such as *rsa_make_key()* require a Pseudo Random Number Generator (PRNG). These functions do not setup the PRNG themselves so it is the responsibility of the calling function to initialize the PRNG before calling them.

Certain PRNG algorithms do not require a *prng_state* argument (sprng for example). The *prng_state* argument may be passed as **NULL** in such situations.

```
#include <tomcrypt.h>
int main(void) {
    rsa_key key;
    int     err;

    /* register the system RNG */
    register_prng(&sprng_desc)

    /* make a 1024-bit RSA key with the system RNG */
    if ((err = rsa_make_key(NULL, find_prng("sprng"), 1024/8, 65537, &key))
        != CRYPT_OK) {
       printf("make_key error: %s\n", error_to_string(err));
       return -1;
    }

    /* use the key ... */

    return 0;
}
```

2.5 Functions that use Arrays of Octets

Most functions require inputs that are arrays of the data type *unsigned char*. Whether it is a symmetric key, IV for a chaining mode or public key packet it is assumed that regardless of the actual size of *unsigned char* only the lower eight bits contain data. For example, if you want to pass a 256 bit key to a symmetric ciphers setup routine, you must pass in (a pointer to) an array of 32 *unsigned char* variables. Certain routines (such as SAFER+) take special care to work properly on platforms where an *unsigned char* is not eight bits.

For the purposes of this library, the term *byte* will refer to an octet or eight bit word. Typically an array of type *byte* will be synonymous with an array of type *unsigned char*.

Chapter 3

Symmetric Block Ciphers

3.1 Core Functions

LibTomCrypt provides several block ciphers with an ECB block mode interface. It is important to first note that you should never use the ECB modes directly to encrypt data. Instead you should use the ECB functions to make a chaining mode, or use one of the provided chaining modes. All of the ciphers are written as ECB interfaces since it allows the rest of the API to grow in a modular fashion.

3.1.1 Key Scheduling

All ciphers store their scheduled keys in a single data type called *symmetric_key*. This allows all ciphers to have the same prototype and store their keys as naturally as possible. This also removes the need for dynamic memory allocation, and allows you to allocate a fixed sized buffer for storing scheduled keys. All ciphers must provide six visible functions which are (given that XXX is the name of the cipher) the following:

```
int XXX_setup(const unsigned char *key,
                              int  keylen,
                              int  rounds,
                    symmetric_key *skey);
```

The XXX_setup() routine will setup the cipher to be used with a given number of rounds and a given key length (in bytes). The number of rounds can be set to zero to use the default, which is generally a good idea.

If the function returns successfully the variable *skey* will have a scheduled key stored in it. It's important to note that you should only used this scheduled key with the intended cipher. For example, if you call *blowfish_setup()* do not pass the scheduled key onto *rc5_ecb_encrypt()*. All built–in setup functions do not allocate memory off the heap so when you are done with a key you can simply discard it (e.g. they can be on the stack). However, to maintain proper coding practices you should always call the respective XXX_done() function. This allows for quicker porting to applications with externally supplied plugins.

3.1.2 ECB Encryption and Decryption

To encrypt or decrypt a block in ECB mode there are these two functions per cipher:

```
int XXX_ecb_encrypt(const unsigned char *pt,
                          unsigned char *ct,
                          symmetric_key *skey);

int XXX_ecb_decrypt(const unsigned char *ct,
                          unsigned char *pt,
                          symmetric_key *skey);
```

These two functions will encrypt or decrypt (respectively) a single block of text[1], storing the result in the *ct* buffer (*pt* resp.). It is possible that the input and output buffer are the same buffer. For the encrypt function *pt*[2] is the input and *ct*[3] is the output. For the decryption function it's the opposite. They both return **CRYPT_OK** on success. To test a particular cipher against test vectors[4] call the following self-test function.

3.1.3 Self–Testing

```
int XXX_test(void);
```

This function will return **CRYPT_OK** if the cipher matches the test vectors from the design publication it is based upon.

3.1.4 Key Sizing

For each cipher there is a function which will help find a desired key size. It is specified as follows:

```
int XXX_keysize(int *keysize);
```

Essentially, it will round the input keysize in *keysize* down to the next appropriate key size. This function will return **CRYPT_OK** if the key size specified is acceptable. For example:

```
#include <tomcrypt.h>
int main(void)
{
   int keysize, err;

   /* now given a 20 byte key what keysize does Twofish want to use? */
   keysize = 20;
   if ((err = twofish_keysize(&keysize)) != CRYPT_OK) {
      printf("Error getting key size: %s\n", error_to_string(err));
      return -1;
   }
   printf("Twofish suggested a key size of %d\n", keysize);
   return 0;
}
```

[1] The size of which depends on which cipher you are using.
[2] pt stands for plaintext.
[3] ct stands for ciphertext.
[4] As published in their design papers.

This should indicate a keysize of sixteen bytes is suggested by storing 16 in *keysize.*

3.1.5 Cipher Termination

When you are finished with a cipher you can de–initialize it with the done function.

```
void XXX_done(symmetric_key *skey);
```

For the software based ciphers within LibTomCrypt, these functions will not do anything. However, user supplied cipher descriptors may require to be called for resource management purposes. To be compliant, all functions which call a cipher setup function must also call the respective cipher done function when finished.

3.1.6 Simple Encryption Demonstration

An example snippet that encodes a block with Blowfish in ECB mode.

```
#include <tomcrypt.h>
int main(void)
{
   unsigned char pt[8], ct[8], key[8];
   symmetric_key skey;
   int err;

   /* ... key is loaded appropriately in key ... */
   /* ... load a block of plaintext in pt ... */

   /* schedule the key */
   if ((err = blowfish_setup(key, /* the key we will use */
                             8, /* key is 8 bytes (64-bits) long */
                             0, /* 0 == use default # of rounds */
                         &skey) /* where to put the scheduled key */
       ) != CRYPT_OK) {
      printf("Setup error: %s\n", error_to_string(err));
      return -1;
   }

   /* encrypt the block */
   blowfish_ecb_encrypt(pt,       /* encrypt this 8-byte array */
                        ct,       /* store encrypted data here */
                        &skey);   /* our previously scheduled key */

   /* now ct holds the encrypted version of pt */

   /* decrypt the block */
   blowfish_ecb_decrypt(ct,       /* decrypt this 8-byte array */
                        pt,       /* store decrypted data here */
                        &skey);   /* our previously scheduled key */

   /* now we have decrypted ct to the original plaintext in pt */
```

```
   /* Terminate the cipher context */
   blowfish_done(&skey);

   return 0;
}
```

3.2 Key Sizes and Number of Rounds

As a general rule of thumb, do not use symmetric keys under 80 bits if you can help it. Only a few of the ciphers support smaller keys (mainly for test vectors anyways). Ideally, your application should be making at least 256 bit keys. This is not because you are to be paranoid. It is because if your PRNG has a bias of any sort the more bits the better. For example, if you have $\Pr[X = 1] = \frac{1}{2} \pm \gamma$ where $|\gamma| > 0$ then the total amount of entropy in N bits is $N \cdot -log_2\left(\frac{1}{2} + |\gamma|\right)$. So if γ were 0.25 (a severe bias) a 256-bit string would have about 106 bits of entropy whereas a 128-bit string would have only 53 bits of entropy.

The number of rounds of most ciphers is not an option you can change. Only RC5 allows you to change the number of rounds. By passing zero as the number of rounds all ciphers will use their default number of rounds. Generally the ciphers are configured such that the default number of rounds provide adequate security for the given block and key size.

3.3 The Cipher Descriptors

To facilitate automatic routines an array of cipher descriptors is provided in the array *cipher_descriptor*. An element of this array has the following (partial) format (See Section 15.2):

```
struct _cipher_descriptor {
   /** name of cipher */
   char *name;

   /** internal ID */
   unsigned char ID;

   /** min keysize (octets) */
   int  min_key_length,

   /** max keysize (octets) */
        max_key_length,

   /** block size (octets) */
        block_length,

   /** default number of rounds */
        default_rounds;
...<snip>...
};
```

Where *name* is the lower case ASCII version of the name. The fields *min_key_length* and *max_key_length* are the minimum and maximum key sizes in bytes. The *block_length* member is the block size of the cipher in bytes. As a good rule of thumb it is assumed that the cipher supports the min and max key lengths but not always everything in between. The *default_rounds* field is the default number of rounds that will be used.

For a plugin to be compliant it must provide at least each function listed before the accelerators begin. Accelerators are optional, and if missing will be emulated in software.

The remaining fields are all pointers to the core functions for each cipher. The end of the cipher_descriptor array is marked when *name* equals **NULL**.

As of this release the current cipher_descriptors elements are the following:

Name	**Descriptor Name**	**Block Size**	**Key Range**	**Rounds**
Blowfish	blowfish_desc	8	8 ... 56	16
X-Tea	xtea_desc	8	16	32
RC2	rc2_desc	8	8 ... 128	16
RC5-32/12/b	rc5_desc	8	8 ... 128	12 ... 24
RC6-32/20/b	rc6_desc	16	8 ... 128	20
SAFER+	saferp_desc	16	16, 24, 32	8, 12, 16
AES	aes_desc	16	16, 24, 32	10, 12, 14
	aes_enc_desc	16	16, 24, 32	10, 12, 14
Twofish	twofish_desc	16	16, 24, 32	16
DES	des_desc	8	8	16
3DES (EDE mode)	des3_desc	8	16, 24	16
CAST5 (CAST-128)	cast5_desc	8	5 ... 16	12, 16
Noekeon	noekeon_desc	16	16	16
Skipjack	skipjack_desc	8	10	32
Anubis	anubis_desc	16	16 ... 40	12 ... 18
Khazad	khazad_desc	8	16	8
SEED	kseed_desc	16	16	16
KASUMI	kasumi_desc	8	16	8

Figure 3.1: Built–In Software Ciphers

3.3.1 Notes

1. For AES, (also known as Rijndael) there are four descriptors which complicate issues a little. The descriptors rijndael_desc and rijndael_enc_desc provide the cipher named *rijndael*. The descriptors aes_desc and aes_enc_desc provide the cipher name *aes*. Functionally both *rijndael* and *aes* are the same cipher. The only difference is when you call find_cipher() you have to pass the correct name. The cipher descriptors with *enc* in the middle (e.g. rijndael_enc_desc) are related to an implementation of Rijndael with only the encryption routine and tables. The decryption and self–test function pointers of both *encrypt only* descriptors are set to **NULL** and should not be called.

   The *encrypt only* descriptors are useful for applications that only use the encryption function of the cipher. Algorithms such as EAX, PMAC and OMAC only require the encryption function. So far this *encrypt only* functionality has only been implemented for Rijndael as it makes the most sense for this cipher.
2. Note that for *DES* and *3DES* they use 8 and 24 byte keys but only 7 and 21 [respectively] bytes of the keys are in fact used for the purposes of encryption. My suggestion is just to use random 8/24 byte keys instead of trying to make a 8/24 byte string from the real 7/21 byte key.
3. Note that *Twofish* has additional configuration options (Figure 3.2) that take place at build time. These options are found in the file *tomcrypt_cfg.h*. The first option is *TWOFISH_SMALL* which when defined will force the Twofish code to not pre-compute the Twofish $g(X)$ function as a set of four 8×32 s-boxes. This means that a scheduled key will require less ram but the resulting cipher will be slower. The second option is *TWOFISH_TABLES* which when defined will force the Twofish code to use pre-computed tables for the two s-boxes q_0, q_1 as well as the multiplication by the polynomials 5B and EF used in the MDS multiplication. As a result the code is faster and slightly larger. The speed increase is useful when *TWOFISH_SMALL* is defined since the s-boxes and MDS multiply form the heart of the Twofish round function.

TWOFISH_SMALL	TWOFISH_TABLES	Speed and Memory (per key)
undefined	undefined	Very fast, 4.2KB of ram.
undefined	defined	Faster key setup, larger code.
defined	undefined	Very slow, 0.2KB of ram.
defined	defined	Faster, 0.2KB of ram, larger code.

Figure 3.2: Twofish Build Options

To work with the cipher_descriptor array there is a function:

```
int find_cipher(char *name)
```

Which will search for a given name in the array. It returns -1 if the cipher is not found, otherwise it returns the location in the array where the cipher was found. For example, to indirectly setup Blowfish you can also use:

```
#include <tomcrypt.h>
int main(void)
{
   unsigned char key[8];
   symmetric_key skey;
   int err;

   /* you must register a cipher before you use it */
```

```
   if (register_cipher(&blowfish_desc)) == -1) {
      printf("Unable to register Blowfish cipher.");
      return -1;
   }

   /* generic call to function (assuming the key
    * in key[] was already setup) */
   if ((err =
        cipher_descriptor[find_cipher("blowfish")].
          setup(key, 8, 0, &skey)) != CRYPT_OK) {
      printf("Error setting up Blowfish: %s\n", error_to_string(err));
      return -1;
   }

   /* ... use cipher ... */
}
```

A good safety would be to check the return value of *find_cipher()* before accessing the desired function. In order to use a cipher with the descriptor table you must register it first using:

```
int register_cipher(const struct _cipher_descriptor *cipher);
```

Which accepts a pointer to a descriptor and returns the index into the global descriptor table. If an error occurs such as there is no more room (it can have 32 ciphers at most) it will return **-1**. If you try to add the same cipher more than once it will just return the index of the first copy. To remove a cipher call:

```
int unregister_cipher(const struct _cipher_descriptor *cipher);
```

Which returns **CRYPT_OK** if it removes the cipher, otherwise it returns **CRYPT_ERROR**.

```
#include <tomcrypt.h>
int main(void)
{
   int err;

   /* register the cipher */
   if (register_cipher(&rijndael_desc) == -1) {
      printf("Error registering Rijndael\n");
      return -1;
   }

   /* use Rijndael */

   /* remove it */
   if ((err = unregister_cipher(&rijndael_desc)) != CRYPT_OK) {
      printf("Error removing Rijndael: %s\n", error_to_string(err));
      return -1;
   }

   return 0;
}
```

This snippet is a small program that registers Rijndael.

3.4 Symmetric Modes of Operations

3.4.1 Background

A typical symmetric block cipher can be used in chaining modes to effectively encrypt messages larger than the block size of the cipher. Given a key k, a plaintext P and a cipher E we shall denote the encryption of the block P under the key k as $E_k(P)$. In some modes there exists an initial vector denoted as C_{-1}.

ECB Mode

ECB or Electronic Codebook Mode is the simplest method to use. It is given as:

$$C_i = E_k(P_i) \tag{3.1}$$

This mode is very weak since it allows people to swap blocks and perform replay attacks if the same key is used more than once.

CBC Mode

CBC or Cipher Block Chaining mode is a simple mode designed to prevent trivial forms of replay and swap attacks on ciphers. It is given as:

$$C_i = E_k(P_i \oplus C_{i-1}) \tag{3.2}$$

It is important that the initial vector be unique and preferably random for each message encrypted under the same key.

CTR Mode

CTR or Counter Mode is a mode which only uses the encryption function of the cipher. Given a initial vector which is treated as a large binary counter the CTR mode is given as:

$$\begin{aligned} C_{-1} &= C_{-1} + 1 \pmod{2^W} \\ C_i &= P_i \oplus E_k(C_{-1}) \end{aligned} \tag{3.3}$$

Where W is the size of a block in bits (e.g. 64 for Blowfish). As long as the initial vector is random for each message encrypted under the same key replay and swap attacks are infeasible. CTR mode may look simple but it is as secure as the block cipher is under a chosen plaintext attack (provided the initial vector is unique).

CFB Mode

CFB or Ciphertext Feedback Mode is a mode akin to CBC. It is given as:

$$\begin{aligned} C_i &= P_i \oplus C_{-1} \\ C_{-1} &= E_k(C_i) \end{aligned} \tag{3.4}$$

Note that in this library the output feedback width is equal to the size of the block cipher. That is this mode is used to encrypt whole blocks at a time. However, the library will buffer data allowing the user to encrypt or decrypt partial blocks without a delay. When this mode is first setup it will initially encrypt the initial vector as required.

OFB Mode

OFB or Output Feedback Mode is a mode akin to CBC as well. It is given as:

$$\begin{aligned} C_{-1} &= E_k(C_{-1}) \\ C_i &= P_i \oplus C_{-1} \end{aligned} \tag{3.5}$$

Like the CFB mode the output width in CFB mode is the same as the width of the block cipher. OFB mode will also buffer the output which will allow you to encrypt or decrypt partial blocks without delay.

3.4.2 Choice of Mode

My personal preference is for the CTR mode since it has several key benefits:

1. No short cycles which is possible in the OFB and CFB modes.
2. Provably as secure as the block cipher being used under a chosen plaintext attack.
3. Technically does not require the decryption routine of the cipher.
4. Allows random access to the plaintext.
5. Allows the encryption of block sizes that are not equal to the size of the block cipher.

The CTR, CFB and OFB routines provided allow you to encrypt block sizes that differ from the ciphers block size. They accomplish this by buffering the data required to complete a block. This allows you to encrypt or decrypt any size block of memory with either of the three modes.

The ECB and CBC modes process blocks of the same size as the cipher at a time. Therefore, they are less flexible than the other modes.

3.4.3 Ciphertext Stealing

Ciphertext stealing is a method of dealing with messages in CBC mode which are not a multiple of the block length. This is accomplished by encrypting the last ciphertext block in ECB mode, and XOR'ing the output against the last partial block of plaintext. LibTomCrypt does not support this mode directly but it is fairly easy to emulate with a call to the cipher's ecb_encrypt() callback function.

The more sane way to deal with partial blocks is to pad them with zeroes, and then use CBC normally.

3.4.4 Initialization

The library provides simple support routines for handling CBC, CTR, CFB, OFB and ECB encoded messages. Assuming the mode you want is XXX there is a structure called *symmetric_XXX* that will contain the information required to use that mode. They have identical setup routines (except CTR and ECB mode):

```
int XXX_start(                int  cipher,
              const unsigned char *IV,
              const unsigned char *key,
                              int  keylen,
                              int  num_rounds,
                    symmetric_XXX *XXX);

int ctr_start(                int  cipher,
              const unsigned char *IV,
              const unsigned char *key,
                              int  keylen,
                              int  num_rounds,
                              int  ctr_mode,
                    symmetric_CTR *ctr);

int ecb_start(                int  cipher,
              const unsigned char *key,
                              int  keylen,
                              int  num_rounds,
                    symmetric_ECB *ecb);
```

In each case, *cipher* is the index into the cipher_descriptor array of the cipher you want to use. The *IV* value is the initialization vector to be used with the cipher. You must fill the IV yourself and it is assumed they are the same length as the block size[5] of the cipher you choose. It is important that the IV be random for each unique message you want to encrypt. The parameters *key*, *keylen* and *num_rounds* are the same as in the XXX_setup() function call. The final parameter is a pointer to the structure you want to hold the information for the mode of operation.

The routines return **CRYPT_OK** if the cipher initialized correctly, otherwise, they return an error code.

CTR Mode

In the case of CTR mode there is an additional parameter *ctr_mode* which specifies the mode that the counter is to be used in. If **CTR_COUNTER_ LITTLE_ENDIAN** was specified then the counter will be treated as a little endian value. Otherwise, if **CTR_COUNTER_BIG_ENDIAN** was specified the counter will be treated as a big endian value. As of v1.15 the RFC 3686 style of increment then encrypt is also supported. By OR'ing **LTC_CTR_RFC3686** with the CTR *mode* value, ctr_start() will increment the counter before encrypting it for the first time.

As of V1.17, the library supports variable length counters for CTR mode. The (optional) counter length is specified by OR'ing the octet length of the counter against the *ctr_mode* parameter. The default, zero, indicates that a full block length counter will be used. This also ensures backwards compatibility with software that uses older versions of the library.

```
symmetric_CTR ctr;
int           err;
unsigned char IV[16], key[16];
```

[5] In other words the size of a block of plaintext for the cipher, e.g. 8 for DES, 16 for AES, etc.

```
/* use a 32-bit little endian counter */
if ((err = ctr_start(find_cipher("aes"),
                     IV, key, 16, 0,
                     CTR_COUNTER_LITTLE_ENDIAN | 4,
                     &ctr)) != CRYPT_OK) {
   handle_error(err);
}
```

Changing the counter size has little (really no) effect on the performance of the CTR chaining mode. It is provided for compatibility with other software (and hardware) which have smaller fixed sized counters.

3.4.5 Encryption and Decryption

To actually encrypt or decrypt the following routines are provided:

```
int XXX_encrypt(const unsigned char *pt,
                      unsigned char *ct,
                      unsigned long  len,
                      symmetric_YYY *YYY);

int XXX_decrypt(const unsigned char *ct,
                      unsigned char *pt,
                      unsigned long  len,
                      symmetric_YYY *YYY);
```

Where XXX is one of $\{ecb, cbc, ctr, cfb, ofb\}$.

In all cases, *len* is the size of the buffer (as number of octets) to encrypt or decrypt. The CTR, OFB and CFB modes are order sensitive but not chunk sensitive. That is you can encrypt $ABCDEF$ in three calls like AB, CD, EF or two like $ABCDE$ and F and end up with the same ciphertext. However, encrypting ABC and $DABC$ will result in different ciphertexts. All five of the modes will return **CRYPT_OK** on success from the encrypt or decrypt functions.

In the ECB and CBC cases, *len* must be a multiple of the ciphers block size. In the CBC case, you must manually pad the end of your message (either with zeroes or with whatever your protocol requires).

To decrypt in either mode, perform the setup like before (recall you have to fetch the IV value you used), and use the decrypt routine on all of the blocks.

3.4.6 IV Manipulation

To change or read the IV of a previously initialized chaining mode use the following two functions.

```
int XXX_getiv(unsigned char *IV,
              unsigned long *len,
              symmetric_XXX *XXX);
```

```
int XXX_setiv(const unsigned char *IV,
                    unsigned long  len,
                    symmetric_XXX *XXX);
```

The XXX_getiv() functions will read the IV out of the chaining mode and store it into *IV* along with the length of the IV stored in *len*. The XXX_setiv will initialize the chaining mode state as if the original IV were the new IV specified. The length of the IV passed in must be the size of the ciphers block size.

The XXX_setiv() functions are handy if you wish to change the IV without re–keying the cipher.

What the *setiv* function will do depends on the mode being changed. In CBC mode, the new IV replaces the existing IV as if it were the last ciphertext block. In CFB mode, the IV is encrypted as if it were the prior encrypted pad. In CTR mode, the IV is encrypted without first incrementing it (regardless of the LTC_RFC_3686 flag presence). In F8 mode, the IV is encrypted and becomes the new pad. It does not change the salted IV, and is only meant to allow seeking within a session. In LRW, it changes the tweak, forcing a computation of the tweak pad, allowing for seeking within the session. In OFB mode, the IV is encrypted and becomes the new pad.

3.4.7 Stream Termination

To terminate an open stream call the done function.

```
int XXX_done(symmetric_XXX *XXX);
```

This will terminate the stream (by terminating the cipher) and return **CRYPT_OK** if successful.

3.4.8 Examples

```
#include <tomcrypt.h>
int main(void)
{
   unsigned char key[16], IV[16], buffer[512];
   symmetric_CTR ctr;
   int x, err;

   /* register twofish first */
   if (register_cipher(&twofish_desc) == -1) {
      printf("Error registering cipher.\n");
      return -1;
   }

   /* somehow fill out key and IV */

   /* start up CTR mode */
   if ((err = ctr_start(
        find_cipher("twofish"), /* index of desired cipher */
                             IV, /* the initial vector */
                            key, /* the secret key */
                             16, /* length of secret key (16 bytes) */
                              0, /* 0 == default # of rounds */
     CTR_COUNTER_LITTLE_ENDIAN, /* Little endian counter */
                           &ctr)  /* where to store the CTR state */
      ) != CRYPT_OK) {
      printf("ctr_start error: %s\n", error_to_string(err));
      return -1;
   }

   /* somehow fill buffer than encrypt it */
   if ((err = ctr_encrypt(        buffer, /* plaintext */
                                  buffer, /* ciphertext */
                          sizeof(buffer), /* length of plaintext pt */
                                   &ctr)  /* CTR state */
      ) != CRYPT_OK) {
      printf("ctr_encrypt error: %s\n", error_to_string(err));
      return -1;
   }

   /* make use of ciphertext... */

   /* now we want to decrypt so let's use ctr_setiv */
   if ((err = ctr_setiv(  IV, /* the initial IV we gave to ctr_start */
                          16, /* the IV is 16 bytes long */
                         &ctr) /* the ctr state we wish to modify */
       ) != CRYPT_OK) {
      printf("ctr_setiv error: %s\n", error_to_string(err));
      return -1;
   }
```

```
   if ((err = ctr_decrypt(        buffer, /* ciphertext */
                                  buffer, /* plaintext */
                          sizeof(buffer), /* length of plaintext */
                                   &ctr)  /* CTR state */
      ) != CRYPT_OK) {
      printf("ctr_decrypt error: %s\n", error_to_string(err));
      return -1;
   }

   /* terminate the stream */
   if ((err = ctr_done(&ctr)) != CRYPT_OK) {
      printf("ctr_done error: %s\n", error_to_string(err));
      return -1;
   }

   /* clear up and return */
   zeromem(key, sizeof(key));
   zeromem(&ctr, sizeof(ctr));

   return 0;
}
```

3.4.9 LRW Mode

LRW mode is a cipher mode which is meant for indexed encryption like used to handle storage media. It is meant to have efficient seeking and overcome the security problems of ECB mode while not increasing the storage requirements. It is used much like any other chaining mode except with two key differences.

The key is specified as two strings the first key K_1 is the (normally AES) key and can be any length (typically 16, 24 or 32 octets long). The second key K_2 is the *tweak* key and is always 16 octets long. The tweak value is **NOT** a nonce or IV value it must be random and secret.

To initialize LRW mode use:

```
int lrw_start(                int  cipher,
              const unsigned char *IV,
              const unsigned char *key,
                              int  keylen,
              const unsigned char *tweak,
                              int  num_rounds,
                    symmetric_LRW *lrw);
```

This will initialize the LRW context with the given (16 octet) *IV*, cipher K_1 *key* of length *keylen* octets and the (16 octet) K_2 *tweak*. While LRW was specified to be used only with AES, LibTomCrypt will allow any 128–bit block cipher to be specified as indexed by *cipher*. The number of rounds for the block cipher *num_rounds* can be 0 to use the default number of rounds for the given cipher.

To process data use the following functions:

```
int lrw_encrypt(const unsigned char *pt,
                      unsigned char *ct,
                      unsigned long  len,
                      symmetric_LRW *lrw);

int lrw_decrypt(const unsigned char *ct,
                      unsigned char *pt,
                      unsigned long  len,
                      symmetric_LRW *lrw);
```

These will encrypt (or decrypt) the plaintext to the ciphertext buffer (or vice versa). The length is specified by *len* in octets but must be a multiple of 16. The LRW code uses a fast tweak update such that consecutive blocks are encrypted faster than if random seeking where used.

To manipulate the IV use the following functions:

```
int lrw_getiv(unsigned char *IV,
              unsigned long *len,
              symmetric_LRW *lrw);

int lrw_setiv(const unsigned char *IV,
                    unsigned long  len,
                    symmetric_LRW *lrw);
```

These will get or set the 16–octet IV. Note that setting the IV is the same as *seeking* and unlike other modes is not a free operation. It requires updating the entire tweak which is slower than sequential use. Avoid seeking excessively in performance constrained code.

To terminate the LRW state use the following:

```
int lrw_done(symmetric_LRW *lrw);
```

3.4.10 XTS Mode

As of v1.17, LibTomCrypt supports XTS mode with code donated by Elliptic Semiconductor Inc.[6]. XTS is a chaining mode for 128–bit block ciphers, recommended by IEEE (P1619) for disk encryption. It is meant to be an encryption mode with random access to the message data without compromising privacy. It requires two private keys (of equal length) to perform the encryption process. Each encryption invocation includes a sector number or unique identifier specified as a 128–bit string.

To initialize XTS mode use the following function call:

```
int xts_start(                int  cipher,
              const unsigned char *key1,
              const unsigned char *key2,
                    unsigned long  keylen,
                               int  num_rounds,
                    symmetric_xts *xts)
```

[6] www.ellipticsemi.com

This will start the XTS mode with the two keys pointed to by *key1* and *key2* of length *keylen* octets each.

To encrypt or decrypt a sector use the following calls:

```
int xts_encrypt(
   const unsigned char *pt, unsigned long ptlen,
         unsigned char *ct,
   const unsigned char *tweak,
         symmetric_xts *xts);

int xts_decrypt(
   const unsigned char *ct, unsigned long ptlen,
         unsigned char *pt,
   const unsigned char *tweak,
         symmetric_xts *xts);
```

The first will encrypt the plaintext pointed to by *pt* of length *ptlen* octets, and store the ciphertext in the array pointed to by *ct*. It uses the 128–bit tweak pointed to by *tweak* to encrypt the block. The decrypt function performs the opposite operation. Both functions support ciphertext stealing (blocks that are not multiples of 16 bytes).

The P1619 specification states the tweak for sector number shall be represented as a 128–bit little endian string.

To terminate the XTS state call the following function:

```
void xts_done(symmetric_xts *xts);
```

3.4.11 F8 Mode

The F8 Chaining mode (see RFC 3711 for instance) is yet another chaining mode for block ciphers. It behaves much like CTR mode in that it XORs a keystream against the plaintext to encrypt. F8 mode comes with the additional twist that the counter value is secret, encrypted by a *salt key*. We initialize F8 mode with the following function call:

```
int f8_start(                int  cipher,
             const unsigned char *IV,
             const unsigned char *key,
                              int  keylen,
             const unsigned char *salt_key,
                              int  skeylen,
                              int  num_rounds,
                    symmetric_F8 *f8);
```

This will start the F8 mode state using *key* as the secret key, *IV* as the counter. It uses the *salt_key* as IV encryption key (m in the RFC 3711). The salt_key can be shorter than the secret key but it should not be longer.

To encrypt or decrypt data we use the following two functions:

```
int f8_encrypt(const unsigned char *pt,
                     unsigned char *ct,
                     unsigned long  len,
                      symmetric_F8 *f8);

int f8_decrypt(const unsigned char *ct,
                     unsigned char *pt,
                     unsigned long  len,
                      symmetric_F8 *f8);
```

These will encrypt or decrypt a variable length array of bytes using the F8 mode state specified. The length is specified in bytes and does not have to be a multiple of the ciphers block size.

To change or retrieve the current counter IV value use the following functions:

```
int f8_getiv(unsigned char *IV,
             unsigned long *len,
              symmetric_F8 *f8);

int f8_setiv(const unsigned char *IV,
                   unsigned long  len,
                    symmetric_F8 *f8);
```

These work with the current IV value only and not the encrypted IV value specified during the call to f8_start(). The purpose of these two functions is to be able to seek within a current session only. If you want to change the session IV you will have to call f8_done() and then start a new state with f8_start().

To terminate an F8 state call the following function:

```
int f8_done(symmetric_F8 *f8);
```

3.5 Encrypt and Authenticate Modes

3.5.1 EAX Mode

LibTomCrypt provides support for a mode called EAX[7] in a manner similar to the way it was intended to be used by the designers. First, a short description of what EAX mode is before we explain how to use it. EAX is a mode that requires a cipher, CTR and OMAC support and provides encryption and authentication[8]. It is initialized with a random *nonce* that can be shared publicly, a *header* which can be fixed and public, and a random secret symmetric key.

The *header* data is meant to be meta–data associated with a stream that isn't private (e.g., protocol messages). It can be added at anytime during an EAX stream, and is part of the authentication tag. That is, changes in the meta-data can be detected by changes in the output tag.

The mode can then process plaintext producing ciphertext as well as compute a partial checksum. The actual checksum called a *tag* is only emitted when the message is finished. In the interim, the user can process any arbitrary sized message block to send to the recipient as ciphertext. This makes the EAX mode especially suited for streaming modes of operation.

The mode is initialized with the following function.

```
int eax_init(          eax_state *eax,
                             int  cipher,
             const unsigned char *key,
                   unsigned long  keylen,
             const unsigned char *nonce,
                   unsigned long  noncelen,
             const unsigned char *header,
                   unsigned long  headerlen);
```

Where *eax* is the EAX state. The *cipher* parameter is the index of the desired cipher in the descriptor table. The *key* parameter is the shared secret symmetric key of length *keylen* octets. The *nonce* parameter is the random public string of length *noncelen* octets. The *header* parameter is the random (or fixed or **NULL**) header for the message of length *headerlen* octets.

When this function completes, the *eax* state will be initialized such that you can now either have data decrypted or encrypted in EAX mode. Note: if *headerlen* is zero you may pass *header* as **NULL** to indicate there is no initial header data.

To encrypt or decrypt data in a streaming mode use the following.

```
int eax_encrypt(          eax_state *eax,
                const unsigned char *pt,
                      unsigned char *ct,
                      unsigned long  length);

int eax_decrypt(          eax_state *eax,
                const unsigned char *ct,
                      unsigned char *pt,
                      unsigned long  length);
```

[7] See M. Bellare, P. Rogaway, D. Wagner, A Conventional Authenticated-Encryption Mode.

[8] Note that since EAX only requires OMAC and CTR you may use *encrypt only* cipher descriptors with this mode.

The function *eax_encrypt* will encrypt the bytes in *pt* of *length* octets, and store the ciphertext in *ct*. Note: *ct* and *pt* may be the same region in memory. This function will also send the ciphertext through the OMAC function. The function *eax_decrypt* decrypts *ct*, and stores it in *pt*. This also allows *pt* and *ct* to be the same region in memory.

You cannot both encrypt or decrypt with the same *eax* context. For bi–directional communication you will need to initialize two EAX contexts (preferably with different headers and nonces).

Note: both of these functions allow you to send the data in any granularity but the order is important. While the eax_init() function allows you to add initial header data to the stream you can also add header data during the EAX stream with the following.

```
int eax_addheader(          eax_state *eax,
                  const unsigned char *header,
                        unsigned long  length);
```

This will add the *length* octet from *header* to the given *eax* header. Once the message is finished, the *tag* (checksum) may be computed with the following function:

```
int eax_done(    eax_state *eax,
             unsigned char *tag,
             unsigned long *taglen);
```

This will terminate the EAX state *eax*, and store up to *taglen* bytes of the message tag in *tag*. The function then stores how many bytes of the tag were written out back in to *taglen*.

The EAX mode code can be tested to ensure it matches the test vectors by calling the following function:

```
int eax_test(void);
```

This requires that the AES (or Rijndael) block cipher be registered with the cipher_descriptor table first.

```
#include <tomcrypt.h>
int main(void)
{
   int           err;
   eax_state     eax;
   unsigned char pt[64], ct[64], nonce[16], key[16], tag[16];
   unsigned long taglen;

   if (register_cipher(&rijndael_desc) == -1) {
      printf("Error registering Rijndael");
      return EXIT_FAILURE;
   }

   /* ... make up random nonce and key ... */

   /* initialize context */
   if ((err = eax_init(            &eax,  /* context */
```

```
                find_cipher("rijndael"),  /* cipher id */
                                  nonce,  /* the nonce */
                                     16,  /* nonce is 16 bytes */
                                "TestApp",  /* example header */
                                      7)  /* header length */
       ) != CRYPT_OK) {
      printf("Error eax_init: %s", error_to_string(err));
      return EXIT_FAILURE;
   }

   /* now encrypt data, say in a loop or whatever */
   if ((err = eax_encrypt(     &eax, /* eax context */
                                 pt, /* plaintext  (source) */
                                 ct, /* ciphertext (destination) */
                           sizeof(pt) /* size of plaintext */
      ) != CRYPT_OK) {
      printf("Error eax_encrypt: %s", error_to_string(err));
      return EXIT_FAILURE;
   }

   /* finish message and get authentication tag */
   taglen = sizeof(tag);
   if ((err = eax_done(   &eax,      /* eax context */
                           tag,      /* where to put tag */
                       &taglen       /* length of tag space */
      ) != CRYPT_OK) {
      printf("Error eax_done: %s", error_to_string(err));
      return EXIT_FAILURE;
   }

   /* now we have the authentication tag in "tag" and
    * it's taglen bytes long */
}
```

You can also perform an entire EAX state on a block of memory in a single function call with the following functions.

```
int eax_encrypt_authenticate_memory(
                     int  cipher,
    const unsigned char *key,    unsigned long keylen,
    const unsigned char *nonce,  unsigned long noncelen,
    const unsigned char *header, unsigned long headerlen,
    const unsigned char *pt,     unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,    unsigned long *taglen);

int eax_decrypt_verify_memory(
```

```
                  int  cipher,
    const unsigned char *key,    unsigned long keylen,
    const unsigned char *nonce,  unsigned long noncelen,
    const unsigned char *header, unsigned long headerlen,
    const unsigned char *ct,     unsigned long ctlen,
          unsigned char *pt,
          unsigned char *tag,    unsigned long taglen,
          int           *res);
```

Both essentially just call eax_init() followed by eax_encrypt() (or eax_decrypt() respectively) and eax_done(). The parameters have the same meaning as with those respective functions.

The only difference is eax_decrypt_verify_memory() does not emit a tag. Instead you pass it a tag as input and it compares it against the tag it computed while decrypting the message. If the tags match then it stores a 1 in *res*, otherwise it stores a 0.

3.5.2 OCB Mode

LibTomCrypt provides support for a mode called OCB[9] . OCB is an encryption protocol that simultaneously provides authentication. It is slightly faster to use than EAX mode but is less flexible. Let's review how to initialize an OCB context.

```
int ocb_init(          ocb_state *ocb,
                             int  cipher,
             const unsigned char *key,
                   unsigned long  keylen,
             const unsigned char *nonce);
```

This will initialize the *ocb* context using cipher descriptor *cipher*. It will use a *key* of length *keylen* and the random *nonce*. Note that *nonce* must be a random (public) string the same length as the block ciphers block size (e.g. 16 bytes for AES).

This mode has no *Associated Data* like EAX mode does which means you cannot authenticate metadata along with the stream. To encrypt or decrypt data use the following.

```
int ocb_encrypt(          ocb_state *ocb,
                const unsigned char *pt,
                      unsigned char *ct);

int ocb_decrypt(          ocb_state *ocb,
                const unsigned char *ct,
                      unsigned char *pt);
```

This will encrypt (or decrypt for the latter) a fixed length of data from *pt* to *ct* (vice versa for the latter). They assume that *pt* and *ct* are the same size as the block cipher's block size. Note that you cannot call both functions given a single *ocb* state. For bi-directional communication you will have to initialize two *ocb* states (with different nonces). Also *pt* and *ct* may point to the same location in memory.

[9]See P. Rogaway, M. Bellare, J. Black, T. Krovetz, *OCB: A Block Cipher Mode of Operation for Efficient Authenticated Encryption*.

State Termination

When you are finished encrypting the message you call the following function to compute the tag.

```
int ocb_done_encrypt(          ocb_state *ocb,
                       const unsigned char *pt,
                             unsigned long  ptlen,
                             unsigned char *ct,
                             unsigned char *tag,
                             unsigned long *taglen);
```

This will terminate an encrypt stream *ocb*. If you have trailing bytes of plaintext that will not complete a block you can pass them here. This will also encrypt the *ptlen* bytes in *pt* and store them in *ct*. It will also store up to *taglen* bytes of the tag into *tag*.

Note that *ptlen* must be less than or equal to the block size of block cipher chosen. Also note that if you have an input message equal to the length of the block size then you pass the data here (not to ocb_encrypt()) only.

To terminate a decrypt stream and compared the tag you call the following.

```
int ocb_done_decrypt(          ocb_state *ocb,
                       const unsigned char *ct,
                             unsigned long  ctlen,
                             unsigned char *pt,
                       const unsigned char *tag,
                             unsigned long  taglen,
                                       int *res);
```

Similarly to the previous function you can pass trailing message bytes into this function. This will compute the tag of the message (internally) and then compare it against the *taglen* bytes of *tag* provided. By default *res* is set to zero. If all *taglen* bytes of *tag* can be verified then *res* is set to one (authenticated message).

Packet Functions

To make life simpler the following two functions are provided for memory bound OCB.

```
int ocb_encrypt_authenticate_memory(
                     int  cipher,
    const unsigned char *key,    unsigned long keylen,
    const unsigned char *nonce,
    const unsigned char *pt,     unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,    unsigned long *taglen);
```

This will OCB encrypt the message *pt* of length *ptlen*, and store the ciphertext in *ct*. The length *ptlen* can be any arbitrary length.

```
int ocb_decrypt_verify_memory(
                   int  cipher,
    const unsigned char *key,    unsigned long keylen,
    const unsigned char *nonce,
    const unsigned char *ct,     unsigned long ctlen,
          unsigned char *pt,
    const unsigned char *tag,    unsigned long taglen,
          int           *res);
```

Similarly, this will OCB decrypt, and compare the internally computed tag against the tag provided. *res* is set appropriately.

3.5.3 CCM Mode

CCM is a NIST proposal for encrypt + authenticate that is centered around using AES (or any 16–byte cipher) as a primitive. Unlike EAX and OCB mode, it is only meant for *packet* mode where the length of the input is known in advance. Since it is a packet mode function, CCM only has one function that performs the protocol.

```
int ccm_memory(
                   int  cipher,
    const unsigned char *key,    unsigned long keylen,
    symmetric_key       *uskey,
    const unsigned char *nonce,  unsigned long noncelen,
    const unsigned char *header, unsigned long headerlen,
          unsigned char *pt,     unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,    unsigned long *taglen,
                   int  direction);
```

This performs the *CCM* operation on the data. The *cipher* variable indicates which cipher in the descriptor table to use. It must have a 16–byte block size for CCM.

The key can be specified in one of two fashions. First, it can be passed as an array of octets in *key* of length *keylen*. Alternatively, it can be passed in as a previously scheduled key in *uskey*. The latter fashion saves time when the same key is used for multiple packets. If *uskey* is not **NULL**, then *key* may be **NULL** (and vice-versa).

The nonce or salt is *nonce* of length *noncelen* octets. The header is meta–data you want to send with the message but not have encrypted, it is stored in *header* of length *headerlen* octets. The header can be zero octets long (if $headerlen = 0$ then you can pass *header* as **NULL**).

The plaintext is stored in *pt*, and the ciphertext in *ct*. The length of both are expected to be equal and is passed in as *ptlen*. It is allowable that $pt = ct$. The *direction* variable indicates whether encryption (direction = **CCM_ENCRYPT**) or decryption (direction = **CCM_DECRYPT**) is to be performed.

As implemented, this version of CCM cannot handle header or plaintext data longer than $2^{32} - 1$ octets long.

You can test the implementation of CCM with the following function.

```
int ccm_test(void);
```

This will return **CRYPT_OK** if the CCM routine passes known test vectors. It requires AES or Rijndael to be registered previously, otherwise it will return **CRYPT_NOP**.

CCM Example

The following is a sample of how to call CCM.

```
#include <tomcrypt.h>
int main(void)
{
   unsigned char key[16], nonce[12], pt[32], ct[32],
                 tag[16], tagcp[16];
   unsigned long taglen;
   int           err;

   /* register cipher */
   register_cipher(&aes_desc);

   /* somehow fill key, nonce, pt */

   /* encrypt it */
   taglen = sizeof(tag);
   if ((err =
       ccm_memory(find_cipher("aes"),
                  key, 16,    /* 128-bit key */
                  NULL,       /* not prescheduled */
                  nonce, 12,  /* 96-bit nonce */
                  NULL, 0,    /* no header */
                  pt, 32,     /* [in]  32-byte plaintext */
                  ct,         /* [out] ciphertext */
                  tag, &taglen,
                  CCM_ENCRYPT)) != CRYPT_OK) {
       printf("ccm_memory error %s\n", error_to_string(err));
       return -1;
   }
   /* ct[0..31] and tag[0..15] now hold the output */

   /* decrypt it */
   taglen = sizeof(tagcp);
   if ((err =
       ccm_memory(find_cipher("aes"),
                  key, 16,    /* 128-bit key */
                  NULL,       /* not prescheduled */
                  nonce, 12,  /* 96-bit nonce */
                  NULL, 0,    /* no header */
                  pt, 32,     /* [out] 32-byte plaintext */
                  ct,         /* [in]  ciphertext */
                  tagcp, &taglen,
                  CCM_DECRYPT)) != CRYPT_OK) {
       printf("ccm_memory error %s\n", error_to_string(err));
```

```
      return -1;
   }

   /* now pt[0..31] should hold the original plaintext,
      tagcp[0..15] and tag[0..15] should have the same contents */
}
```

3.5.4 GCM Mode

Galois counter mode is an IEEE proposal for authenticated encryption (also it is a planned NIST standard). Like EAX and OCB mode, it can be used in a streaming capacity however, unlike EAX it cannot accept *additional authentication data* (meta–data) after plaintext has been processed. This mode also only works with block ciphers with a 16–byte block.

A GCM stream is meant to be processed in three modes, one after another. First, the initial vector (per session) data is processed. This should be unique to every session. Next, the the optional additional authentication data is processed, and finally the plaintext (or ciphertext depending on the direction).

Initialization

To initialize the GCM context with a secret key call the following function.

```
int gcm_init(          gcm_state *gcm,
                             int  cipher,
             const unsigned char *key,
                             int  keylen);
```

This initializes the GCM state *gcm* for the given cipher indexed by *cipher*, with a secret key *key* of length *keylen* octets. The cipher chosen must have a 16–byte block size (e.g., AES).

Initial Vector

After the state has been initialized (or reset) the next step is to add the session (or packet) initial vector. It should be unique per packet encrypted.

```
int gcm_add_iv(          gcm_state *gcm,
               const unsigned char *IV,
                     unsigned long  IVlen);
```

This adds the initial vector octets from *IV* of length *IVlen* to the GCM state *gcm*. You can call this function as many times as required to process the entire IV.

Note: the GCM protocols provides a *shortcut* for 12–byte IVs where no pre-processing is to be done. If you want to minimize per packet latency it is ideal to only use 12–byte IVs. You can just increment it like a counter for each packet.

Additional Authentication Data

After the entire IV has been processed, the additional authentication data can be processed. Unlike the IV, a packet/session does not require additional authentication data (AAD) for security. The AAD is meant to be used as side–channel data you want to be authenticated with the packet. Note: once you begin adding AAD to the GCM state you cannot return to adding IV data until the state has been reset.

```
int gcm_add_aad(          gcm_state *gcm,
                const unsigned char *adata,
                      unsigned long  adatalen);
```

This adds the additional authentication data *adata* of length *adatalen* to the GCM state *gcm*.

Plaintext Processing

After the AAD has been processed, the plaintext (or ciphertext depending on the direction) can be processed.

```
int gcm_process(    gcm_state *gcm,
                unsigned char *pt,
                unsigned long  ptlen,
                unsigned char *ct,
                          int  direction);
```

This processes message data where *pt* is the plaintext and *ct* is the ciphertext. The length of both are equal and stored in *ptlen*. Depending on the mode *pt* is the input and *ct* is the output (or vice versa). When *direction* equals **GCM_ENCRYPT** the plaintext is read, encrypted and stored in the ciphertext buffer. When *direction* equals **GCM_DECRYPT** the opposite occurs.

State Termination

To terminate a GCM state and retrieve the message authentication tag call the following function.

```
int gcm_done(    gcm_state *gcm,
             unsigned char *tag,
             unsigned long *taglen);
```

This terminates the GCM state *gcm* and stores the tag in *tag* of length *taglen* octets.

State Reset

The call to gcm_init() will perform considerable pre–computation (when **GCM_TABLES** is defined) and if you're going to be dealing with a lot of packets it is very costly to have to call it repeatedly. To aid in this endeavour, the reset function has been provided.

```
int gcm_reset(gcm_state *gcm);
```

This will reset the GCM state *gcm* to the state that gcm_init() left it. The user would then call gcm_add_iv(), gcm_add_aad(), etc.

One–Shot Packet

To process a single packet under any given key the following helper function can be used.

```
int gcm_memory(
                     int  cipher,
    const unsigned char *key,
          unsigned long keylen,
    const unsigned char *IV,    unsigned long IVlen,
    const unsigned char *adata, unsigned long adatalen,
          unsigned char *pt,    unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,   unsigned long *taglen,
                     int  direction);
```

This will initialize the GCM state with the given key, IV and AAD value then proceed to encrypt or decrypt the message text and store the final message tag. The definition of the variables is the same as it is for all the manual functions.

If you are processing many packets under the same key you shouldn't use this function as it invokes the pre–computation with each call.

Example Usage

The following is an example usage of how to use GCM over multiple packets with a shared secret key.

```
#include <tomcrypt.h>

int send_packet(const unsigned char *pt,  unsigned long ptlen,
                const unsigned char *iv,  unsigned long ivlen,
                const unsigned char *aad, unsigned long aadlen,
                      gcm_state     *gcm)
{
   int           err;
   unsigned long taglen;
   unsigned char tag[16];

   /* reset the state */
   if ((err = gcm_reset(gcm)) != CRYPT_OK) {
      return err;
   }

   /* Add the IV */
   if ((err = gcm_add_iv(gcm, iv, ivlen)) != CRYPT_OK) {
      return err;
   }

   /* Add the AAD (note: aad can be NULL if aadlen == 0) */
   if ((err = gcm_add_aad(gcm, aad, aadlen)) != CRYPT_OK) {
      return err;
```

```
   }

   /* process the plaintext */
   if ((err =
        gcm_process(gcm, pt, ptlen, pt, GCM_ENCRYPT)) != CRYPT_OK) {
      return err;
   }

   /* Finish up and get the MAC tag */
   taglen = sizeof(tag);
   if ((err = gcm_done(gcm, tag, &taglen)) != CRYPT_OK) {
      return err;
   }

   /* ... send a header describing the lengths ... */

   /* depending on the protocol and how IV is
    * generated you may have to send it too... */
   send(socket, iv, ivlen, 0);

   /* send the aad */
   send(socket, aad, aadlen, 0);

   /* send the ciphertext */
   send(socket, pt, ptlen, 0);

   /* send the tag */
   send(socket, tag, taglen, 0);

   return CRYPT_OK;
}

int main(void)
{
   gcm_state     gcm;
   unsigned char key[16], IV[12], pt[PACKET_SIZE];
   int           err, x;
   unsigned long ptlen;

   /* somehow fill key/IV with random values */

   /* register AES */
   register_cipher(&aes_desc);

   /* init the GCM state */
   if ((err =
        gcm_init(&gcm, find_cipher("aes"), key, 16)) != CRYPT_OK) {
      whine_and_pout(err);
   }
```

```
   /* handle us some packets */
   for (;;) {
       ptlen = make_packet_we_want_to_send(pt);

       /* use IV as counter (12 byte counter) */
       for (x = 11; x >= 0; x--) {
           if (++IV[x]) {
              break;
           }
       }

       if ((err = send_packet(pt, ptlen, iv, 12, NULL, 0, &gcm))
           != CRYPT_OK) {
           whine_and_pout(err);
       }
   }
   return EXIT_SUCCESS;
}
```

Chapter 4

One-Way Cryptographic Hash Functions

4.1 Core Functions

Like the ciphers, there are hash core functions and a universal data type to hold the hash state called *hash_state*. To initialize hash XXX (where XXX is the name) call:

```
void XXX_init(hash_state *md);
```

This simply sets up the hash to the default state governed by the specifications of the hash. To add data to the message being hashed call:

```
int XXX_process(         hash_state *md,
                const unsigned char *in,
                      unsigned long  inlen);
```

Essentially all hash messages are virtually infinitely[1] long message which are buffered. The data can be passed in any sized chunks as long as the order of the bytes are the same the message digest (hash output) will be the same. For example, this means that:

```
md5_process(&md, "hello ", 6);
md5_process(&md, "world", 5);
```

Will produce the same message digest as the single call:

```
md5_process(&md, "hello world", 11);
```

To finally get the message digest (the hash) call:

```
int XXX_done(   hash_state *md,
             unsigned char *out);
```

[1]Most hashes are limited to 2^{64} bits or 2,305,843,009,213,693,952 bytes.

This function will finish up the hash and store the result in the *out* array. You must ensure that *out* is long enough for the hash in question. Often hashes are used to get keys for symmetric ciphers so the *XXX_done()* functions will wipe the *md* variable before returning automatically.

To test a hash function call:

```
int XXX_test(void);
```

This will return **CRYPT_OK** if the hash matches the test vectors, otherwise it returns an error code. An example snippet that hashes a message with md5 is given below.

```
#include <tomcrypt.h>
int main(void)
{
    hash_state md;
    unsigned char *in = "hello world", out[16];

    /* setup the hash */
    md5_init(&md);

    /* add the message */
    md5_process(&md, in, strlen(in));

    /* get the hash in out[0..15] */
    md5_done(&md, out);

    return 0;
}
```

4.2 Hash Descriptors

Like the set of ciphers, the set of hashes have descriptors as well. They are stored in an array called *hash_descriptor* and are defined by:

```
struct _hash_descriptor {
    char *name;

    unsigned long hashsize;    /* digest output size in bytes  */
    unsigned long blocksize;   /* the block size the hash uses */

    void (*init)   (hash_state *hash);

    int  (*process)(         hash_state *hash,
                    const unsigned char *in,
                          unsigned long  inlen);

    int  (*done)   (hash_state *hash, unsigned char *out);

    int  (*test)   (void);
};
```

The *name* member is the name of the hash function (all lowercase). The *hashsize* member is the size of the digest output in bytes, while *blocksize* is the size of blocks the hash expects to the compression function. Technically, this detail is not important for high level developers but is useful to know for performance reasons.

The *init* member initializes the hash, *process* passes data through the hash, *done* terminates the hash and retrieves the digest. The *test* member tests the hash against the specified test vectors.

There is a function to search the array as well called *int find_hash(char *name)*. It returns -1 if the hash is not found, otherwise, the position in the descriptor table of the hash.

In addition, there is also find_hash_oid() which finds a hash by the ASN.1 OBJECT IDENTIFIER string.

```
int find_hash_oid(const unsigned long *ID, unsigned long IDlen);
```

You can use the table to indirectly call a hash function that is chosen at run-time. For example:

```
#include <tomcrypt.h>
int main(void)
{
   unsigned char buffer[100], hash[MAXBLOCKSIZE];
   int idx, x;
   hash_state md;

   /* register hashes .... */
   if (register_hash(&md5_desc) == -1) {
      printf("Error registering MD5.\n");
      return -1;
   }

   /* register other hashes ... */

   /* prompt for name and strip newline */
   printf("Enter hash name: \n");
   fgets(buffer, sizeof(buffer), stdin);
   buffer[strlen(buffer) - 1] = 0;

   /* get hash index */
   idx = find_hash(buffer);
   if (idx == -1) {
      printf("Invalid hash name!\n");
      return -1;
   }

   /* hash input until blank line */
   hash_descriptor[idx].init(&md);
   while (fgets(buffer, sizeof(buffer), stdin) != NULL)
         hash_descriptor[idx].process(&md, buffer, strlen(buffer));
   hash_descriptor[idx].done(&md, hash);

   /* dump to screen */
   for (x = 0; x < hash_descriptor[idx].hashsize; x++)
       printf("%02x ", hash[x]);
```

```
   printf("\n");
   return 0;
}
```

Note the usage of **MAXBLOCKSIZE**. In LibTomCrypt, no symmetric block, key or hash digest is larger than **MAXBLOCKSIZE** in length. This provides a simple size you can set your automatic arrays to that will not get overrun.

There are three helper functions to make working with hashes easier. The first is a function to hash a buffer, and produce the digest in a single function call.

```
int hash_memory(                int  hash,
                const unsigned char *in,
                      unsigned long  inlen,
                      unsigned char *out,
                      unsigned long *outlen);
```

This will hash the data pointed to by *in* of length *inlen*. The hash used is indexed by the *hash* parameter. The message digest is stored in *out*, and the *outlen* parameter is updated to hold the message digest size.

The next helper function allows for the hashing of a file based on a file name.

```
int hash_file(          int  hash,
                 const char *fname,
              unsigned char *out,
              unsigned long *outlen);
```

This will hash the file named by *fname* using the hash indexed by *hash*. The file named in this function call must be readable by the user owning the process performing the request. This function can be omitted by the **LTC_NO_FILE** define, which forces it to return **CRYPT_NOP** when it is called. The message digest is stored in *out*, and the *outlen* parameter is updated to hold the message digest size.

```
int hash_filehandle(          int  hash,
                             FILE *in,
                    unsigned char *out,
                    unsigned long *outlen);
```

This will hash the file identified by the handle *in* using the hash indexed by *hash*. This will begin hashing from the current file pointer position, and will not rewind the file pointer when finished. This function can be omitted by the **LTC_NO_FILE** define, which forces it to return **CRYPT_NOP** when it is called. The message digest is stored in *out*, and the *outlen* parameter is updated to hold the message digest size.

To perform the above hash with md5 the following code could be used:

```
#include <tomcrypt.h>
int main(void)
{
   int idx, err;
   unsigned long len;
```

```
   unsigned char out[MAXBLOCKSIZE];

   /* register the hash */
   if (register_hash(&md5_desc) == -1) {
      printf("Error registering MD5.\n");
      return -1;
   }

   /* get the index of the hash  */
   idx = find_hash("md5");

   /* call the hash */
   len = sizeof(out);
   if ((err =
       hash_memory(idx, "hello world", 11, out, &len)) != CRYPT_OK) {
      printf("Error hashing data: %s\n", error_to_string(err));
      return -1;
   }
   return 0;
}
```

4.2.1 Hash Registration

Similar to the cipher descriptor table you must register your hash algorithms before you can use them. These functions work exactly like those of the cipher registration code. The functions are:

```
int register_hash(const struct _hash_descriptor *hash);

int unregister_hash(const struct _hash_descriptor *hash);
```

The following hashes are provided as of this release within the LibTomCrypt library:

Name	Descriptor Name	Size of Message Digest (bytes)
WHIRLPOOL	whirlpool_desc	64
SHA-512	sha512_desc	64
SHA-384	sha384_desc	48
RIPEMD-320	rmd160_desc	40
SHA-256	sha256_desc	32
RIPEMD-256	rmd160_desc	32
SHA-224	sha224_desc	28
TIGER-192	tiger_desc	24
SHA-1	sha1_desc	20
RIPEMD-160	rmd160_desc	20
RIPEMD-128	rmd128_desc	16
MD5	md5_desc	16
MD4	md4_desc	16
MD2	md2_desc	16

Figure 4.1: Built–In Software Hashes

4.3 Cipher Hash Construction

An addition to the suite of hash functions is the *Cipher Hash Construction* or *CHC* mode. In this mode applicable block ciphers (such as AES) can be turned into hash functions that other LTC functions can use. In particular this allows a cryptosystem to be designed using very few moving parts.

In order to use the CHC system the developer will have to take a few extra steps. First the *chc_desc* hash descriptor must be registered with register_hash(). At this point the CHC hash cannot be used to hash data. While it is in the hash system you still have to tell the CHC code which cipher to use. This is accomplished via the chc_register() function.

```
int chc_register(int cipher);
```

A cipher has to be registered with CHC (and also in the cipher descriptor tables with register_cipher()). The chc_register() function will bind a cipher to the CHC system. Only one cipher can be bound to the CHC hash at a time. There are additional requirements for the system to work.

1. The cipher must have a block size greater than 64–bits.
2. The cipher must allow an input key the size of the block size.

Example of using CHC with the AES block cipher.

```
#include <tomcrypt.h>
int main(void)
{
   int err;
```

```
   /* register cipher and hash */
   if (register_cipher(&aes_enc_desc) == -1) {
      printf("Could not register cipher\n");
      return EXIT_FAILURE;
   }
   if (register_hash(&chc_desc) == -1) {
      printf("Could not register hash\n");
      return EXIT_FAILURE;
   }

   /* start chc with AES */
   if ((err = chc_register(find_cipher("aes"))) != CRYPT_OK) {
      printf("Error binding AES to CHC: %s\n",
             error_to_string(err));
   }

   /* now you can use chc_hash in any LTC function
    * [aside from pkcs...] */
}
```

4.4 Notice

It is highly recommended that you **not** use the MD4 or MD5 hashes for the purposes of digital signatures or authentication codes. These hashes are provided for completeness and they still can be used for the purposes of password hashing or one-way accumulators (e.g. Yarrow).

The other hashes such as the SHA-1, SHA-2 (that includes SHA-512, SHA-384 and SHA-256) and TIGER-192 are still considered secure for all purposes you would normally use a hash for.

Chapter 5

Message Authentication Codes

5.1 HMAC Protocol

Thanks to Dobes Vandermeer, the library now includes support for hash based message authentication codes, or HMAC for short. An HMAC of a message is a keyed authentication code that only the owner of a private symmetric key will be able to verify. The purpose is to allow an owner of a private symmetric key to produce an HMAC on a message then later verify if it is correct. Any impostor or eavesdropper will not be able to verify the authenticity of a message.

The HMAC support works much like the normal hash functions except that the initialization routine requires you to pass a key and its length. The key is much like a key you would pass to a cipher. That is, it is simply an array of octets stored in unsigned characters. The initialization routine is:

```
int hmac_init(         hmac_state *hmac,
                             int  hash,
              const unsigned char *key,
                    unsigned long  keylen);
```

The *hmac* parameter is the state for the HMAC code. The *hash* parameter is the index into the descriptor table of the hash you want to use to authenticate the message. The *key* parameter is the pointer to the array of chars that make up the key. The *keylen* parameter is the length (in octets) of the key you want to use to authenticate the message. To send octets of a message through the HMAC system you must use the following function:

```
int hmac_process(         hmac_state *hmac,
                 const unsigned char *in,
                       unsigned long  inlen);
```

hmac is the HMAC state you are working with. *in* is the array of octets to send into the HMAC process. *inlen* is the number of octets to process. Like the hash process routines, you can send the data in arbitrarily sized chunks. When you are finished with the HMAC process you must call the following function to get the HMAC code:

```
int hmac_done(   hmac_state *hmac,
```

```
                unsigned char *out,
                unsigned long *outlen);
```

The *hmac* parameter is the HMAC state you are working with. The *out* parameter is the array of octets where the HMAC code should be stored. You must set *outlen* to the size of the destination buffer before calling this function. It is updated with the length of the HMAC code produced (depending on which hash was picked). If *outlen* is less than the size of the message digest (and ultimately the HMAC code) then the HMAC code is truncated as per FIPS-198 specifications (e.g. take the first *outlen* bytes).

There are two utility functions provided to make using HMACs easier to do. They accept the key and information about the message (file pointer, address in memory), and produce the HMAC result in one shot. These are useful if you want to avoid calling the three step process yourself.

```
int hmac_memory(
                     int  hash,
   const unsigned char *key, unsigned long  keylen,
   const unsigned char *in,  unsigned long  inlen,
         unsigned char *out, unsigned long *outlen);
```

This will produce an HMAC code for the array of octets in *in* of length *inlen*. The index into the hash descriptor table must be provided in *hash*. It uses the key from *key* with a key length of *keylen*. The result is stored in the array of octets *out* and the length in *outlen*. The value of *outlen* must be set to the size of the destination buffer before calling this function. Similarly for files there is the following function:

```
int hmac_file(
                     int  hash,
              const char *fname,
   const unsigned char *key, unsigned long  keylen,
         unsigned char *out, unsigned long *outlen);
```

hash is the index into the hash descriptor table of the hash you want to use. *fname* is the filename to process. *key* is the array of octets to use as the key of length *keylen*. *out* is the array of octets where the result should be stored.

To test if the HMAC code is working there is the following function:

```
int hmac_test(void);
```

Which returns **CRYPT_OK** if the code passes otherwise it returns an error code. Some example code for using the HMAC system is given below.

```
#include <tomcrypt.h>
int main(void)
{
   int idx, err;
   hmac_state hmac;
   unsigned char key[16], dst[MAXBLOCKSIZE];
   unsigned long dstlen;
```

```
   /* register SHA-1 */
   if (register_hash(&sha1_desc) == -1) {
      printf("Error registering SHA1\n");
      return -1;
   }

   /* get index of SHA1 in hash descriptor table */
   idx = find_hash("sha1");

   /* we would make up our symmetric key in "key[]" here */

   /* start the HMAC */
   if ((err = hmac_init(&hmac, idx, key, 16)) != CRYPT_OK) {
      printf("Error setting up hmac: %s\n", error_to_string(err));
      return -1;
   }

   /* process a few octets */
   if((err = hmac_process(&hmac, "hello", 5) != CRYPT_OK) {
      printf("Error processing hmac: %s\n", error_to_string(err));
      return -1;
   }

   /* get result (presumably to use it somehow...) */
   dstlen = sizeof(dst);
   if ((err = hmac_done(&hmac, dst, &dstlen)) != CRYPT_OK) {
      printf("Error finishing hmac: %s\n", error_to_string(err));
      return -1;
   }
   printf("The hmac is %lu bytes long\n", dstlen);

   /* return */
   return 0;
}
```

5.2 OMAC Support

OMAC[1], which stands for *One-Key CBC MAC* is an algorithm which produces a Message Authentication Code (MAC) using only a block cipher such as AES. Note: OMAC has been standardized as CMAC within NIST, for the purposes of this library OMAC and CMAC are synonymous. From an API standpoint, the OMAC routines work much like the HMAC routines. Instead, in this case a cipher is used instead of a hash.

To start an OMAC state you call

```
int omac_init(         omac_state *omac,
                               int  cipher,
              const unsigned char *key,
```

[1] http://crypt.cis.ibaraki.ac.jp/omac/omac.html

```
                      unsigned long  keylen);
```

The *omac* parameter is the state for the OMAC algorithm. The *cipher* parameter is the index into the cipher_descriptor table of the cipher[2] you wish to use. The *key* and *keylen* parameters are the keys used to authenticate the data.

To send data through the algorithm call

```
int omac_process(         omac_state *state,
                 const unsigned char *in,
                       unsigned long  inlen);
```

This will send *inlen* bytes from *in* through the active OMAC state *state*. Returns **CRYPT_OK** if the function succeeds. The function is not sensitive to the granularity of the data. For example,

```
omac_process(&mystate, "hello",  5);
omac_process(&mystate, " world", 6);
```

Would produce the same result as,

```
omac_process(&mystate, "hello world",  11);
```

When you are done processing the message you can call the following to compute the message tag.

```
int omac_done(   omac_state *state,
              unsigned char *out,
              unsigned long *outlen);
```

Which will terminate the OMAC and output the *tag* (MAC) to *out*. Note that unlike the HMAC and other code *outlen* can be smaller than the default MAC size (for instance AES would make a 16-byte tag). Part of the OMAC specification states that the output may be truncated. So if you pass in $outlen = 5$ and use AES as your cipher than the output MAC code will only be five bytes long. If *outlen* is larger than the default size it is set to the default size to show how many bytes were actually used.

Similar to the HMAC code the file and memory functions are also provided. To OMAC a buffer of memory in one shot use the following function.

```
int omac_memory(
                     int  cipher,
    const unsigned char *key, unsigned long keylen,
    const unsigned char *in,  unsigned long inlen,
          unsigned char *out, unsigned long *outlen);
```

This will compute the OMAC of *inlen* bytes of *in* using the key *key* of length *keylen* bytes and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as omac_done.

To OMAC a file use

[2]The cipher must have a 64 or 128 bit block size. Such as CAST5, Blowfish, DES, AES, Twofish, etc.

```
int omac_file(
                     int  cipher,
    const unsigned char *key,      unsigned long keylen,
             const char *filename,
           unsigned char *out,      unsigned long *outlen);
```

Which will OMAC the entire contents of the file specified by *filename* using the key *key* of length *keylen* bytes and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as omac_done.

To test if the OMAC code is working there is the following function:

```
int omac_test(void);
```

Which returns **CRYPT_OK** if the code passes otherwise it returns an error code. Some example code for using the OMAC system is given below.

```
#include <tomcrypt.h>
int main(void)
{
   int idx, err;
   omac_state omac;
   unsigned char key[16], dst[MAXBLOCKSIZE];
   unsigned long dstlen;

   /* register Rijndael */
   if (register_cipher(&rijndael_desc) == -1) {
      printf("Error registering Rijndael\n");
      return -1;
   }

   /* get index of Rijndael in cipher descriptor table */
   idx = find_cipher("rijndael");

   /* we would make up our symmetric key in "key[]" here */

   /* start the OMAC */
   if ((err = omac_init(&omac, idx, key, 16)) != CRYPT_OK) {
      printf("Error setting up omac: %s\n", error_to_string(err));
      return -1;
   }

   /* process a few octets */
   if((err = omac_process(&omac, "hello", 5) != CRYPT_OK) {
      printf("Error processing omac: %s\n", error_to_string(err));
      return -1;
   }

   /* get result (presumably to use it somehow...) */
   dstlen = sizeof(dst);
   if ((err = omac_done(&omac, dst, &dstlen)) != CRYPT_OK) {
      printf("Error finishing omac: %s\n", error_to_string(err));
```

```
      return -1;
   }
   printf("The omac is %lu bytes long\n", dstlen);

   /* return */
   return 0;
}
```

5.3 PMAC Support

The PMAC[3] protocol is another MAC algorithm that relies solely on a symmetric-key block cipher. It uses essentially the same API as the provided OMAC code.

A PMAC state is initialized with the following.

```
int pmac_init(         pmac_state *pmac,
                              int  cipher,
              const unsigned char *key,
                    unsigned long  keylen);
```

Which initializes the *pmac* state with the given *cipher* and *key* of length *keylen* bytes. The chosen cipher must have a 64 or 128 bit block size (e.x. AES).

To MAC data simply send it through the process function.

```
int pmac_process(         pmac_state *state,
                 const unsigned char *in,
                       unsigned long  inlen);
```

This will process *inlen* bytes of *in* in the given *state*. The function is not sensitive to the granularity of the data. For example,

```
pmac_process(&mystate, "hello",  5);
pmac_process(&mystate, " world", 6);
```

Would produce the same result as,

```
pmac_process(&mystate, "hello world",  11);
```

When a complete message has been processed the following function can be called to compute the message tag.

```
int pmac_done(   pmac_state *state,
              unsigned char *out,
              unsigned long *outlen);
```

This will store up to *outlen* bytes of the tag for the given *state* into *out*. Note that if *outlen* is larger than the size of the tag it is set to the amount of bytes stored in *out*.

Similar to the OMAC code the file and memory functions are also provided. To PMAC a buffer of memory in one shot use the following function.

[3] J.Black, P.Rogaway, *A Block–Cipher Mode of Operation for Parallelizable Message Authentication*

```
int pmac_memory(
                    int  cipher,
    const unsigned char *key, unsigned long  keylen,
    const unsigned char *in,  unsigned long  inlen,
          unsigned char *out, unsigned long *outlen);
```

This will compute the PMAC of *msglen* bytes of *msg* using the key *key* of length *keylen* bytes, and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as pmac_done().

To PMAC a file use

```
int pmac_file(
                    int  cipher,
    const unsigned char *key,      unsigned long keylen,
             const char *filename,
          unsigned char *out,      unsigned long *outlen);
```

Which will PMAC the entire contents of the file specified by *filename* using the key *key* of length *keylen* bytes, and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as pmac_done().

To test if the PMAC code is working there is the following function:

```
int pmac_test(void);
```

Which returns **CRYPT_OK** if the code passes otherwise it returns an error code.

5.4 Pelican MAC

Pelican MAC is a new (experimental) MAC by the AES team that uses four rounds of AES as a *mixing function*. It achieves a very high rate of processing and is potentially very secure. It requires AES to be enabled to function. You do not have to register_cipher() AES first though as it calls AES directly.

```
int pelican_init(      pelican_state *pelmac,
                 const unsigned char *key,
                       unsigned long  keylen);
```

This will initialize the Pelican state with the given AES key. Once this has been done you can begin processing data.

```
int pelican_process(      pelican_state *pelmac,
                    const unsigned char *in,
                          unsigned long  inlen);
```

This will process *inlen* bytes of *in* through the Pelican MAC. It's best that you pass in multiples of 16 bytes as it makes the routine more efficient but you may pass in any length of text. You can call this function as many times as required to process an entire message.

```
int pelican_done(pelican_state *pelmac, unsigned char *out);
```

This terminates a Pelican MAC and writes the 16–octet tag to *out*.

5.4.1 Example

```
#include <tomcrypt.h>
int main(void)
{
   pelican_state pelstate;
   unsigned char key[32], tag[16];
   int           err;

   /* somehow initialize a key */

   /* initialize pelican mac */
   if ((err = pelican_init(&pelstate, /* the state */
                           key,       /* user key */
                           32         /* key length in octets */
                          )) != CRYPT_OK) {
      printf("Error initializing Pelican: %s",
              error_to_string(err));
      return EXIT_FAILURE;
   }

   /* MAC some data */
   if ((err = pelican_process(&pelstate,       /* the state */
                              "hello world",   /* data to mac */
                              11               /* length of data */
                              )) != CRYPT_OK) {
      printf("Error processing Pelican: %s",
             error_to_string(err));
      return EXIT_FAILURE;
   }

   /* Terminate the MAC */
   if ((err = pelican_done(&pelstate,/* the state */
                           tag       /* where to store the tag */
                           )) != CRYPT_OK) {
      printf("Error terminating Pelican: %s",
             error_to_string(err));
      return EXIT_FAILURE;
   }

   /* tag[0..15] has the MAC output now */

   return EXIT_SUCCESS;
}
```

5.5 XCBC-MAC

As of LibTomCrypt v1.15, XCBC-MAC (RFC 3566) has been provided to support TLS encryption suites. Like OMAC, it computes a message authentication code by using a cipher in CBC mode. It also uses a single key which it expands into the requisite three keys for the MAC function. A XCBC–MAC state is initialized with the following function:

```
int xcbc_init(         xcbc_state *xcbc,
                              int  cipher,
              const unsigned char *key,
                    unsigned long  keylen);
```

This will initialize the XCBC–MAC state *xcbc*, with the key specified in *key* of length *keylen* octets. The cipher indicated by the *cipher* index can be either a 64 or 128–bit block cipher. This will return **CRYPT_OK** on success.

It is possible to use XCBC in a three key mode by OR'ing the value **LTC_XCBC_PURE** against the *keylen* parameter. In this mode, the key is interpretted as three keys. If the cipher has a block size of n octets, the first key is then $keylen - 2n$ octets and is the encryption key. The next $2n$ octets are the K_1 and K_2 padding keys (used on the last block). For example, to use AES–192 *keylen* should be $24 + 2 \cdot 16 = 56$ octets. The three keys are interpretted as if they were concatenated in the *key* buffer.

To process data through XCBC–MAC use the following function:

```
int xcbc_process(         xcbc_state *state,
                 const unsigned char *in,
                       unsigned long  inlen);
```

This will add the message octets pointed to by *in* of length *inlen* to the XCBC–MAC state pointed to by *state*. Like the other MAC functions, the granularity of the input is not important but the order is. This will return **CRYPT_OK** on success.

To compute the MAC tag value use the following function:

```
int xcbc_done(   xcbc_state *state,
              unsigned char *out,
              unsigned long *outlen);
```

This will retrieve the XCBC–MAC tag from the state pointed to by *state*, and store it in the array pointed to by *out*. The *outlen* parameter specifies the maximum size of the destination buffer, and is updated to hold the final size of the tag when the function returns. This will return **CRYPT_OK** on success.

Helper functions are provided to make parsing memory buffers and files easier. The following functions are provided:

```
int xcbc_memory(
                     int  cipher,
    const unsigned char *key, unsigned long  keylen,
    const unsigned char *in,  unsigned long  inlen,
          unsigned char *out, unsigned long *outlen);
```

This will compute the XCBC–MAC of *msglen* bytes of *msg*, using the key *key* of length *keylen* bytes, and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as xcbc_done().

To xcbc a file use

```
int xcbc_file(
                    int  cipher,
    const unsigned char *key,      unsigned long keylen,
             const char *filename,
           unsigned char *out,      unsigned long *outlen);
```

Which will XCBC–MAC the entire contents of the file specified by *filename* using the key *key* of length *keylen* bytes, and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as xcbc_done().

To test XCBC–MAC for RFC 3566 compliance use the following function:

```
int xcbc_test(void);
```

This will return **CRYPT_OK** on success. This requires the AES or Rijndael descriptor be previously registered, otherwise, it will return **CRYPT_NOP**.

5.6 F9–MAC

The F9–MAC is yet another CBC–MAC variant proposed for the 3GPP standard. Originally specified to be used with the KASUMI block cipher, it can also be used with other ciphers. For LibTomCrypt, the F9–MAC code can use any cipher.

5.6.1 Usage Notice

F9–MAC differs slightly from the other MAC functions in that it requires the caller to perform the final message padding. The padding quite simply is a direction bit followed by a 1 bit and enough zeros to make the message a multiple of the cipher block size. If the message is byte aligned, the padding takes on the form of a single 0x40 or 0xC0 byte followed by enough 0x00 bytes to make the message proper multiple.

If the user simply wants a MAC function (hint: use OMAC) padding with a single 0x40 byte should be sufficient for security purposes and still be reasonably compatible with F9–MAC.

5.6.2 F9–MAC Functions

A F9–MAC state is initialized with the following function:

```
int f9_init(           f9_state *f9,
                            int  cipher,
            const unsigned char *key,
                  unsigned long  keylen);
```

This will initialize the F9–MAC state *f9*, with the key specified in *key* of length *keylen* octets. The cipher indicated by the *cipher* index can be either a 64 or 128–bit block cipher. This will return **CRYPT_OK** on success.

To process data through F9–MAC use the following function:

```
int f9_process(           f9_state *state,
               const unsigned char *in,
                     unsigned long  inlen);
```

This will add the message octets pointed to by *in* of length *inlen* to the F9–MAC state pointed to by *state*. Like the other MAC functions, the granularity of the input is not important but the order is. This will return **CRYPT_OK** on success.

To compute the MAC tag value use the following function:

```
int f9_done(     f9_state *state,
            unsigned char *out,
            unsigned long *outlen);
```

This will retrieve the F9–MAC tag from the state pointed to by *state*, and store it in the array pointed to by *out*. The *outlen* parameter specifies the maximum size of the destination buffer, and is updated to hold the final size of the tag when the function returns. This will return **CRYPT_OK** on success.

Helper functions are provided to make parsing memory buffers and files easier. The following functions are provided:

```
int f9_memory(
                     int  cipher,
    const unsigned char *key, unsigned long  keylen,
    const unsigned char *in,  unsigned long  inlen,
          unsigned char *out, unsigned long *outlen);
```

This will compute the F9–MAC of *msglen* bytes of *msg*, using the key *key* of length *keylen* bytes, and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as f9_done().

To F9–MAC a file use

```
int f9_file(
                     int  cipher,
    const unsigned char *key,      unsigned long keylen,
             const char *filename,
          unsigned char *out,      unsigned long *outlen);
```

Which will F9–MAC the entire contents of the file specified by *filename* using the key *key* of length *keylen* bytes, and the cipher specified by the *cipher*'th entry in the cipher_descriptor table. It will store the MAC in *out* with the same rules as f9_done().

To test f9–MAC for RFC 3566 compliance use the following function:

```
int f9_test(void);
```

This will return **CRYPT_OK** on success. This requires the AES or Rijndael descriptor be previously registered, otherwise, it will return **CRYPT_NOP**.

Chapter 6

Pseudo-Random Number Generators

6.1 Core Functions

The library provides an array of core functions for Pseudo-Random Number Generators (PRNGs) as well. A cryptographic PRNG is used to expand a shorter bit string into a longer bit string. PRNGs are used wherever random data is required such as Public Key (PK) key generation. There is a universal structure called *prng_state*. To initialize a PRNG call:

```
int XXX_start(prng_state *prng);
```

This will setup the PRNG for future use and not seed it. In order for the PRNG to be cryptographically useful you must give it entropy. Ideally you'd have some OS level source to tap like in UNIX. To add entropy to the PRNG call:

```
int XXX_add_entropy(const unsigned char *in,
                          unsigned long  inlen,
                             prng_state *prng);
```

Which returns **CRYPT_OK** if the entropy was accepted. Once you think you have enough entropy you call another function to put the entropy into action.

```
int XXX_ready(prng_state *prng);
```

Which returns **CRYPT_OK** if it is ready. Finally to actually read bytes call:

```
unsigned long XXX_read(unsigned char *out,
                       unsigned long  outlen,
                          prng_state *prng);
```

Which returns the number of bytes read from the PRNG. When you are finished with a PRNG state you call the following.

```
void XXX_done(prng_state *prng);
```

This will terminate a PRNG state and free any memory (if any) allocated. To export a PRNG state so that you can later resume the PRNG call the following.

```
int XXX_export(unsigned char *out,
               unsigned long *outlen,
                  prng_state *prng);
```

This will write a *PRNG state* to the buffer *out* of length *outlen* bytes. The idea of the export is meant to be used as a *seed file*. That is, when the program starts up there will not likely be that much entropy available. To import a state to seed a PRNG call the following function.

```
int XXX_import(const unsigned char *in,
                     unsigned long  inlen,
                        prng_state *prng);
```

This will call the start and add_entropy functions of the given PRNG. It will use the state in *in* of length *inlen* as the initial seed. You must pass the same seed length as was exported by the corresponding export function.

Note that importing a state will not *resume* the PRNG from where it left off. That is, if you export a state, emit (say) 8 bytes and then import the previously exported state the next 8 bytes will not specifically equal the 8 bytes you generated previously.

When a program is first executed the normal course of operation is:

1. Gather entropy from your sources for a given period of time or number of events.
2. Start, use your entropy via add_entropy and ready the PRNG yourself.

When your program is finished you simply call the export function and save the state to a medium (disk, flash memory, etc). The next time your application starts up you can detect the state, feed it to the import function and go on your way. It is ideal that (as soon as possible) after start up you export a fresh state. This helps in the case that the program aborts or the machine is powered down without being given a chance to exit properly.

Note that even if you have a state to import it is important to add new entropy to the state. However, there is less pressure to do so.

To test a PRNG for operational conformity call the following functions.

```
int XXX_test(void);
```

This will return **CRYPT_OK** if PRNG is operating properly.

6.1.1 Remarks

It is possible to be adding entropy and reading from a PRNG at the same time. For example, if you first seed the PRNG and call ready() you can now read from it. You can also keep adding new entropy to it. The new entropy will not be used in the PRNG until ready() is called again. This allows the PRNG to be used and re-seeded at the same time. No real error checking is guaranteed to see if the entropy is sufficient, or if the PRNG is even in a ready state before reading.

6.1.2 Example

Below is a simple snippet to read 10 bytes from Yarrow. It is important to note that this snippet is **NOT** secure since the entropy added is not random.

```
#include <tomcrypt.h>
int main(void)
{
   prng_state prng;
   unsigned char buf[10];
   int err;

   /* start it */
   if ((err = yarrow_start(&prng)) != CRYPT_OK) {
      printf("Start error: %s\n", error_to_string(err));
   }
   /* add entropy */
   if ((err = yarrow_add_entropy("hello world", 11, &prng))
       != CRYPT_OK) {
      printf("Add_entropy error: %s\n", error_to_string(err));
   }
   /* ready and read */
   if ((err = yarrow_ready(&prng)) != CRYPT_OK) {
      printf("Ready error: %s\n", error_to_string(err));
   }
   printf("Read %lu bytes from yarrow\n",
          yarrow_read(buf, sizeof(buf), &prng));
   return 0;
}
```

6.2 PRNG Descriptors

PRNGs have descriptors that allow plugin driven functions to be created using PRNGs. The plugin descriptors are stored in the structure *prng_descriptor*. The format of an element is:

```
struct _prng_descriptor {
    char *name;
    int  export_size;    /* size in bytes of exported state */

    int (*start)      (prng_state *);

    int (*add_entropy)(const unsigned char *, unsigned long,
                       prng_state *);

    int (*ready)      (prng_state *);

    unsigned long (*read)(unsigned char *, unsigned long len,
```

```
                          prng_state *);

    void (*done)(prng_state *);

    int (*export)(unsigned char *, unsigned long *, prng_state *);

    int (*import)(const unsigned char *, unsigned long, prng_state *);

    int (*test)(void);
};
```

To find a PRNG in the descriptor table the following function can be used:

```
int find_prng(const char *name);
```

This will search the PRNG descriptor table for the PRNG named *name*. It will return -1 if the PRNG is not found, otherwise, it returns the index into the descriptor table.

Just like the ciphers and hashes, you must register your prng before you can use it. The two functions provided work exactly as those for the cipher registry functions. They are the following:

```
int register_prng(const struct _prng_descriptor *prng);
int unregister_prng(const struct _prng_descriptor *prng);
```

The register function will register the PRNG, and return the index into the table where it was placed (or -1 for error). It will avoid registering the same descriptor twice, and will return the index of the current placement in the table if the caller attempts to register it more than once. The unregister function will return **CRYPT_OK** if the PRNG was found and removed. Otherwise, it returns **CRYPT_ERROR**.

6.2.1 PRNGs Provided

Name	Descriptor	Usage
Yarrow	yarrow_desc	Fast short-term PRNG
Fortuna	fortuna_desc	Fast long-term PRNG (recommended)
RC4	rc4_desc	Stream Cipher
SOBER-128	sober128_desc	Stream Cipher (also very fast PRNG)

Figure 6.1: List of Provided PRNGs

Yarrow

Yarrow is fast PRNG meant to collect an unspecified amount of entropy from sources (keyboard, mouse, interrupts, etc), and produce an unbounded string of random bytes.

Note: This PRNG is still secure for most tasks but is no longer recommended. Users should use Fortuna instead.

Fortuna

Fortuna is a fast attack tolerant and more thoroughly designed PRNG suitable for long term usage. It is faster than the default implementation of Yarrow[1] while providing more security.

Fortuna is slightly less flexible than Yarrow in the sense that it only works with the AES block cipher and SHA–256 hash function. Technically, Fortuna will work with any block cipher that accepts a 256–bit key, and any hash that produces at least a 256–bit output. However, to make the implementation simpler it has been fixed to those choices.

Fortuna is more secure than Yarrow in the sense that attackers who learn parts of the entropy being added to the PRNG learn far less about the state than that of Yarrow. Without getting into to many details Fortuna has the ability to recover from state determination attacks where the attacker starts to learn information from the PRNGs output about the internal state. Yarrow on the other hand, cannot recover from that problem until new entropy is added to the pool and put to use through the ready() function.

RC4

RC4 is an old stream cipher that can also double duty as a PRNG in a pinch. You key RC4 by calling add_entropy(), and setup the key by calling ready(). You can only add up to 256 bytes via add_entropy().

When you read from RC4, the output is XOR'ed against your buffer you provide. In this manner, you can use rc4_read() as an encrypt (and decrypt) function.

You really should not use RC4. This is not because RC4 is weak, (though biases are known to exist) but simply due to the fact that faster alternatives exist.

SOBER-128

SOBER–128 is a stream cipher designed by the QUALCOMM Australia team. Like RC4, you key it by calling add_entropy(). There is no need to call ready() for this PRNG as it does not do anything.

Note: this cipher has several oddities about how it operates. The first call to add_entropy() sets the cipher's key. Every other time call to the add_entropy() function sets the cipher's IV variable. The IV mechanism allows you to encrypt several messages with the same key, and not re–use the same key material.

Unlike Yarrow and Fortuna, all of the entropy (and hence security) of this algorithm rests in the data you pass it on the **first** call to add_entropy(). All buffers sent to add_entropy() must have a length that is a multiple of four bytes.

Like RC4, the output of SOBER–128 is XOR'ed against the buffer you provide it. In this manner, you can use sober128_read() as an encrypt (and decrypt) function.

Since SOBER-128 has a fixed keying scheme, and is very fast (faster than RC4) the ideal usage of SOBER-128 is to key it from the output of Fortuna (or Yarrow), and use it to encrypt messages. It is also ideal for simulations which need a high quality (and fast) stream of bytes.

Example Usage

```
#include <tomcrypt.h>
```

[1] Yarrow has been implemented to work with most cipher and hash combos based on which you have chosen to build into the library.

```
int main(void)
{
   prng_state prng;
   unsigned char buf[32];
   int err;

   if ((err = rc4_start(&prng)) != CRYPT_OK) {
      printf("RC4 init error: %s\n", error_to_string(err));
      exit(-1);
   }

   /* use "key" as the key */
   if ((err = rc4_add_entropy("key", 3, &prng)) != CRYPT_OK) {
      printf("RC4 add entropy error: %s\n", error_to_string(err));
      exit(-1);
   }

   /* setup RC4 for use */
   if ((err = rc4_ready(&prng)) != CRYPT_OK) {
      printf("RC4 ready error: %s\n", error_to_string(err));
      exit(-1);
   }

   /* encrypt buffer */
   strcpy(buf,"hello world");
   if (rc4_read(buf, 11, &prng) != 11) {
      printf("RC4 read error\n");
      exit(-1);
   }
   return 0;
}
```

To decrypt you have to do the exact same steps.

6.3 The Secure RNG

An RNG is related to a PRNG in many ways, except that it does not expand a smaller seed to get the data. They generate their random bits by performing some computation on fresh input bits. Possibly the hardest thing to get correctly in a cryptosystem is the PRNG. Computers are deterministic that try hard not to stray from pre–determined paths. This makes gathering entropy needed to seed a PRNG a hard task.

There is one small function that may help on certain platforms:

```
unsigned long rng_get_bytes(
    unsigned char *buf,
    unsigned long  len,
    void         (*callback)(void));
```

Which will try one of three methods of getting random data. The first is to open the popular

/dev/random device which on most *NIX platforms provides cryptographic random bits[2]. The second method is to try the Microsoft Cryptographic Service Provider, and read the RNG. The third method is an ANSI C clock drift method that is also somewhat popular but gives bits of lower entropy. The *callback* parameter is a pointer to a function that returns void. It is used when the slower ANSI C RNG must be used so the calling application can still work. This is useful since the ANSI C RNG has a throughput of roughly three bytes a second. The callback pointer may be set to **NULL** to avoid using it if you do not want to. The function returns the number of bytes actually read from any RNG source. There is a function to help setup a PRNG as well:

```
int rng_make_prng(       int  bits,
                         int  wprng,
                  prng_state *prng,
                       void (*callback)(void));
```

This will try to initialize the prng with a state of at least *bits* of entropy. The *callback* parameter works much like the callback in *rng_get_bytes()*. It is highly recommended that you use this function to setup your PRNGs unless you have a platform where the RNG does not work well. Example usage of this function is given below:

```
#include <tomcrypt.h>
int main(void)
{
   ecc_key mykey;
   prng_state prng;
   int err;

   /* register yarrow */
   if (register_prng(&yarrow_desc) == -1) {
      printf("Error registering Yarrow\n");
      return -1;
   }

   /* setup the PRNG */
   if ((err = rng_make_prng(128, find_prng("yarrow"), &prng, NULL))
       != CRYPT_OK) {
      printf("Error setting up PRNG, %s\n", error_to_string(err));
      return -1;
   }

   /* make a 192-bit ECC key */
   if ((err = ecc_make_key(&prng, find_prng("yarrow"), 24, &mykey))
       != CRYPT_OK) {
      printf("Error making key: %s\n", error_to_string(err));
      return -1;
   }
   return 0;
}
```

[2]This device is available in Windows through the Cygwin compiler suite. It emulates */dev/random* via the Microsoft CSP.

6.3.1 The Secure PRNG Interface

It is possible to access the secure RNG through the PRNG interface, and in turn use it within dependent functions such as the PK API. This simplifies the cryptosystem on platforms where the secure RNG is fast. The secure PRNG never requires to be started, that is you need not call the start, add_entropy, or ready functions. For example, consider the previous example using this PRNG.

```
#include <tomcrypt.h>
int main(void)
{
   ecc_key mykey;
   int err;

   /* register SPRNG */
   if (register_prng(&sprng_desc) == -1) {
      printf("Error registering SPRNG\n");
      return -1;
   }

   /* make a 192-bit ECC key */
   if ((err = ecc_make_key(NULL, find_prng("sprng"), 24, &mykey))
       != CRYPT_OK) {
      printf("Error making key: %s\n", error_to_string(err));
      return -1;
   }
   return 0;
}
```

Chapter 7

RSA Public Key Cryptography

7.1 Introduction

RSA wrote the PKCS #1 specifications which detail RSA Public Key Cryptography. In the specifications are padding algorithms for encryption and signatures. The standard includes the *v1.5* and *v2.1* algorithms. To simplify matters a little the v2.1 encryption and signature padding algorithms are called OAEP and PSS respectively.

7.2 PKCS #1 Padding

PKCS #1 v1.5 padding is so simple that both signature and encryption padding are performed by the same function. Note: the signature padding does **not** include the ASN.1 padding required. That is performed by the rsa_sign_hash_ex() function documented later on in this chapter.

7.2.1 PKCS #1 v1.5 Encoding

The following function performs PKCS #1 v1.5 padding:

```
int pkcs_1_v1_5_encode(
    const unsigned char *msg,
          unsigned long  msglen,
                   int  block_type,
          unsigned long  modulus_bitlen,
             prng_state *prng,
                   int  prng_idx,
          unsigned char *out,
          unsigned long *outlen);
```

This will encode the message pointed to by *msg* of length *msglen* octets. The *block_type* parameter must be set to **LTC_PKCS_1_EME** to perform encryption padding. It must be set to **LTC_PKCS_1_EMSA** to perform signature padding. The *modulus_bitlen* parameter indicates the length of the modulus in bits. The padded data is stored in *out* with a length of *outlen* octets. The output will not be longer than the modulus which helps allocate the correct output buffer size.

Only encryption padding requires a PRNG. When performing signature padding the *prng_idx* parameter may be left to zero as it is not checked for validity.

7.2.2 PKCS #1 v1.5 Decoding

The following function performs PKCS #1 v1.5 de–padding:

```
int pkcs_1_v1_5_decode(
    const unsigned char *msg,
          unsigned long  msglen,
                    int  block_type,
          unsigned long  modulus_bitlen,
          unsigned char *out,
          unsigned long *outlen,
                    int *is_valid);
```

This will remove the PKCS padding data pointed to by *msg* of length *msglen*. The decoded data is stored in *out* of length *outlen*. If the padding is valid, a 1 is stored in *is_valid*, otherwise, a 0 is stored. The *block_type* parameter must be set to either **LTC_PKCS_1_EME** or **LTC_PKCS_1_EMSA** depending on whether encryption or signature padding is being removed.

7.3 PKCS #1 v2.1 Encryption

PKCS #1 RSA Encryption amounts to OAEP padding of the input message followed by the modular exponentiation. As far as this portion of the library is concerned we are only dealing with th OAEP padding of the message.

7.3.1 OAEP Encoding

The following function performs PKCS #1 v2.1 encryption padding:

```
int pkcs_1_oaep_encode(
    const unsigned char *msg,
          unsigned long  msglen,
    const unsigned char *lparam,
          unsigned long  lparamlen,
          unsigned long  modulus_bitlen,
             prng_state *prng,
                    int  prng_idx,
                    int  hash_idx,
          unsigned char *out,
          unsigned long *outlen);
```

This accepts *msg* as input of length *msglen* which will be OAEP padded. The *lparam* variable is an additional system specific tag that can be applied to the encoding. This is useful to identify which system encoded the message. If no variance is desired then *lparam* can be set to **NULL**.

OAEP encoding requires the length of the modulus in bits in order to calculate the size of the output. This is passed as the parameter *modulus_bitlen*. *hash_idx* is the index into the hash descriptor table of the hash desired. PKCS #1 allows any hash to be used but both the encoder and decoder must use the same hash in order for this to succeed. The size of hash output affects the maximum sized input message. *prng_idx* and *prng* are the random number generator arguments required to randomize the padding process. The padded message is stored in *out* along with the length in *outlen*.

If h is the length of the hash and m the length of the modulus (both in octets) then the maximum payload for *msg* is $m - 2h - 2$. For example, with a 1024–bit RSA key and SHA–1 as the hash the maximum payload is 86 bytes.

Note that when the message is padded it still has not been RSA encrypted. You must pass the output of this function to rsa_exptmod() to encrypt it.

7.3.2 OAEP Decoding

```
int pkcs_1_oaep_decode(
    const unsigned char *msg,
          unsigned long  msglen,
    const unsigned char *lparam,
          unsigned long  lparamlen,
          unsigned long  modulus_bitlen,
                    int  hash_idx,
          unsigned char *out,
          unsigned long *outlen,
                    int *res);
```

This function decodes an OAEP encoded message and outputs the original message that was passed to the OAEP encoder. *msg* is the output of pkcs_1_oaep_encode() of length *msglen*. *lparam* is the same system variable passed to the OAEP encoder. If it does not match what was used during encoding this function will not decode the packet. *modulus_bitlen* is the size of the RSA modulus in bits and must match what was used during encoding. Similarly the *hash_idx* index into the hash descriptor table must match what was used during encoding.

If the function succeeds it decodes the OAEP encoded message into *out* of length *outlen* and stores a 1 in *res*. If the packet is invalid it stores 0 in *res* and if the function fails for another reason it returns an error code.

7.4 PKCS #1 Digital Signatures

7.4.1 PSS Encoding

PSS encoding is the second half of the PKCS #1 standard which is padding to be applied to messages that are signed.

```
int pkcs_1_pss_encode(
    const unsigned char *msghash,
          unsigned long  msghashlen,
          unsigned long  saltlen,
```

```
                prng_state *prng,
                       int  prng_idx,
                       int  hash_idx,
              unsigned long  modulus_bitlen,
              unsigned char *out,
              unsigned long *outlen);
```

This function assumes the message to be PSS encoded has previously been hashed. The input hash *msghash* is of length *msghashlen*. PSS allows a variable length random salt (it can be zero length) to be introduced in the signature process. *hash_idx* is the index into the hash descriptor table of the hash to use. *prng_idx* and *prng* are the random number generator information required for the salt.

Similar to OAEP encoding *modulus_bitlen* is the size of the RSA modulus (in bits). It limits the size of the salt. If m is the length of the modulus h the length of the hash output (in octets) then there can be $m - h - 2$ bytes of salt.

This function does not actually sign the data it merely pads the hash of a message so that it can be processed by rsa_exptmod().

7.4.2 PSS Decoding

To decode a PSS encoded signature block you have to use the following.

```
int pkcs_1_pss_decode(
    const unsigned char *msghash,
          unsigned long  msghashlen,
    const unsigned char *sig,
          unsigned long  siglen,
          unsigned long  saltlen,
                    int  hash_idx,
          unsigned long  modulus_bitlen,
                    int *res);
```

This will decode the PSS encoded message in *sig* of length *siglen* and compare it to values in *msghash* of length *msghashlen*. If the block is a valid PSS block and the decoded hash equals the hash supplied *res* is set to non–zero. Otherwise, it is set to zero. The rest of the parameters are as in the PSS encode call.

It's important to use the same *saltlen* and hash for both encoding and decoding as otherwise the procedure will not work.

7.5 RSA Key Operations

7.5.1 Background

RSA is a public key algorithm that is based on the inability to find the *e-th* root modulo a composite of unknown factorization. Normally the difficulty of breaking RSA is associated with the integer factoring problem but they are not strictly equivalent.

The system begins with with two primes p and q and their product $N = pq$. The order or *Euler totient* of the multiplicative sub-group formed modulo N is given as $\varphi(N) = (p-1)(q-1)$ which can be reduced to $\text{lcm}(p-1, q-1)$. The public key consists of the composite N and some integer e such that $\gcd(e, \varphi(N)) = 1$. The private key consists of the composite N and the inverse of e modulo $\varphi(N)$ often simply denoted as $de \equiv 1 \pmod{\varphi(N)}$.

A person who wants to encrypt with your public key simply forms an integer (the plaintext) M such that $1 < M < N - 2$ and computes the ciphertext $C = M^e \pmod{N}$. Since finding the inverse exponent d given only N and e appears to be intractable only the owner of the private key can decrypt the ciphertext and compute $C^d \equiv (M^e)^d \equiv M^1 \equiv M \pmod{N}$. Similarly the owner of the private key can sign a message by *decrypting* it. Others can verify it by *encrypting* it.

Currently RSA is a difficult system to cryptanalyze provided that both primes are large and not close to each other. Ideally e should be larger than 100 to prevent direct analysis. For example, if e is three and you do not pad the plaintext to be encrypted than it is possible that $M^3 < N$ in which case finding the cube-root would be trivial. The most often suggested value for e is 65537 since it is large enough to make such attacks impossible and also well designed for fast exponentiation (requires 16 squarings and one multiplication).

It is important to pad the input to RSA since it has particular mathematical structure. For instance $M_1^d M_2^d = (M_1 M_2)^d$ which can be used to forge a signature. Suppose $M_3 = M_1 M_2$ is a message you want to have a forged signature for. Simply get the signatures for M_1 and M_2 on their own and multiply the result together. Similar tricks can be used to deduce plaintexts from ciphertexts. It is important not only to sign the hash of documents only but also to pad the inputs with data to remove such structure.

7.5.2 RSA Key Generation

For RSA routines a single *rsa_key* structure is used. To make a new RSA key call:

```
int rsa_make_key(prng_state *prng,
                         int  wprng,
                         int  size,
                        long  e,
                     rsa_key *key);
```

Where *wprng* is the index into the PRNG descriptor array. The *size* parameter is the size in bytes of the RSA modulus desired. The e parameter is the encryption exponent desired, typical values are 3, 17, 257 and 65537. Stick with 65537 since it is big enough to prevent trivial math attacks, and not super slow. The *key* parameter is where the constructed key is placed. All keys must be at least 128 bytes, and no more than 512 bytes in size (*that is from 1024 to 4096 bits*).

Note: the *rsa_make_key()* function allocates memory at run–time when you make the key. Make sure to call *rsa_free()* (see below) when you are finished with the key. If *rsa_make_key()* fails it will automatically free the memory allocated.

There are two types of RSA keys. The types are **PK_PRIVATE** and **PK_PUBLIC**. The first type is a private RSA key which includes the CRT parameters[1] in the form of a RSAPrivateKey (PKCS #1 compliant). The second type, is a public RSA key which only includes the modulus and public exponent. It takes the form of a RSAPublicKey (PKCS #1 compliant).

[1] As of v0.99 the PK_PRIVATE_OPTIMIZED type has been deprecated, and has been replaced by the PK_PRIVATE type.

7.5.3 RSA Exponentiation

To do raw work with the RSA function, that is without padding, use the following function:

```
int rsa_exptmod(const unsigned char *in,
                      unsigned long  inlen,
                      unsigned char *out,
                      unsigned long *outlen,
                                int  which,
                           rsa_key *key);
```

This will load the bignum from *in* as a big endian integer in the format PKCS #1 specifies, raises it to either e or d and stores the result in *out* and the size of the result in *outlen*. *which* is set to **PK_PUBLIC** to use e (i.e. for encryption/verifying) and set to **PK_PRIVATE** to use d as the exponent (i.e. for decrypting/signing).

Note: the output of this function is zero–padded as per PKCS #1 specification. This allows this routine to work with PKCS #1 padding functions properly.

7.6 RSA Key Encryption

Normally RSA is used to encrypt short symmetric keys which are then used in block ciphers to encrypt a message. To facilitate encrypting short keys the following functions have been provided.

```
int rsa_encrypt_key(
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
          unsigned long *outlen,
    const unsigned char *lparam,
          unsigned long  lparamlen,
             prng_state *prng,
                    int  prng_idx,
                    int  hash_idx,
                rsa_key *key);
```

This function will OAEP pad *in* of length *inlen* bytes, RSA encrypt it, and store the ciphertext in *out* of length *outlen* octets. The *lparam* and *lparamlen* are the same parameters you would pass to pkcs_1_oaep_encode().

7.6.1 Extended Encryption

As of v1.15, the library supports both v1.5 and v2.1 PKCS #1 style paddings in these higher level functions. The following is the extended encryption function:

```
int rsa_encrypt_key_ex(
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
```

```
      unsigned long *outlen,
const unsigned char *lparam,
      unsigned long  lparamlen,
         prng_state *prng,
                int  prng_idx,
                int  hash_idx,
                int  padding,
            rsa_key *key);
```

The parameters are all the same as for rsa_encrypt_key() except for the addition of the *padding* parameter. It must be set to **LTC_PKCS_1_V1_5** to perform v1.5 encryption, or set to **LTC_PKCS_1_OAEP** to perform v2.1 encryption.

When performing v1.5 encryption, the hash and lparam parameters are totally ignored and can be set to **NULL** or zero (respectively).

7.7 RSA Key Decryption

```
int rsa_decrypt_key(
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
          unsigned long *outlen,
    const unsigned char *lparam,
          unsigned long  lparamlen,
                    int  hash_idx,
                    int *stat,
                rsa_key *key);
```

This function will RSA decrypt *in* of length *inlen* then OAEP de-pad the resulting data and store it in *out* of length *outlen*. The *lparam* and *lparamlen* are the same parameters you would pass to pkcs_1_oaep_decode().

If the RSA decrypted data is not a valid OAEP packet then *stat* is set to 0. Otherwise, it is set to 1.

7.7.1 Extended Decryption

As of v1.15, the library supports both v1.5 and v2.1 PKCS #1 style paddings in these higher level functions. The following is the extended decryption function:

```
int rsa_decrypt_key_ex(
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
          unsigned long *outlen,
    const unsigned char *lparam,
          unsigned long  lparamlen,
```

```
                  int  hash_idx,
                  int  padding,
                  int *stat,
              rsa_key *key);
```

Similar to the extended encryption, the new parameter *padding* indicates which version of the PKCS #1 standard to use. It must be set to **LTC_PKCS_1_V1_5** to perform v1.5 decryption, or set to **LTC_PKCS_1_OAEP** to perform v2.1 decryption.

When performing v1.5 decryption, the hash and lparam parameters are totally ignored and can be set to **NULL** or zero (respectively).

7.8 RSA Signature Generation

Similar to RSA key encryption RSA is also used to *digitally sign* message digests (hashes). To facilitate this process the following functions have been provided.

```
int rsa_sign_hash(const unsigned char *in,
                        unsigned long  inlen,
                        unsigned char *out,
                        unsigned long *outlen,
                           prng_state *prng,
                                  int  prng_idx,
                                  int  hash_idx,
                        unsigned long  saltlen,
                              rsa_key *key);
```

This will PSS encode the message digest pointed to by *in* of length *inlen* octets. Next, the PSS encoded hash will be RSA *signed* and the output stored in the buffer pointed to by *out* of length *outlen* octets.

The *hash_idx* parameter indicates which hash will be used to create the PSS encoding. It should be the same as the hash used to hash the message being signed. The *saltlen* parameter indicates the length of the desired salt, and should typically be small. A good default value is between 8 and 16 octets. Strictly, it must be small than $modulus\_len - hLen - 2$ where *modulus_len* is the size of the RSA modulus (in octets), and *hLen* is the length of the message digest produced by the chosen hash.

7.8.1 Extended Signatures

As of v1.15, the library supports both v1.5 and v2.1 signatures. The extended signature generation function has the following prototype:

```
int rsa_sign_hash_ex(
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
          unsigned long *outlen,
                    int  padding,
```

```
          prng_state    *prng,
                    int  prng_idx,
                    int  hash_idx,
          unsigned long  saltlen,
                rsa_key *key);
```

This will PKCS encode the message digest pointed to by *in* of length *inlen* octets. Next, the PKCS encoded hash will be RSA *signed* and the output stored in the buffer pointed to by *out* of length *outlen* octets. The *padding* parameter must be set to **LTC_PKCS_1_V1_5** to produce a v1.5 signature, otherwise, it must be set to **LTC_PKCS_1_PSS** to produce a v2.1 signature.

When performing a v1.5 signature the *prng*, *prng_idx*, and *hash_idx* parameters are not checked and can be left to any values such as {**NULL**, 0, 0}.

7.9 RSA Signature Verification

```
int rsa_verify_hash(const unsigned char *sig,
                          unsigned long  siglen,
                    const unsigned char *msghash,
                          unsigned long  msghashlen,
                                    int  hash_idx,
                          unsigned long  saltlen,
                                    int *stat,
                               rsa_key *key);
```

This will RSA *verify* the signature pointed to by *sig* of length *siglen* octets. Next, the RSA decoded data is PSS decoded and the extracted hash is compared against the message digest pointed to by *msghash* of length *msghashlen* octets.

If the RSA decoded data is not a valid PSS message, or if the PSS decoded hash does not match the *msghash* value, *res* is set to 0. Otherwise, if the function succeeds, and signature is valid *res* is set to 1.

7.9.1 Extended Verification

As of v1.15, the library supports both v1.5 and v2.1 signature verification. The extended signature verification function has the following prototype:

```
int rsa_verify_hash_ex(
    const unsigned char *sig,
          unsigned long  siglen,
    const unsigned char *hash,
          unsigned long  hashlen,
                    int  padding,
                    int  hash_idx,
          unsigned long  saltlen,
                    int *stat,
                rsa_key *key);
```

This will RSA *verify* the signature pointed to by *sig* of length *siglen* octets. Next, the RSA decoded data is PKCS decoded and the extracted hash is compared against the message digest pointed to by *msghash* of length *msghashlen* octets.

If the RSA decoded data is not a valid PSS message, or if the PKCS decoded hash does not match the *msghash* value, *res* is set to 0. Otherwise, if the function succeeds, and signature is valid *res* is set to 1.

The *padding* parameter must be set to **LTC_PKCS_1_V1_5** to perform a v1.5 verification. Otherwise, it must be set to **LTC_PKCS_1_PSS** to perform a v2.1 verification. When performing a v1.5 verification the *hash_idx* parameter is ignored.

7.10 RSA Encryption Example

```
#include <tomcrypt.h>
int main(void)
{
   int           err, hash_idx, prng_idx, res;
   unsigned long l1, l2;
   unsigned char pt[16], pt2[16], out[1024];
   rsa_key       key;

   /* register prng/hash */
   if (register_prng(&sprng_desc) == -1) {
      printf("Error registering sprng");
      return EXIT_FAILURE;
   }

   /* register a math library (in this case TomsFastMath)
   ltc_mp = tfm_desc;

   if (register_hash(&sha1_desc) == -1) {
      printf("Error registering sha1");
      return EXIT_FAILURE;
   }
   hash_idx = find_hash("sha1");
   prng_idx = find_prng("sprng");

   /* make an RSA-1024 key */
   if ((err = rsa_make_key(NULL,     /* PRNG state */
                           prng_idx, /* PRNG idx */
                           1024/8,   /* 1024-bit key */
                           65537,    /* we like e=65537 */
                           &key)     /* where to store the key */
       ) != CRYPT_OK) {
       printf("rsa_make_key %s", error_to_string(err));
       return EXIT_FAILURE;
   }

   /* fill in pt[] with a key we want to send ... */
   l1 = sizeof(out);
```

```
   if ((err = rsa_encrypt_key(pt, /* data we wish to encrypt */
                              16, /* data is 16 bytes long */
                             out, /* where to store ciphertext */
                             &l1, /* length of ciphertext */
                       "TestApp", /* our lparam for this program */
                               7, /* lparam is 7 bytes long */
                            NULL, /* PRNG state */
                        prng_idx, /* prng idx */
                        hash_idx, /* hash idx */
                            &key) /* our RSA key */
       ) != CRYPT_OK) {
       printf("rsa_encrypt_key %s", error_to_string(err));
       return EXIT_FAILURE;
   }

   /* now let's decrypt the encrypted key */
   l2 = sizeof(pt2);
   if ((err = rsa_decrypt_key(out, /* encrypted data */
                              l1, /* length of ciphertext */
                             pt2, /* where to put plaintext */
                             &l2, /* plaintext length */
                        "TestApp", /* lparam for this program */
                               7, /* lparam is 7 bytes long */
                         hash_idx, /* hash idx */
                             &res, /* validity of data */
                             &key) /* our RSA key */
        ) != CRYPT_OK) {
       printf("rsa_decrypt_key %s", error_to_string(err));
       return EXIT_FAILURE;
   }
   /* if all went well pt == pt2, l2 == 16, res == 1 */
}
```

7.11 RSA Key Format

The RSA key format adopted for exporting and importing keys is the PKCS #1 format defined by the ASN.1 constructs known as RSAPublicKey and RSAPrivateKey. Additionally, the OpenSSL key format is supported by the import function only.

7.11.1 RSA Key Export

To export a RSA key use the following function.

```
int rsa_export(unsigned char *out,
               unsigned long *outlen,
                         int  type,
                     rsa_key *key);
```

This will export the RSA key in either a RSAPublicKey or RSAPrivateKey (PKCS #1 types) depending on the value of *type*. When it is set to **PK_PRIVATE** the export format will be RSAPrivateKey and otherwise it will be RSAPublicKey.

7.11.2 RSA Key Import

To import a RSA key use the following function.

```
int rsa_import(const unsigned char *in,
                     unsigned long  inlen,
                           rsa_key *key);
```

This will import the key stored in *inlen* and import it to *key*. If the function fails it will automatically free any allocated memory. This function can import both RSAPublicKey and RSAPrivateKey formats.

As of v1.06 this function can also import OpenSSL DER formatted public RSA keys. They are essentially encapsulated RSAPublicKeys. LibTomCrypt will import the key, strip off the additional data (it's the preferred hash) and fill in the rsa_key structure as if it were a native RSAPublicKey. Note that there is no function provided to export in this format.

Chapter 8

Diffie-Hellman Key Exchange

8.1 Background

Diffie-Hellman was the original public key system proposed. The system is based upon the group structure of finite fields. For Diffie-Hellman a prime p is chosen and a "base" b such that $b^x \pmod p$ generates a large sub-group of prime order (for unique values of x).

A secret key is an exponent x and a public key is the value of $y \equiv g^x \pmod p$. The term "discrete logarithm" denotes the action of finding x given only y, g and p. The key exchange part of Diffie-Hellman arises from the fact that two users A and B with keys (A_x, A_y) and (B_x, B_y) can exchange a shared key $K \equiv B_y^{A_x} \equiv A_y^{B_x} \equiv g^{A_x B_x} \pmod p$.

From this public encryption and signatures can be developed. The trivial way to encrypt (for example) using a public key y is to perform the key exchange offline. The sender invents a key k and its public copy $k' \equiv g^k \pmod p$ and uses $K \equiv k'^{A_x} \pmod p$ as a key to encrypt the message with. Typically K would be sent to a one-way hash and the message digested used as a key in a symmetric cipher.

It is important that the order of the sub-group that g generates not only be large but also prime. There are discrete logarithm algorithms that take $\sqrt{r}$ time given the order r. The discrete logarithm can be computed modulo each prime factor of r and the results combined using the Chinese Remainder Theorem. In the cases where r is "B-Smooth" (e.g. all small factors or powers of small prime factors) the solution is trivial to find.

To thwart such attacks the primes and bases in the library have been designed and fixed. Given a prime p the order of the sub-group generated is a large prime namely $\frac{p-1}{2}$. Such primes are known as "strong primes" and the smaller prime (e.g. the order of the base) are known as Sophie-Germaine primes.

8.2 Core Functions

This library also provides core Diffie-Hellman functions so you can negotiate keys over insecure mediums. The routines provided are relatively easy to use and only take two function calls to negotiate a shared key. There is a structure called "dh_key" which stores the Diffie-Hellman key in a format these routines can use. The first routine is to make a Diffie-Hellman private key pair:

```
int dh_make_key(prng_state *prng, int wprng,
                int keysize, dh_key *key);
```

The "keysize" is the size of the modulus you want in bytes. Currently support sizes are 96 to 512 bytes which correspond to key sizes of 768 to 4096 bits. The smaller the key the faster it is to use however it will be less secure. When specifying a size not explicitly supported by the library it will round *up* to the next key size. If the size is above 512 it will return an error. So if you pass "keysize == 32" it will use a 768 bit key but if you pass "keysize == 20000" it will return an error. The primes and generators used are built-into the library and were designed to meet very specific goals. The primes are strong primes which means that if p is the prime then $p - 1$ is equal to $2r$ where r is a large prime. The bases are chosen to generate a group of order r to prevent leaking a bit of the key. This means the bases generate a very large prime order group which is good to make cryptanalysis hard.

The next two routines are for exporting/importing Diffie-Hellman keys in a binary format. This is useful for transport over communication mediums.

```
int dh_export(unsigned char *out, unsigned long *outlen,
              int type, dh_key *key);

int dh_import(const unsigned char *in, unsigned long inlen, dh_key *key);
```

These two functions work just like the "rsa_export()" and "rsa_import()" functions except these work with Diffie-Hellman keys. Its important to note you do not have to free the ram for a "dh_key" if an import fails. You can free a "dh_key" using:

```
void dh_free(dh_key *key);
```

After you have exported a copy of your public key (using **PK_PUBLIC** as "type") you can now create a shared secret with the other user using:

```
int dh_shared_secret(dh_key *private_key,
                     dh_key *public_key,
                     unsigned char *out, unsigned long *outlen);
```

Where "private_key" is the key you made and "public_key" is the copy of the public key the other user sent you. The result goes into "out" and the length into "outlen". If all went correctly the data in "out" should be identical for both parties. It is important to note that the two keys have to be the same size in order for this to work. There is a function to get the size of a key:

```
int dh_get_size(dh_key *key);
```

This returns the size in bytes of the modulus chosen for that key.

8.2.1 Remarks on Usage

Its important that you hash the shared key before trying to use it as a key for a symmetric cipher or something. An example program that communicates over sockets, using MD5 and 1024-bit DH keys is[1]:

[1]This function is a small example. It is suggested that proper packaging be used. For example, if the public key sent is truncated these routines will not detect that.

```
int establish_secure_socket(int sock, int mode, unsigned char *key,
                            prng_state *prng, int wprng)
{
   unsigned char buf[4096], buf2[4096];
   unsigned long x, len;
   int res, err, inlen;
   dh_key mykey, theirkey;

   /* make up our private key */
   if ((err = dh_make_key(prng, wprng, 128, &mykey)) != CRYPT_OK)  {
      return err;
   }

   /* export our key as public */
   x = sizeof(buf);
   if ((err = dh_export(buf, &x, PK_PUBLIC, &mykey)) != CRYPT_OK) {
      res = err;
      goto done2;
   }

   if (mode == 0) {
      /* mode 0 so we send first */
      if (send(sock, buf, x, 0) != x) {
         res = CRYPT_ERROR;
         goto done2;
      }

      /* get their key */
      if ((inlen = recv(sock, buf2, sizeof(buf2), 0)) <= 0) {
         res = CRYPT_ERROR;
         goto done2;
      }
   } else {
      /* mode >0 so we send second */
      if ((inlen = recv(sock, buf2, sizeof(buf2), 0)) <= 0) {
         res = CRYPT_ERROR;
         goto done2;
      }

      if (send(sock, buf, x, 0) != x) {
         res = CRYPT_ERROR;
         goto done2;
      }
   }

   if ((err = dh_import(buf2, inlen, &theirkey)) != CRYPT_OK) {
      res = err;
      goto done2;
   }
```

```
   /* make shared secret */
   x = sizeof(buf);
   if ((err = dh_shared_secret(&mykey, &theirkey, buf, &x)) != CRYPT_OK) {
      res = err;
      goto done;
   }

   /* hash it */
   len = 16;        /* default is MD5 so "key" must be at least 16 bytes long */
   if ((err = hash_memory(find_hash("md5"), buf, x, key, &len)) != CRYPT_OK) {
      res = err;
      goto done;
   }

   /* clean up and return */
   res = CRYPT_OK;
done:
   dh_free(&theirkey);
done2:
   dh_free(&mykey);
   zeromem(buf,  sizeof(buf));
   zeromem(buf2, sizeof(buf2));
   return res;
}
```

8.2.2 Remarks on The Snippet

When the above code snippet is done (assuming all went well) their will be a shared 128-bit key in the "key" array passed to "establish_secure_socket()".

8.3 Other Diffie-Hellman Functions

In order to test the Diffie-Hellman function internal workings (e.g. the primes and bases) their is a test function made available:

```
int dh_test(void);
```

This function returns **CRYPT_OK** if the bases and primes in the library are correct. There is one last helper function:

```
void dh_sizes(int *low, int *high);
```

Which stores the smallest and largest key sizes support into the two variables.

8.4 DH Packet

Similar to the RSA related functions there are functions to encrypt or decrypt symmetric keys using the DH public key algorithms.

```
int dh_encrypt_key(const unsigned char *in,   unsigned long  inlen,
                         unsigned char *out,  unsigned long *len,
                         prng_state *prng, int wprng, int hash,
                         dh_key *key);

int dh_decrypt_key(const unsigned char *in,  unsigned long  inlen,
                         unsigned char *out, unsigned long *outlen,
                         dh_key *key);
```

Where "in" is an input symmetric key of no more than 32 bytes. Essentially these routines created a random public key and find the hash of the shared secret. The message digest is than XOR'ed against the symmetric key. All of the required data is placed in "out" by "dh_encrypt_key()". The hash must produce a message digest at least as large as the symmetric key you are trying to share.

Similar to the RSA system you can sign and verify a hash of a message.

```
int dh_sign_hash(const unsigned char *in,  unsigned long inlen,
                       unsigned char *out, unsigned long *outlen,
                       prng_state *prng, int wprng, dh_key *key);

int dh_verify_hash(const unsigned char *sig, unsigned long siglen,
                         const unsigned char *hash, unsigned long hashlen,
                         int *stat, dh_key *key);
```

The "dh_sign_hash" function signs the message hash in "in" of length "inlen" and forms a DH packet in "out". The "dh_verify_hash" function verifies the DH signature in "sig" against the hash in "hash". It sets "stat" to non-zero if the signature passes or zero if it fails.

Chapter 9

Elliptic Curve Cryptography

9.1 Background

The library provides a set of core ECC functions as well that are designed to be the Elliptic Curve analogy of all of the Diffie-Hellman routines in the previous chapter. Elliptic curves (of certain forms) have the benefit that they are harder to attack (no sub-exponential attacks exist unlike normal DH crypto) in fact the fastest attack requires the square root of the order of the base point in time. That means if you use a base point of order 2^{192} (which would represent a 192-bit key) then the work factor is 2^{96} in order to find the secret key.

The curves in this library are taken from the following website:

```
http://csrc.nist.gov/cryptval/dss.htm
```

As of v1.15 three new curves from the SECG standards are also included they are the secp112r1, secp128r1, and secp160r1 curves. These curves were added to support smaller devices which do not need as large keys for security.

They are all curves over the integers modulo a prime. The curves have the basic equation that is:

$$y^2 = x^3 - 3x + b \pmod{p} \tag{9.1}$$

The variable b is chosen such that the number of points is nearly maximal. In fact the order of the base points β provided are very close to p that is $||\varphi(\beta)||\sim||p||$. The curves range in order from $\sim 2^{112}$ points to $\sim 2^{521}$. According to the source document any key size greater than or equal to 256-bits is sufficient for long term security.

9.2 Fixed Point Optimizations

As of v1.12 of LibTomCrypt, support for Fixed Point ECC point multiplication has been added. It is a generic optimization that is supported by any conforming math plugin. It is enabled by defining **MECC_FP** during the build, such as

```
CFLAGS="-DTFM_DESC -DMECC_FP" make
```

which will build LTC using the TFM math library and enabling this new feature. The feature is not enabled by default as it is **NOT** thread safe (by default). It supports the LTC locking macros (such as by enabling LTC_PTHREAD), but by default is not locked.

The optimization works by using a Fixed Point multiplier on any base point you use twice or more in a short period of time. It has a limited size cache (of FP_ENTRIES entries) which it uses to hold recent bases passed to ltc_ecc_mulmod(). Any base detected to be used twice is sent through the pre–computation phase, and then the fixed point algorithm can be used. For example, if you use a NIST base point twice in a row, the 2^{nd} and all subsequent point multiplications with that point will use the faster algorithm.

The optimization uses a window on the multiplicand of FP_LUT bits (default: 8, min: 2, max: 12), and this controls the memory/time trade-off. The larger the value the faster the algorithm will be but the more memory it will take. The memory usage is $3 \cdot 2^{FP\_LUT}$ integers which by default with TFM amounts to about 400kB of memory. Tuning TFM (by changing FP_SIZE) can decrease the usage by a fair amount. Memory is only used by a cache entry if it is active. Both FP_ENTRIES and FP_LUT are definable on the command line if you wish to override them. For instance,

```
CFLAGS="-DTFM_DESC -DMECC_FP -DFP_ENTRIES=8 -DFP_LUT=6" make
```

would define a window of 6 bits and limit the cache to 8 entries. Generally, it is better to first tune TFM by adjusting FP_SIZE (from tfm.h). It defaults to 4096 bits (512 bytes) which is way more than what is required by ECC. At most, you need 1152 bits to accommodate ECC–521. If you're only using (say) ECC–256 you will only need 576 bits, which would reduce the memory usage by 700%.

9.3 Key Format

LibTomCrypt uses a unique format for ECC public and private keys. While ANSI X9.63 partially specifies key formats, it does it in a less than ideally simple manner. In the case of LibTomCrypt, it is meant **solely** for NIST and SECG $GF(p)$ curves. The format of the keys is as follows:

```
ECCPublicKey ::= SEQUENCE {
    flags       BIT STRING(0), -- public/private flag (always zero),
    keySize     INTEGER,       -- Curve size (in bits) divided by eight
                               -- and rounded down, e.g. 521 => 65
    pubkey.x    INTEGER,       -- The X co-ordinate of the public key point
    pubkey.y    INTEGER,       -- The Y co-ordinate of the public key point
}

ECCPrivateKey ::= SEQUENCE {
    flags       BIT STRING(1), -- public/private flag (always one),
    keySize     INTEGER,       -- Curve size (in bits) divided by eight
                               -- and rounded down, e.g. 521 => 65
    pubkey.x    INTEGER,       -- The X co-ordinate of the public key point
    pubkey.y    INTEGER,       -- The Y co-ordinate of the public key point
    secret.k    INTEGER,       -- The secret key scalar
}
```

The first flags bit denotes whether the key is public (zero) or private (one).

9.4 ECC Curve Parameters

The library uses the following structure to describe an elliptic curve. This is used internally, as well as by the new extended ECC functions which allow the user to specify their own curves.

```
/** Structure defines a NIST GF(p) curve */
typedef struct {
   /** The size of the curve in octets */
   int size;

   /** name of curve */
   char *name;

   /** The prime that defines the field (encoded in hex) */
   char *prime;

   /** The fields B param (hex) */
   char *B;

   /** The order of the curve (hex) */
   char *order;

   /** The x co-ordinate of the base point on the curve (hex) */
   char *Gx;

   /** The y co-ordinate of the base point on the curve (hex) */
   char *Gy;
} ltc_ecc_set_type;
```

The curve must be of the form $y^2 = x^3 - 3x + b$, and all of the integer parameters are encoded in hexadecimal format.

9.5 Core Functions

9.5.1 ECC Key Generation

There is a key structure called *ecc_key* used by the ECC functions. There is a function to make a key:

```
int ecc_make_key(prng_state *prng,
                         int  wprng,
                         int  keysize,
                     ecc_key *key);
```

The *keysize* is the size of the modulus in bytes desired. Currently directly supported values are 12, 16, 20, 24, 28, 32, 48, and 65 bytes which correspond to key sizes of 112, 128, 160, 192, 224, 256, 384, and 521 bits respectively. If you pass a key size that is between any key size it will round the keysize up to the next available one.

The function will free any internally allocated resources if there is an error.

9.5.2 Extended Key Generation

As of v1.16, the library supports an extended key generation routine which allows the user to specify their own curve. It is specified as follows:

```
int  ecc_make_key_ex(
                prng_state *prng,
                       int  wprng,
                   ecc_key *key,
     const ltc_ecc_set_type *dp);
```

This function generates a random ECC key over the curve specified by the parameters by *dp*. The rest of the parameters are equivalent to those from the original key generation function.

9.5.3 ECC Key Free

To free the memory allocated by a ecc_make_key(), ecc_make_key_ex(), ecc_import(), or ecc_import_ex() call use the following function:

```
void ecc_free(ecc_key *key);
```

9.5.4 ECC Key Export

To export an ECC key using the LibTomCrypt format call the following function:

```
int ecc_export(unsigned char *out,
               unsigned long *outlen,
                        int  type,
                    ecc_key *key);
```

This will export the key with the given *type* (**PK_PUBLIC** or **PK_PRIVATE**), and store it to *out*.

9.5.5 ECC Key Import

The following function imports a LibTomCrypt format ECC key:

```
int ecc_import(const unsigned char *in,
                     unsigned long  inlen,
                           ecc_key *key);
```

This will import the ECC key from *in*, and store it in the ecc_key structure pointed to by *key*. If the operation fails it will free any allocated memory automatically.

9.5.6 Extended Key Import

The following function imports a LibTomCrypt format ECC key using a specified set of curve parameters:

```
int  ecc_import_ex(const unsigned char *in,
                         unsigned long  inlen,
                               ecc_key *key,
                const ltc_ecc_set_type *dp);
```

This will import the key from the array pointed to by *in* of length *inlen* octets. The key is stored in the ECC structure pointed to by *key*. The curve is specified by the parameters pointed to by *dp*. The function will free all internally allocated memory upon error.

9.5.7 ANSI X9.63 Export

The following function exports an ECC public key in the ANSI X9.63 format:

```
int ecc_ansi_x963_export(      ecc_key *key,
                         unsigned char *out,
                         unsigned long *outlen);
```

The ECC key pointed to by *key* is exported in public fashion to the array pointed to by *out*. The ANSI X9.63 format used is from section 4.3.6 of the standard. It does not allow for the export of private keys.

9.5.8 ANSI X9.63 Import

The following function imports an ANSI X9.63 section 4.3.6 format public ECC key:

```
int ecc_ansi_x963_import(const unsigned char *in,
                               unsigned long  inlen,
                                     ecc_key *key);
```

This will import the key stored in the array pointed to by *in* of length *inlen* octets. The imported key is stored in the ECC key pointed to by *key*. The function will free any allocated memory upon error.

9.5.9 Extended ANSI X9.63 Import

The following function allows the importing of an ANSI x9.63 section 4.3.6 format public ECC key using user specified domain parameters:

```
int ecc_ansi_x963_import_ex(const unsigned char *in,
                                  unsigned long  inlen,
                                        ecc_key *key,
                               ltc_ecc_set_type *dp);
```

This will import the key stored in the array pointed to by *in* of length *inlen* octets using the domain parameters pointed to by *dp*. The imported key is stored in the ECC key pointed to by *key*. The function will free any allocated memory upon error.

9.5.10 ECC Shared Secret

To construct a Diffie-Hellman shared secret with a private and public ECC key, use the following function:

```
int ecc_shared_secret(      ecc_key *private_key,
                            ecc_key *public_key,
                      unsigned char *out,
                      unsigned long *outlen);
```

The *private_key* is typically the local private key, and *public_key* is the key the remote party has shared. Note: this function stores only the x co-ordinate of the shared elliptic point as described in ANSI X9.63 ECC–DH.

9.6 ECC Diffie-Hellman Encryption

ECC–DH Encryption is performed by producing a random key, hashing it, and XOR'ing the digest against the plaintext. It is not strictly ANSI X9.63 compliant but it is very similar. It has been extended by using an ASN.1 sequence and hash object identifiers to allow portable usage. The following function encrypts a short string (no longer than the message digest) using this technique:

9.6.1 ECC-DH Encryption

```
int ecc_encrypt_key(const unsigned char *in,
                          unsigned long  inlen,
                          unsigned char *out,
                          unsigned long *outlen,
                             prng_state *prng,
                                    int  wprng,
                                    int  hash,
                                ecc_key *key);
```

As the name implies this function encrypts a (symmetric) key, and is not intended for encrypting long messages directly. It will encrypt the plaintext in the array pointed to by *in* of length *inlen* octets. It uses the public ECC key pointed to by *key*, and hash algorithm indexed by *hash* to construct a shared secret which may be XOR'ed against the plaintext. The ciphertext is stored in the output buffer pointed to by *out* of length *outlen* octets.

The data is encrypted to the public ECC *key* such that only the holder of the private key can decrypt the payload. To have multiple recipients multiple call to this function for each public ECC key is required.

9.6.2 ECC-DH Decryption

```
int ecc_decrypt_key(const unsigned char *in,
                          unsigned long  inlen,
                          unsigned char *out,
                          unsigned long *outlen,
                                ecc_key *key);
```

This function will decrypt an encrypted payload. The *key* provided must be the private key corresponding to the public key used during encryption. If the wrong key is provided the function will not specifically return an error code. It is important to use some form of challenge response in that case (e.g. compute a MAC of a known string).

9.6.3 ECC Encryption Format

The packet format for the encrypted keys is the following ASN.1 SEQUENCE:

```
ECCEncrypt ::= SEQUENCE {
   hashID        OBJECT IDENTIFIER, -- OID of hash used
   pubkey        OCTET STRING     , -- Encapsulated ECCPublicKey
   skey          OCTET STRING       -- xor of plaintext and
                                    --"hash of shared secret"
}
```

9.7 EC DSA Signatures

There are also functions to sign and verify messages. They use the ANSI X9.62 EC-DSA algorithm to generate and verify signatures in the ANSI X9.62 format.

9.7.1 EC-DSA Signature Generation

To sign a message digest (hash) use the following function:

```
int ecc_sign_hash(const unsigned char *in,
                        unsigned long  inlen,
                        unsigned char *out,
                        unsigned long *outlen,
                           prng_state *prng,
                                  int  wprng,
                              ecc_key *key);
```

This function will EC–DSA sign the message digest stored in the array pointed to by *in* of length *inlen* octets. The signature will be stored in the array pointed to by *out* of length *outlen* octets. The function requires a properly seeded PRNG, and the ECC *key* provided must be a private key.

9.7.2 EC-DSA Signature Verification

```
int ecc_verify_hash(const unsigned char *sig,
                          unsigned long  siglen,
                    const unsigned char *hash,
                          unsigned long  hashlen,
                                    int *stat,
                                ecc_key *key);
```

This function will verify the EC-DSA signature in the array pointed to by *sig* of length *siglen* octets, against the message digest pointed to by the array *hash* of length *hashlen*. It will store a non–zero value in *stat* if the signature is valid. Note: the function will not return an error if the signature is invalid. It will return an error, if the actual signature payload is an invalid format. The ECC *key* must be the public (or private) ECC key corresponding to the key that performed the signature.

9.7.3 Signature Format

The signature code is an implementation of X9.62 EC–DSA, and the output is compliant for GF(p) curves.

9.8 ECC Keysizes

With ECC if you try to sign a hash that is bigger than your ECC key you can run into problems. The math will still work, and in effect the signature will still work. With ECC keys the strength of the signature is limited by the size of the hash, or the size of they key, whichever is smaller. For example, if you sign with SHA256 and an ECC-192 key, you in effect have 96–bits of security.

The library will not warn you if you make this mistake, so it is important to check yourself before using the signatures.

Chapter 10

Digital Signature Algorithm

10.1 Introduction

The Digital Signature Algorithm (or DSA) is a variant of the ElGamal Signature scheme which has been modified to reduce the bandwidth of the signatures. For example, to have *80-bits of security* with ElGamal, you need a group with an order of at least 1024–bits. With DSA, you need a group of order at least 160–bits. By comparison, the ElGamal signature would require at least 256 bytes of storage, whereas the DSA signature would require only at least 40 bytes.

10.2 Key Format

Since no useful public standard for DSA key storage was presented to me during the course of this development I made my own ASN.1 SEQUENCE which I document now so that others can interoperate with this library.

```
DSAPublicKey ::= SEQUENCE {
    publicFlags    BIT STRING(0), -- must be 0
    g              INTEGER      , -- base generator
                                  -- check that g^q mod p == 1
                                  -- and that 1 < g < p - 1
    p              INTEGER      , -- prime modulus
    q              INTEGER      , -- order of sub-group
                                  -- (must be prime)
    y              INTEGER      , -- public key, specifically,
                                  -- g^x mod p,
                                  -- check that y^q mod p == 1
                                  -- and that 1 < y < p - 1
}

DSAPrivateKey ::= SEQUENCE {
    publicFlags    BIT STRING(1), -- must be 1
    g              INTEGER      , -- base generator
```

```
                                -- check that g^q mod p == 1
                                -- and that 1 < g < p - 1
    p               INTEGER   , -- prime modulus
    q               INTEGER   , -- order of sub-group
                                -- (must be prime)
    y               INTEGER   , -- public key, specifically,
                                -- g^x mod p,
                                -- check that y^q mod p == 1
                                -- and that 1 < y < p - 1
    x               INTEGER     -- private key
}
```

The leading BIT STRING has a single bit in it which is zero for public keys and one for private keys. This makes the structure uniquely decodable, and easy to work with.

10.3 Key Generation

To make a DSA key you must call the following function

```
int dsa_make_key(prng_state *prng,
                         int  wprng,
                         int  group_size,
                         int  modulus_size,
                     dsa_key *key);
```

The variable *prng* is an active PRNG state and *wprng* the index to the descriptor. *group_size* and *modulus_size* control the difficulty of forging a signature. Both parameters are in bytes. The larger the *group_size* the more difficult a forgery becomes upto a limit. The value of *group_size* is limited by $15 < group\_size < 1024$ and $modulus\_size - group\_size < 512$. Suggested values for the pairs are as follows.

Bits of Security	group_size	modulus_size
80	20	128
120	30	256
140	35	384
160	40	512

Figure 10.1: DSA Key Sizes

When you are finished with a DSA key you can call the following function to free the memory used.

```
void dsa_free(dsa_key *key);
```

10.4 Key Verification

Each DSA key is composed of the following variables.

1. q a small prime of magnitude 256^{group_size}.
2. $p = qr + 1$ a large prime of magnitude $256^{modulus\_size}$ where r is a random even integer.
3. $g = h^r \pmod p$ a generator of order q modulo p. h can be any non-trivial random value. For this library they start at $h = 2$ and step until g is not 1.
4. x a random secret (the secret key) in the range $1 < x < q$
5. $y = g^x \pmod p$ the public key.

A DSA key is considered valid if it passes all of the following tests.

1. q must be prime.
2. p must be prime.
3. g cannot be one of $\{-1, 0, 1\}$ (modulo p).
4. g must be less than p.
5. $(p - 1) \equiv 0 \pmod q$.
6. $g^q \equiv 1 \pmod p$.
7. $1 < y < p - 1$
8. $y^q \equiv 1 \pmod p$.

Tests one and two ensure that the values will at least form a field which is required for the signatures to function. Tests three and four ensure that the generator g is not set to a trivial value which would make signature forgery easier. Test five ensures that q divides the order of multiplicative sub-group of $\mathbb{Z}/p\mathbb{Z}$. Test six ensures that the generator actually generates a prime order group. Tests seven and eight ensure that the public key is within range and belongs to a group of prime order. Note that test eight does not prove that g generated y only that y belongs to a multiplicative sub-group of order q.

The following function will perform these tests.

```
int dsa_verify_key(dsa_key *key, int *stat);
```

This will test *key* and store the result in *stat*. If the result is $stat = 0$ the DSA key failed one of the tests and should not be used at all. If the result is $stat = 1$ the DSA key is valid (as far as valid mathematics are concerned).

10.5 Signatures

10.5.1 Signature Generation

To generate a DSA signature call the following function:

```
int dsa_sign_hash(const unsigned char *in,
                        unsigned long  inlen,
                        unsigned char *out,
                        unsigned long *outlen,
                           prng_state *prng,
                                  int  wprng,
                              dsa_key *key);
```

Which will sign the data in *in* of length *inlen* bytes. The signature is stored in *out* and the size of the signature in *outlen*. If the signature is longer than the size you initially specify in *outlen* nothing is stored and the function returns an error code. The DSA *key* must be of the **PK_PRIVATE** persuasion.

10.5.2 Signature Verification

To verify a hash created with that function use the following function:

```
int dsa_verify_hash(const unsigned char *sig,
                          unsigned long  siglen,
                    const unsigned char *hash,
                          unsigned long  inlen,
                                    int *stat,
                                dsa_key *key);
```

Which will verify the data in *hash* of length *inlen* against the signature stored in *sig* of length *siglen*. It will set *stat* to 1 if the signature is valid, otherwise it sets *stat* to 0.

10.6 DSA Encrypt and Decrypt

As of version 1.07, the DSA keys can be used to encrypt and decrypt small payloads. It works similar to the ECC encryption where a shared key is computed, and the hash of the shared key XOR'ed against the plaintext forms the ciphertext. The format used is functional port of the ECC encryption format to the DSA algorithm.

10.6.1 DSA Encryption

This function will encrypt a small payload with a recipients public DSA key.

```
int dsa_encrypt_key(const unsigned char *in,
                          unsigned long  inlen,
                          unsigned char *out,
                          unsigned long *outlen,
                             prng_state *prng,
                                    int  wprng,
                                    int  hash,
                                dsa_key *key);
```

This will encrypt the payload in *in* of length *inlen* and store the ciphertext in the output buffer *out*. The length of the ciphertext *outlen* must be originally set to the length of the output buffer. The DSA *key* can be a public key.

10.6.2 DSA Decryption

```
int dsa_decrypt_key(const unsigned char *in,
                          unsigned long  inlen,
                          unsigned char *out,
                          unsigned long *outlen,
                                dsa_key *key);
```

This will decrypt the ciphertext *in* of length *inlen*, and store the original payload in *out* of length *outlen*. The DSA *key* must be a private key.

10.7 DSA Key Import and Export

10.7.1 DSA Key Export

To export a DSA key so that it can be transported use the following function:

```
int dsa_export(unsigned char *out,
               unsigned long *outlen,
                         int  type,
                     dsa_key *key);
```

This will export the DSA *key* to the buffer *out* and set the length in *outlen* (which must have been previously initialized to the maximum buffer size). The *type* variable may be either **PK_PRIVATE** or **PK_PUBLIC** depending on whether you want to export a private or public copy of the DSA key.

10.7.2 DSA Key Import

To import an exported DSA key use the following function :

```
int dsa_import(const unsigned char *in,
                     unsigned long  inlen,
                           dsa_key *key);
```

This will import the DSA key from the buffer *in* of length *inlen* to the *key*. If the process fails the function will automatically free all of the heap allocated in the process (you don't have to call dsa_free()).

Chapter 11

Standards Support

11.1 ASN.1 Formats

LibTomCrypt supports a variety of ASN.1 data types encoded with the Distinguished Encoding Rules (DER) suitable for various cryptographic protocols. The data types are all provided with three basic functions with *similar* prototypes. One function has been dedicated to calculate the length in octets of a given format, and two functions have been dedicated to encoding and decoding the format.

On top of the basic data types are the SEQUENCE and SET data types which are collections of other ASN.1 types. They are provided in the same manner as the other data types except they use list of objects known as the **ltc_asn1_list** structure. It is defined as the following:

```
typedef struct {
   int                    type;
   void                  *data;
   unsigned long          size;
   int                    used;
   struct ltc_asn1_list_ *prev,  *next,
                         *child, *parent;
} ltc_asn1_list;
```

The *type* field is one of the following ASN.1 field definitions. The *data* pointer is a void pointer to the data to be encoded (or the destination) and the *size* field is specific to what you are encoding (e.g. number of bits in the BIT STRING data type). The *used* field is primarily for the CHOICE decoder and reflects if the particular member of a list was the decoded data type. To help build the lists in an orderly fashion the macro *LTC_SET_ASN1(list, index, Type, Data, Size)* has been provided.

It will assign to the *index*th position in the *list* the triplet (Type, Data, Size). An example usage would be:

```
...
ltc_asn1_list   sequence[3];
unsigned long   three=3;
```

```
LTC_SET_ASN1(sequence, 0, LTC_ASN1_IA5_STRING,    "hello", 5);
LTC_SET_ASN1(sequence, 1, LTC_ASN1_SHORT_INTEGER, &three,  1);
LTC_SET_ASN1(sequence, 2, LTC_ASN1_NULL,           NULL,   0);
```

The macro is relatively safe with respect to modifying variables, for instance the following code is equivalent.

```
...
ltc_asn1_list   sequence[3];
unsigned long   three=3;
int             x=0;
LTC_SET_ASN1(sequence, x++, LTC_ASN1_IA5_STRING,    "hello", 5);
LTC_SET_ASN1(sequence, x++, LTC_ASN1_SHORT_INTEGER, &three,  1);
LTC_SET_ASN1(sequence, x++, LTC_ASN1_NULL,           NULL,   0);
```

Definition	ASN.1 Type
LTC_ASN1_EOL	End of a ASN.1 list structure.
LTC_ASN1_BOOLEAN	BOOLEAN type
LTC_ASN1_INTEGER	INTEGER (uses mp_int)
LTC_ASN1_SHORT_INTEGER	INTEGER (32–bit using unsigned long)
LTC_ASN1_BIT_STRING	BIT STRING (one bit per char)
LTC_ASN1_OCTET_STRING	OCTET STRING (one octet per char)
LTC_ASN1_NULL	NULL
LTC_ASN1_OBJECT_IDENTIFIER	OBJECT IDENTIFIER
LTC_ASN1_IA5_STRING	IA5 STRING (one octet per char)
LTC_ASN1_UTF8_STRING	UTF8 STRING (one wchar_t per char)
LTC_ASN1_PRINTABLE_STRING	PRINTABLE STRING (one octet per char)
LTC_ASN1_UTCTIME	UTCTIME (see ltc_utctime structure)
LTC_ASN1_SEQUENCE	SEQUENCE (and SEQUENCE OF)
LTC_ASN1_SET	SET
LTC_ASN1_SETOF	SET OF
LTC_ASN1_CHOICE	CHOICE

Figure 11.1: List of ASN.1 Supported Types

11.1.1 SEQUENCE Type

The SEQUENCE data type is a collection of other ASN.1 data types encapsulated with a small header which is a useful way of sending multiple data types in one packet.

SEQUENCE Encoding

To encode a sequence a **ltc_asn1_list** array must be initialized with the members of the sequence and their respective pointers. The encoding is performed with the following function.

```
int der_encode_sequence(ltc_asn1_list *list,
                        unsigned long  inlen,
                        unsigned char *out,
                        unsigned long *outlen);
```

This encodes a sequence of items pointed to by *list* where the list has *inlen* items in it. The SEQUENCE will be encoded to *out* and of length *outlen*. The function will terminate when it reads all the items out of the list (upto *inlen*) or it encounters an item in the list with a type of **LTC_ASN1_EOL**.

The *data* pointer in the list would be the same pointer you would pass to the respective ASN.1 encoder (e.g. der_encode_bit_string()) and it is simply passed on verbatim to the dependent encoder. The list can contain other SEQUENCE or SET types which enables you to have nested SEQUENCE and SET definitions. In these cases the *data* pointer is simply a pointer to another **ltc_asn1_list**.

SEQUENCE Decoding

Decoding a SEQUENCE is similar to encoding. You set up an array of **ltc_asn1_list** where in this case the *size* member is the maximum size (in certain cases). For types such as IA5 STRING, BIT STRING, OCTET STRING (etc) the *size* field is updated after successful decoding to reflect how many units of the respective type has been loaded.

```
int der_decode_sequence(const unsigned char *in,
                              unsigned long  inlen,
                              ltc_asn1_list *list,
                              unsigned long  outlen);
```

This will decode upto *outlen* items from the input buffer *in* of length *inlen* octets. The function will stop (gracefully) when it runs out of items to decode. It will fail (for among other reasons) when it runs out of input bytes to read, a data type is invalid or a heap failure occurred.

For the following types the *size* field will be updated to reflect the number of units read of the given type.

1. BIT STRING
2. OCTET STRING
3. OBJECT IDENTIFIER
4. IA5 STRING
5. PRINTABLE STRING

SEQUENCE Length

The length of a SEQUENCE can be determined with the following function.

```
int der_length_sequence(ltc_asn1_list *list,
                        unsigned long  inlen,
                        unsigned long *outlen);
```

This will get the encoding size for the given *list* of length *inlen* and store it in *outlen*.

SEQUENCE Multiple Argument Lists

For small or simple sequences an encoding or decoding can be performed with one of the following two functions.

```
int der_encode_sequence_multi(unsigned char *out,
                              unsigned long *outlen, ...);

int der_decode_sequence_multi(const unsigned char *in,
                                    unsigned long  inlen, ...);
```

These either encode or decode (respectively) a SEQUENCE data type where the items in the sequence are specified after the length parameter.

The list of items are specified as a triple of the form *(type, size, data)* where *type* is an **int**, *size* is a **unsigned long** and *data* is **void** pointer. The list of items must be terminated with an item with the type **LTC_ASN1_EOL**.

It is ideal that you cast the *size* values to unsigned long to ensure that the proper data type is passed to the function. Constants such as *1* without a cast or prototype are of type **int** by default. Appending *UL* or pre-pending *(unsigned long)* is enough to cast it to the correct type.

```
unsigned char buf[MAXBUFSIZE];
unsigned long buflen;
int           err;

   buflen = sizeof(buf);
   if ((err =
        der_encode_sequence_multi(buf, &buflen,
        LTC_ASN1_IA5_STRING, 5UL, "Hello",
        LTC_ASN1_IA5_STRING, 7UL, " World!",
        LTC_ASN1_EOL,        0UL, NULL)) != CRYPT_OK) {
      // error handling
   }
```

This example encodes a SEQUENCE with two IA5 STRING types containing “Hello” and “ World!” respectively. Note the usage of the **UL** modifier on the size parameters. This forces the compiler to pass the numbers as the required **unsigned long** type that the function expects.

11.1.2 SET and SET OF

SET and SET OF are related to the SEQUENCE type in that they can be pretty much be decoded with the same code. However, they are different, and they should be carefully noted. The SET type is an unordered array of ASN.1 types sorted by the TAG (type identifier), whereas the SET OF type is an ordered array of a **single** ASN.1 object sorted in ascending order by the DER their respective encodings.

SET Encoding

SETs use the same array structure of ltc_asn1_list that the SEQUENCE functions use. They are encoded with the following function:

```
int der_encode_set(ltc_asn1_list *list,
                   unsigned long  inlen,
                   unsigned char *out,
                   unsigned long *outlen);
```

This will encode the list of ASN.1 objects in *list* of length *inlen* objects, and store the output in *out* of length *outlen* bytes. The function will make a copy of the list provided, and sort it by the TAG. Objects with identical TAGs are additionally sorted on their original placement in the array (to make the process deterministic).

This function will **NOT** recognize *DEFAULT* objects, and it is the responsibility of the caller to remove them as required.

SET Decoding

The SET type can be decoded with the following function.

```
int der_decode_set(const unsigned char *in,
                         unsigned long  inlen,
                         ltc_asn1_list *list,
                         unsigned long  outlen);
```

This will decode the SET specified by *list* of length *outlen* objects from the input buffer *in* of length *inlen* octets.

It handles the fact that SETs are not strictly ordered and will make multiple passes (as required) through the list to decode all the objects.

SET Length

The length of a SET can be determined by calling der_length_sequence() since they have the same encoding length.

SET OF Encoding

A *SET OF* object is an array of identical objects (e.g. OCTET STRING) sorted in ascending order by the DER encoding of the object. They are used to store objects deterministically based solely on their encoding. It uses the same array structure of ltc_asn1_list that the SEQUENCE functions use. They are encoded with the following function.

```
int der_encode_setof(ltc_asn1_list *list,
                     unsigned long  inlen,
                     unsigned char *out,
                     unsigned long *outlen);
```

This will encode a *SET OF* containing the *list* of *inlen* ASN.1 objects and store the encoding in the output buffer *out* of length *outlen*.

The routine will first encode the SET OF in an unordered fashion (in a temporary buffer) then sort using the XQSORT macro and copy back to the output buffer. This means you need at least enough memory to keep an additional copy of the output on the heap.

SET OF Decoding

Since the decoding of a *SET OF* object is unambiguous it can be decoded with der_decode_sequence().

SET OF Length

Like the SET type the der_length_sequence() function can be used to determine the length of a *SET OF* object.

11.1.3 ASN.1 INTEGER

To encode or decode INTEGER data types use the following functions.

```
int der_encode_integer(         void *num,
                       unsigned char *out,
                       unsigned long *outlen);

int der_decode_integer(const unsigned char *in,
                             unsigned long  inlen,
                                      void *num);

int der_length_integer(         void *num,
                       unsigned long *len);
```

These will encode or decode a signed INTEGER data type using the bignum data type to store the large INTEGER. To encode smaller values without allocating a bignum to store the value, the *short* INTEGER functions were made available.

```
int der_encode_short_integer(unsigned long  num,
                             unsigned char *out,
                             unsigned long *outlen);

int der_decode_short_integer(const unsigned char *in,
                                   unsigned long  inlen,
                                   unsigned long *num);

int der_length_short_integer(unsigned long  num,
                             unsigned long *outlen);
```

These will encode or decode an unsigned **unsigned long** type (only reads upto 32–bits). For values in the range $0 \ldots 2^{32} - 1$ the integer and short integer functions can encode and decode each others outputs.

11.1.4 ASN.1 BIT STRING

```
int der_encode_bit_string(const unsigned char *in,
                                unsigned long  inlen,
                                unsigned char *out,
```

```
                                unsigned long *outlen);

int der_decode_bit_string(const unsigned char *in,
                                unsigned long  inlen,
                                unsigned char *out,
                                unsigned long *outlen);

int der_length_bit_string(unsigned long  nbits,
                          unsigned long *outlen);
```

These will encode or decode a BIT STRING data type. The bits are passed in (or read out) using one **char** per bit. A non–zero value will be interpreted as a one bit, and a zero value a zero bit.

11.1.5 ASN.1 OCTET STRING

```
int der_encode_octet_string(const unsigned char *in,
                                 unsigned long  inlen,
                                 unsigned char *out,
                                 unsigned long *outlen);

int der_decode_octet_string(const unsigned char *in,
                                 unsigned long  inlen,
                                 unsigned char *out,
                                 unsigned long *outlen);

int der_length_octet_string(unsigned long  noctets,
                            unsigned long *outlen);
```

These will encode or decode an OCTET STRING data type. The octets are stored using one **unsigned char** each.

11.1.6 ASN.1 OBJECT IDENTIFIER

```
int der_encode_object_identifier(unsigned long *words,
                                 unsigned long  nwords,
                                 unsigned char *out,
                                 unsigned long *outlen);

int der_decode_object_identifier(const unsigned char *in,
                                       unsigned long  inlen,
                                       unsigned long *words,
                                       unsigned long *outlen);

int der_length_object_identifier(unsigned long *words,
                                 unsigned long  nwords,
                                 unsigned long *outlen);
```

These will encode or decode an OBJECT IDENTIFIER object. The words of the OID are stored in individual **unsigned long** elements, and must be in the range $0 \ldots 2^{32} - 1$.

11.1.7 ASN.1 IA5 STRING

```
int der_encode_ia5_string(const unsigned char *in,
                                unsigned long  inlen,
                                unsigned char *out,
                                unsigned long *outlen);

int der_decode_ia5_string(const unsigned char *in,
                                unsigned long  inlen,
                                unsigned char *out,
                                unsigned long *outlen);

int der_length_ia5_string(const unsigned char *octets,
                                unsigned long  noctets,
                                unsigned long *outlen);
```

These will encode or decode an IA5 STRING. The characters are read or stored in individual **char** elements. These functions performs internal character to numerical conversions based on the conventions of the compiler being used. For instance, on an x86_32 machine 'A' == 65 but the same may not be true on say a SPARC machine. Internally, these functions have a table of literal characters and their numerical ASCII values. This provides a stable conversion provided that the build platform honours the run–time platforms character conventions.

11.1.8 ASN.1 PRINTABLE STRING

```
int der_encode_printable_string(const unsigned char *in,
                                      unsigned long  inlen,
                                      unsigned char *out,
                                      unsigned long *outlen);

int der_decode_printable_string(const unsigned char *in,
                                      unsigned long  inlen,
                                      unsigned char *out,
                                      unsigned long *outlen);

int der_length_printable_string(const unsigned char *octets,
                                      unsigned long  noctets,
                                      unsigned long *outlen);
```

These will encode or decode an PRINTABLE STRING. The characters are read or stored in individual **char** elements. These functions performs internal character to numerical conversions based on the conventions of the compiler being used. For instance, on an x86_32 machine 'A' == 65 but the same may not be true on say a SPARC machine. Internally, these functions have a table of literal characters and their numerical ASCII values. This provides a stable conversion provided that the build platform honours the run-time platforms character conventions.

11.1.9 ASN.1 UTF8 STRING

```
int der_encode_utf8_string(const wchar_t *in,
                           unsigned long  inlen,
                           unsigned char *out,
                           unsigned long *outlen);

int der_decode_utf8_string(const unsigned char *in,
                                 unsigned long  inlen,
                                       wchar_t *out,
                                 unsigned long *outlen);

int der_length_utf8_string(const wchar_t *octets,
                           unsigned long  noctets,
                           unsigned long *outlen);
```

These will encode or decode an UTF8 STRING. The characters are read or stored in individual **wchar_t** elements. These function performs no internal mapping and treat the characters as literals.

These functions use the **wchar_t** type which is not universally available. In those cases, the library will typedef it to **unsigned long**. If you intend to use the ISO C functions for working with wide–char arrays, you should make sure that wchar_t has been defined previously.

11.1.10 ASN.1 UTCTIME

The UTCTIME type is to store a date and time in ASN.1 format. It uses the following structure to organize the time.

```
typedef struct {
   unsigned YY, /* year    00--99 */
            MM, /* month   01--12 */
            DD, /* day     01--31 */
            hh, /* hour    00--23 */
            mm, /* minute  00--59 */
            ss, /* second  00--59 */
            off_dir, /* timezone offset direction 0 == +, 1 == - */
            off_hh, /* timezone offset hours */
            off_mm; /* timezone offset minutes */
} ltc_utctime;
```

The time can be offset plus or minus a set amount of hours (off_hh) and minutes (off_mm). When *off_dir* is zero, the time will be added otherwise it will be subtracted. For instance, the array $\{5, 6, 20, 22, 4, 00, 0, 5, 0\}$ represents the current time of *2005, June 20th, 22:04:00* with a time offset of +05h00.

```
int der_encode_utctime(  ltc_utctime *utctime,
                         unsigned char *out,
                         unsigned long *outlen);
```

```
int der_decode_utctime(const unsigned char *in,
                             unsigned long *inlen,
                               ltc_utctime *out);

int der_length_utctime(  ltc_utctime *utctime,
                       unsigned long *outlen);
```

The encoder will store time in one of the two ASN.1 formats, either *YYMMDDhhmmssZ* or *YYMMDDhhmmss±hhmm*, and perform minimal error checking on the input. The decoder will read all valid ASN.1 formats and perform range checking on the values (not complete but rational) useful for catching packet errors.

It is suggested that decoded data be further scrutinized (e.g. days of month in particular).

11.1.11 ASN.1 CHOICE

The CHOICE ASN.1 type represents a union of ASN.1 types all of which are stored in a *ltc_asn1_list*. There is no encoder for the CHOICE type, only a decoder. The decoder will scan through the provided list attempting to use the appropriate decoder on the input packet. The list can contain any ASN.1 data type[1] except for other CHOICE types.

There is no encoder for the CHOICE type as the actual DER encoding is the encoding of the chosen type.

```
int der_decode_choice(const unsigned char *in,
                            unsigned long *inlen,
                            ltc_asn1_list *list,
                            unsigned long  outlen);
```

This will decode the input in the *in* field of length *inlen*. It uses the provided ASN.1 list specified in the *list* field which has *outlen* elements. The *inlen* field will be updated with the length of the decoded data type, as well as the respective entry in the *list* field will have the *used* flag set to non–zero to reflect it was the data type decoded.

11.1.12 ASN.1 Flexi Decoder

The ASN.1 *flexi* decoder allows the developer to decode arbitrary ASN.1 DER packets (provided they use data types LibTomCrypt supports) without first knowing the structure of the data. Where der_decode _sequence() requires the developer to specify the data types to decode in advance the flexi decoder is entirely free form.

The flexi decoder uses the same *ltc_asn1_list* but instead of being stored in an array it uses the linked list pointers *prev*, *next*, *parent* and *child*. The list works as a *doubly-linked list* structure where decoded items at the same level are siblings (using next and prev) and items encoded in a SEQUENCE are stored as a child element.

When a SEQUENCE or SET has been encountered a SEQUENCE (or SET resp.) item will be added as a sibling (e.g. list.type == LTC_ASN1_SEQUENCE) and the child pointer points to a new list of items contained within the object.

[1] Except it cannot have LTC_ASN1_INTEGER and LTC_ASN1_SHORT_INTEGER simultaneously.

```
int  der_decode_sequence_flexi(const unsigned char *in,
                                     unsigned long *inlen,
                                    ltc_asn1_list **out);
```

This will decode items in the *in* buffer of max input length *inlen* and store the newly created pointer to the list in *out*. This function allocates all required memory for the decoding. It stores the number of octets read back into *inlen*.

The function will terminate when either it hits an invalid ASN.1 tag, or it reads *inlen* octets. An early termination is a soft error, and returns normally. The decoded list *out* will point to the very first element of the list (e.g. both parent and prev pointers will be **NULL**).

An invalid decoding will terminate the process, and free the allocated memory automatically.

Note: the list decoded by this function is **NOT** in the correct form for der_encode_sequence() to use directly. You will first have to convert the list by first storing all of the siblings in an array then storing all the children as sub-lists of a sequence using the *.data* pointer. Currently no function in LibTomCrypt provides this ability.

Sample Decoding

Suppose we decode the following structure:

```
User ::= SEQUENCE {
   Name        IA5 STRING
   LoginToken  SEQUENCE {
      passwdHash   OCTET STRING
      pubkey       ECCPublicKey
   }
   LastOn      UTCTIME
}
```

and we decoded it with the following code:

```
unsigned char inbuf[MAXSIZE];
unsigned long inbuflen;
ltc_asn1_list *list;
int           err;

/* somehow fill inbuf/inbuflen */
if ((err = der_decode_sequence_flexi(inbuf, inbuflen, &list)) != CRYPT_OK) {
   printf("Error decoding: %s\n", error_to_string(err));
   exit(EXIT_FAILURE);
}
```

At this point *list* would point to the SEQUENCE identified by *User*. It would have no sibblings (prev or next), and only a child node. Walking to the child node with the following code will bring us to the *Name* portion of the SEQUENCE:

```
list = list->child;
```

Now *list* points to the *Name* member (with the tag IA5 STRING). The *data*, *size*, and *type* members of *list* should reflect that of an IA5 STRING. The sibbling will now be the *LoginToken* SEQUENCE. The sibbling has a child node which points to the *passwdHash* OCTET STRING. We can walk to this node with the following code:

```
/* list already pointing to 'Name' */
list = list->next->child;
```

At this point, *list* will point to the *passwdHash* member of the innermost SEQUENCE. This node has a sibbling, the *pubkey* member of the SEQUENCE. The *LastOn* member of the SEQUENCE is a sibbling of the LoginToken node, if we wanted to walk there we would have to go up and over via:

```
list = list->parent->next;
```

At this point, we are pointing to the last node of the list. Lists are terminated in all directions by a **NULL** pointer. All nodes are doubly linked so that you can walk up and down the nodes without keeping pointers lying around.

Free'ing a Flexi List

To free the list use the following function.

```
void der_sequence_free(ltc_asn1_list *in);
```

This will free all of the memory allocated by der_decode_sequence_flexi().

11.2 Password Based Cryptography

11.2.1 PKCS #5

In order to securely handle user passwords for the purposes of creating session keys and chaining IVs the PKCS #5 was drafted. PKCS #5 is made up of two algorithms, Algorithm One and Algorithm Two. Algorithm One is the older fairly limited algorithm which has been implemented for completeness. Algorithm Two is a bit more modern and more flexible to work with.

11.2.2 Algorithm One

Algorithm One accepts as input a password, an 8–byte salt, and an iteration counter. The iteration counter is meant to act as delay for people trying to brute force guess the password. The higher the iteration counter the longer the delay. This algorithm also requires a hash algorithm and produces an output no longer than the output of the hash.

```
int pkcs_5_alg1(const unsigned char *password,
                      unsigned long  password_len,
                const unsigned char *salt,
                                int  iteration_count,
                                int  hash_idx,
                      unsigned char *out,
                      unsigned long *outlen)
```

Where *password* is the user's password. Since the algorithm allows binary passwords you must also specify the length in *password_len*. The *salt* is a fixed size 8–byte array which should be random

for each user and session. The *iteration_count* is the delay desired on the password. The *hash_idx* is the index of the hash you wish to use in the descriptor table.

The output of length up to *outlen* is stored in *out*. If *outlen* is initially larger than the size of the hash functions output it is set to the number of bytes stored. If it is smaller than not all of the hash output is stored in *out*.

11.2.3 Algorithm Two

Algorithm Two is the recommended algorithm for this task. It allows variable length salts, and can produce outputs larger than the hash functions output. As such, it can easily be used to derive session keys for ciphers and MACs as well initial vectors as required from a single password and invocation of this algorithm.

```
int pkcs_5_alg2(const unsigned char *password,
                      unsigned long  password_len,
                const unsigned char *salt,
                      unsigned long  salt_len,
                                int  iteration_count,
                                int  hash_idx,
                      unsigned char *out,
                      unsigned long *outlen)
```

Where *password* is the users password. Since the algorithm allows binary passwords you must also specify the length in *password_len*. The *salt* is an array of size *salt_len*. It should be random for each user and session. The *iteration_count* is the delay desired on the password. The *hash_idx* is the index of the hash you wish to use in the descriptor table. The output of length up to *outlen* is stored in *out*.

```
/* demo to show how to make session state material
 * from a password */
#include <tomcrypt.h>
int main(void)
{
    unsigned char password[100], salt[100],
                  cipher_key[16], cipher_iv[16],
                  mac_key[16], outbuf[48];
    int           err, hash_idx;
    unsigned long outlen, password_len, salt_len;

    /* register hash and get it's idx .... */

    /* get users password and make up a salt ... */

    /* create the material (100 iterations in algorithm) */
    outlen = sizeof(outbuf);
    if ((err = pkcs_5_alg2(password, password_len, salt,
                           salt_len, 100, hash_idx, outbuf,
```

```
                          &outlen))
       != CRYPT_OK) {
       /* error handle */
    }

    /* now extract it */
    memcpy(cipher_key, outbuf, 16);
    memcpy(cipher_iv,  outbuf+16, 16);
    memcpy(mac_key,    outbuf+32, 16);

    /* use material (recall to store the salt in the output) */
}
```

11.3 Key Derviation Functions

11.3.1 HKDF

A key derivation function (KDF) is a basic and essential component of cryptographic systems. Its goal is to take some source of initial keying material and derive from it one or more cryptographically strong secret keys.

HKDF follows the "extract-then-expand" paradigm, where the KDF logically consists of two modules. The first stage takes the input keying material and "extracts" from it a fixed-length pseudorandom key K. The second stage "expands" the key K into several additional pseudorandom keys (the output of the KDF).

In many applications, the input keying material is not necessarily distributed uniformly, and the attacker may have some partial knowledge about it (for example, a Diffie-Hellman value computed by a key exchange protocol) or even partial control of it (as in some entropy-gathering applications). Thus, the goal of the "extract" stage is to "concentrate" the possibly dispersed entropy of the input keying material into a short, but cryptographically strong, pseudorandom key. In some applications, the input may already be a good pseudorandom key; in these cases, the "extract" stage is not necessary, and the "expand" part can be used alone.

The second stage "expands" the pseudorandom key to the desired length; the number and lengths of the output keys depend on the specific cryptographic algorithms for which the keys are needed.

11.3.2 HKDF Extract

To perform the extraction phase, use the following function:

```
int hkdf_extract(   int  hash_idx,
    const unsigned char *salt,
          unsigned long  saltlen,
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
          unsigned long *outlen);
```

The *hash_idx* parameter is the index into the descriptor table of the hash you want to use. The *salt* parameter is a pointer to the array of octets of length *saltlen* containing the salt or a NULL pointer if a salt is not being used (in that case set saltlen to 0). *in* is a pointer to an array of octets of length *inlen* containing the source entropy. The extracted output is stored in the location pointed to by *out*. You must set *outlen* to the size of the destination buffer before calling this function. It is updated to the length of the extracted output. If *outlen* is too small the extracted output will be truncated.

While the salt is optional, using one improves HKDF's security. If used, the salt should be randomly chosen, but does not need to be secret and may be re-used. Please see RFC5869 section 3.1 for more details.

11.3.3 HKDF Expand

To perform the expansion phase, use the following function:

```
int hkdf_expand(    int  hash_idx,
    const unsigned char *info,
          unsigned long  infolen,
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
          unsigned long  outlen);
```

The *hash_idx* parameter is the index into the descriptor table of the hash you want to use. The *info* parameter, an array of octets of length *infolen*, is an optional parameter (set *info* to NULL and *infolen* to 0 if not using it) which may be used to bind the derived keys to some application and context specific information. This prevents the same keying material from being generated in different contexts. Please see RFC5869 section 3.2 for more information. The extracted keying material is passed as octet array *in* of length *inlen*. Expanded output of length *outlen* is generated and stored in octet arrat *out*.

11.3.4 HKDF Extract-and-Expand

To perform both phases together, use the following function:

```
int hkdf(           int  hash_idx,
    const unsigned char *salt,
          unsigned long  saltlen,
    const unsigned char *info,
          unsigned long  infolen,
    const unsigned char *in,
          unsigned long  inlen,
          unsigned char *out,
          unsigned long  outlen);
```

Parameters are as in *hkdf_extract()* and *hkdf_expand()*.

Chapter 12

Miscellaneous

12.1 Base64 Encoding and Decoding

The library provides functions to encode and decode a RFC 1521 base–64 coding scheme. The characters used in the mappings are:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

Those characters are supported in the 7-bit ASCII map, which means they can be used for transport over common e-mail, usenet and HTTP mediums. The format of an encoded stream is just a literal sequence of ASCII characters where a group of four represent 24-bits of input. The first four chars of the encoders output is the length of the original input. After the first four characters is the rest of the message.

Often, it is desirable to line wrap the output to fit nicely in an e-mail or usenet posting. The decoder allows you to put any character (that is not in the above sequence) in between any character of the encoders output. You may not however, break up the first four characters.

To encode a binary string in base64 call:

```
int base64_encode(const unsigned char *in,
                        unsigned long  len,
                        unsigned char *out,
                        unsigned long *outlen);
```

Where *in* is the binary string and *out* is where the ASCII output is placed. You must set the value of *outlen* prior to calling this function and it sets the length of the base64 output in *outlen* when it is done. To decode a base64 string call:

```
int base64_decode(const unsigned char *in,
                        unsigned long  len,
                        unsigned char *out,
                        unsigned long *outlen);
```

12.2 Primality Testing

The library includes primality testing and random prime functions as well. The primality tester will perform the test in two phases. First it will perform trial division by the first few primes. Second it will perform eight rounds of the Rabin-Miller primality testing algorithm. If the candidate passes both phases it is declared prime otherwise it is declared composite. No prime number will fail the two phases but composites can. Each round of the Rabin-Miller algorithm reduces the probability of a pseudo-prime by $\frac{1}{4}$ therefore after sixteen rounds the probability is no more than $\left(\frac{1}{4}\right)^8 = 2^{-16}$. In practice the probability of error is in fact much lower than that.

When making random primes the trial division step is in fact an optimized implementation of *Implementation of Fast RSA Key Generation on Smart Cards*[1]. In essence a table of machine-word sized residues are kept of a candidate modulo a set of primes. When the candidate is rejected and ultimately incremented to test the next number the residues are updated without using multi-word precision math operations. As a result the routine can scan ahead to the next number required for testing with very little work involved.

In the event that a composite did make it through it would most likely cause the the algorithm trying to use it to fail. For instance, in RSA two primes p and q are required. The order of the multiplicative sub-group (modulo pq) is given as $\varphi(pq)$ or $(p-1)(q-1)$. The decryption exponent d is found as $de \equiv 1 \pmod{\varphi(pq)}$. If either p or q is composite the value of d will be incorrect and the user will not be able to sign or decrypt messages at all. Suppose p was prime and q was composite this is just a variation of the multi-prime RSA. Suppose $q = rs$ for two primes r and s then $\varphi(pq) = (p-1)(r-1)(s-1)$ which clearly is not equal to $(p-1)(rs-1)$.

These are not technically part of the LibTomMath library but this is the best place to document them. To test if a *mp_int* is prime call:

```
int is_prime(mp_int *N, int *result);
```

This puts a one in *result* if the number is probably prime, otherwise it places a zero in it. It is assumed that if it returns an error that the value in *result* is undefined. To make a random prime call:

```
int rand_prime(       mp_int *N,
               unsigned long len,
                  prng_state *prng,
                         int  wprng);
```

Where *len* is the size of the prime in bytes ($2 \le len \le 256$). You can set *len* to the negative size you want to get a prime of the form $p \equiv 3 \pmod 4$. So if you want a 1024-bit prime of this sort pass *len = -128* to the function. Upon success it will return **CRYPT_OK** and N will contain an integer which is very likely prime.

[1] Chenghuai Lu, Andre L. M. dos Santos and Francisco R. Pimentel

Chapter 13

Programming Guidelines

13.1 Secure Pseudo Random Number Generators

Probably the single most vulnerable point of any cryptosystem is the PRNG. Without one, generating and protecting secrets would be impossible. The requirement that one be setup correctly is vitally important, and to address this point the library does provide two RNG sources that will address the largest amount of end users as possible. The *sprng* PRNG provides an easy to access source of entropy for any application on a UNIX (and the like) or Windows computer.

However, when the end user is not on one of these platforms, the application developer must address the issue of finding entropy. This manual is not designed to be a text on cryptography. I would just like to highlight that when you design a cryptosystem make sure the first problem you solve is getting a fresh source of entropy.

13.2 Preventing Trivial Errors

Two simple ways to prevent trivial errors is to prevent overflows, and to check the return values. All of the functions which output variable length strings will require you to pass the length of the destination. If the size of your output buffer is smaller than the output it will report an error. Therefore, make sure the size you pass is correct!

Also, virtually all of the functions return an error code or **CRYPT_OK**. You should detect all errors, as simple typos can cause algorithms to fail to work as desired.

13.3 Registering Your Algorithms

To avoid linking and other run–time errors it is important to register the ciphers, hashes and PRNGs you intend to use before you try to use them. This includes any function which would use an algorithm indirectly through a descriptor table.

A neat bonus to the registry system is that you can add external algorithms that are not part of the library without having to hack the library. For example, suppose you have a hardware specific PRNG on your system. You could easily write the few functions required plus a descriptor. After

registering your PRNG, all of the library functions that need a PRNG can instantly take advantage of it. The same applies for ciphers, hashes, and bignum math routines.

13.4 Key Sizes

13.4.1 Symmetric Ciphers

For symmetric ciphers, use as large as of a key as possible. For the most part *bits are cheap* so using a 256–bit key is not a hard thing to do. As a good rule of thumb do not use a key smaller than 128 bits.

13.4.2 Asymmetric Ciphers

The following chart gives the work factor for solving a DH/RSA public key using the NFS. The work factor for a key of order n is estimated to be

$$e^{1.923 \cdot ln(n)^{\frac{1}{3}} \cdot ln(ln(n))^{\frac{2}{3}}} \tag{13.1}$$

Note that n is not the bit-length but the magnitude. For example, for a 1024-bit key $n = 2^{1024}$. The work required is:

RSA/DH Key Size (bits)	Work Factor (log_2)
512	63.92
768	76.50
1024	86.76
1536	103.37
2048	116.88
2560	128.47
3072	138.73
4096	156.49

Figure 13.1: RSA/DH Key Strength

The work factor for ECC keys is much higher since the best attack is still fully exponential. Given a key of magnitude n it requires $\sqrt{n}$ work. The following table summarizes the work required:

Using the above tables the following suggestions for key sizes seems appropriate:

Security Goal	RSA/DH Key Size (bits)	ECC Key Size (bits)
Near term	1024	160
Short term	1536	192
Long Term	2560	384

ECC Key Size (bits)	Work Factor (log_2)
112	56
128	64
160	80
192	96
224	112
256	128
384	192
521	260.5

Figure 13.2: ECC Key Strength

13.5 Thread Safety

The library is not fully thread safe but several simple precautions can be taken to avoid any problems. The registry functions such as register_cipher() are not thread safe no matter what you do. It is best to call them from your programs initialization code before threads are initiated.

The rest of the code uses state variables you must pass it such as hash_state, hmac_state, etc. This means that if each thread has its own state variables then they will not affect each other, and are fully thread safe. This is fairly simple with symmetric ciphers and hashes.

The only sticky issue is a shared PRNG which can be alleviated with the careful use of mutex devices. Defining LTC_PTHREAD for instance, enables pthreads based mutex locking in various routines such as the Yarrow and Fortuna PRNGs, the fixed point ECC multiplier, and other routines.

Chapter 14

Configuring and Building the Library

14.1 Introduction

The library is fairly flexible about how it can be built, used, and generally distributed. Additions are being made with each new release that will make the library even more flexible. Each of the classes of functions can be disabled during the build process to make a smaller library. This is particularly useful for shared libraries.

As of v1.06 of the library, the build process has been moved to two steps for the typical LibTomCrypt application. This is because LibTomCrypt no longer provides a math API on its own and relies on third party libraries (such as LibTomMath, GnuMP, or TomsFastMath).

The build process now consists of installing a math library first, and then building and installing LibTomCrypt with a math library configured. Note that LibTomCrypt can be built with no internal math descriptors. This means that one must be provided at either build, or run time for the application. LibTomCrypt comes with three math descriptors that provide a standard interface to math libraries.

14.2 Makefile variables

All GNU driven makefiles (including the makefile for ICC) use a set of common variables to control the build and install process. Most of the settings can be overwritten from the command line which makes custom installation a breeze.

14.2.1 MAKE, CC and AR

The MAKE, CC and AR flags can all be overwritten. They default to *make*, *$CC* and *$AR* respectively. Changing MAKE allows you to change what program will be invoked to handle sub–directories. For example, this

```
MAKE=gmake gmake install
```

will build and install the libraries with the *gmake* tool. Similarly,

```
CC=arm-gcc AR=arm-ar make
```

will build the library using *arm–gcc* as the compiler and *arm–ar* as the archiver.

14.2.2 IGNORE_SPEED

When **IGNORE_SPEED** has been defined the default optimization flags for CFLAGS will be disabled which allows the developer to specify new CFLAGS on the command line. E.g. to add debugging

```
CFLAGS="-g3" make IGNORE_SPEED=1
```

This will turn off optimizations and add *-g3* to the CFLAGS which enables debugging.

14.2.3 LIBNAME and LIBNAME_S

LIBNAME is the name of the output library (archive) to create. It defaults to *libtomcrypt.a* for static builds and *libtomcrypt.la* for shared. The **LIBNAME_S** variable is the static name while doing shared builds. Ideally they should have the same prefix but don't have to.

Similarly **LIBTEST** and **LIBTEST_S** are the names for the profiling and testing library. The default is *libtomcrypt_prof.a* for static and *libtomcrypt_prof.la* for shared.

14.2.4 Installation Directories

DESTDIR is the prefix for the installation directories. It defaults to an empty string. **LIBPATH** is the prefix for the library directory which defaults to */usr/lib*. **INCPATH** is the prefix for the header file directory which defaults to */usr/include*. **DATADIR** is the prefix for the data (documentation) directory which defaults to */usr/share/doc/libtomcrypt/pdf*.

All four can be used to create custom install locations depending on the nature of the OS and file system in use.

```
make LIBPATH=/home/tom/project/lib INCPATH=/home/tom/project/include \
     DATAPATH=/home/tom/project/docs install
```

This will build the library and install it to the directories under */home/tom/project/*. e.g.

```
/home/tom/project/:
total 1
drwxr-xr-x  2 tom users  80 Jul 30 16:02 docs
drwxr-xr-x  2 tom users 528 Jul 30 16:02 include
drwxr-xr-x  2 tom users  80 Jul 30 16:02 lib

/home/tom/project/docs:
total 452
-rwxr-xr-x  1 tom users 459009 Jul 30 16:02 crypt.pdf

/home/tom/project/include:
```

```
total 132
-rwxr-xr-x  1 tom users  2482 Jul 30 16:02 tomcrypt.h
-rwxr-xr-x  1 tom users   702 Jul 30 16:02 tomcrypt_argchk.h
-rwxr-xr-x  1 tom users  2945 Jul 30 16:02 tomcrypt_cfg.h
-rwxr-xr-x  1 tom users 22763 Jul 30 16:02 tomcrypt_cipher.h
-rwxr-xr-x  1 tom users  5174 Jul 30 16:02 tomcrypt_custom.h
-rwxr-xr-x  1 tom users 11314 Jul 30 16:02 tomcrypt_hash.h
-rwxr-xr-x  1 tom users 11571 Jul 30 16:02 tomcrypt_mac.h
-rwxr-xr-x  1 tom users 13614 Jul 30 16:02 tomcrypt_macros.h
-rwxr-xr-x  1 tom users 14714 Jul 30 16:02 tomcrypt_math.h
-rwxr-xr-x  1 tom users   632 Jul 30 16:02 tomcrypt_misc.h
-rwxr-xr-x  1 tom users 10934 Jul 30 16:02 tomcrypt_pk.h
-rwxr-xr-x  1 tom users  2634 Jul 30 16:02 tomcrypt_pkcs.h
-rwxr-xr-x  1 tom users  7067 Jul 30 16:02 tomcrypt_prng.h
-rwxr-xr-x  1 tom users  1467 Jul 30 16:02 tomcrypt_test.h

/home/tom/project/lib:
total 1073
-rwxr-xr-x  1 tom users 1096284 Jul 30 16:02 libtomcrypt.a
```

14.3 Extra libraries

EXTRALIBS specifies any extra libraries required to link the test programs and shared libraries. They are specified in the notation that GCC expects for global archives.

```
CFLAGS="-DTFM_DESC -DUSE_TFM" EXTRALIBS=-ltfm make install \
                                                    test timing
```

This will install the library using the TomsFastMath library and link the *libtfm.a* library out of the default library search path. The two defines are explained below. You can specify multiple archives (say if you want to support two math libraries, or add on additional code) to the **EXTRALIBS** variable by separating them by a space.

Note that **EXTRALIBS** is not required if you are only making and installing the static library but none of the test programs.

14.4 Building a Static Library

Building a static library is fairly trivial as it only requires one invocation of the GNU make command.

```
CFLAGS="-DTFM_DESC" make install
```

That will build LibTomCrypt (including the TomsFastMath descriptor), and install it in the default locations indicated previously. You can enable the built–in LibTomMath descriptor as well (or in place of the TomsFastMath descriptor). Similarly, you can build the library with no built–in math descriptors.

```
make install
```

In this case, no math descriptors are present in the library and they will have to be made available at build or run time before you can use any of the public key functions.

Note that even if you include the built–in descriptors you must link against the source library as well.

```
gcc -DTFM_DESC myprogram.c -ltomcrypt -ltfm -o myprogram
```

This will compile *myprogram* and link it against the LibTomCrypt library as well as TomsFastMath (which must have been previously installed). Note that we define **TFM_DESC** for compilation. This is so that the TFM descriptor symbol will be defined for the client application to make use of without giving warnings.

14.5 Building a Shared Library

LibTomCrypt can also be built as a shared library through the *makefile.shared* make script. It is similar to use as the static script except that you **must** specify the **EXTRALIBS** variable at install time.

```
CFLAGS="-DTFM_DESC" EXTRALIBS=-ltfm make -f makefile.shared install
```

This will build and install the library and link the shared object against the TomsFastMath library (which must be installed as a shared object as well). The shared build process requires libtool to be installed.

14.6 Header Configuration

The file *tomcrypt_cfg.h* is what lets you control various high level macros which control the behaviour of the library. Build options are also stored in *tomcrypt_custom.h* which allow the enabling and disabling of various algorithms.

ARGTYPE

This lets you control how the LTC_ARGCHK macro will behave. The macro is used to check pointers inside the functions against NULL. There are four settings for ARGTYPE. When set to 0, it will have the default behaviour of printing a message to stderr and raising a SIGABRT signal. This is provided so all platforms that use LibTomCrypt can have an error that functions similarly. When set to 1, it will simply pass on to the assert() macro. When set to 2, the macro will display the error to stderr then return execution to the caller. This could lead to a segmentation fault (e.g. when a pointer is **NULL**) but is useful if you handle signals on your own. When set to 3, it will resolve to a empty macro and no error checking will be performed. Finally, when set to 4, it will return CRYPT_INVALID_ARG to the caller.

Endianness

There are five macros related to endianess issues. For little endian platforms define, **ENDIAN_LITTLE**. For big endian platforms define **ENDIAN_BIG**. Similarly when the default word size of an *unsigned long* is 32-bits define **ENDIAN_32BITWORD** or define **ENDIAN_64BITWORD** when

its 64-bits. If you do not define any of them the library will automatically use **ENDIAN_NEUTRAL** which will work on all platforms.

Currently LibTomCrypt will detect x86-32, x86-64, MIPS R5900, SPARC and SPARC64 running GCC as well as x86-32 running MSVC.

14.7 The Configure Script

There are also options you can specify from the *tomcrypt_custom.h* header file.

14.7.1 X memory routines

At the top of tomcrypt_custom.h are a series of macros denoted as XMALLOC, XCALLOC, XREALLOC, XFREE, and so on. They resolve to the name of the respective functions from the standard C library by default. This lets you substitute in your own memory routines. If you substitute in your own functions they must behave like the standard C library functions in terms of what they expect as input and output.

These macros are handy for working with platforms which do not have a standard C library. For instance, the OLPC[1] bios code uses these macros to redirect to very compact heap and string operations.

14.7.2 X clock routines

The rng_get_bytes() function can call a function that requires the clock() function. These macros let you override the default clock() used with a replacement. By default the standard C library clock() function is used.

14.7.3 LTC_NO_FILE

During the build if LTC_NO_FILE is defined then any function in the library that uses file I/O will not call the file I/O functions and instead simply return CRYPT_NOP. This should help resolve any linker errors stemming from a lack of file I/O on embedded platforms.

14.7.4 LTC_CLEAN_STACK

When this functions is defined the functions that store key material on the stack will clean up afterwards. Assumes that you have no memory paging with the stack.

14.7.5 LTC_TEST

When this has been defined the various self–test functions (for ciphers, hashes, prngs, etc) are included in the build. This is the default configuration. If LTC_NO_TEST has been defined, the testing routines will be compacted and only return CRYPT_NOP.

[1] See http://dev.laptop.org/git?p=bios-crypto;a=summary

14.7.6 LTC_NO_FAST

When this has been defined the library will not use faster word oriented operations. By default, they are only enabled for platforms which can be auto-detected. This macro ensures that they are never enabled.

14.7.7 LTC_FAST

This mode (auto-detected with x86_32,x86_64 platforms with GCC or MSVC) configures various routines such as ctr_encrypt() or cbc_encrypt() that it can safely XOR multiple octets in one step by using a larger data type. This has the benefit of cutting down the overhead of the respective functions.

This mode does have one downside. It can cause unaligned reads from memory if you are not careful with the functions. This is why it has been enabled by default only for the x86 class of processors where unaligned accesses are allowed. Technically LTC_FAST is not *portable* since unaligned accesses are not covered by the ISO C specifications.

In practice however, you can use it on pretty much any platform (even MIPS) with care.

By design the *fast* mode functions won't get unaligned on their own. For instance, if you call ctr_encrypt() right after calling ctr_start() and all the inputs you gave are aligned than ctr_encrypt() will perform aligned memory operations only. However, if you call ctr_encrypt() with an odd amount of plaintext then call it again the CTR pad (the IV) will be partially used. This will cause the ctr routine to first use up the remaining pad bytes. Then if there are enough plaintext bytes left it will use whole word XOR operations. These operations will be unaligned.

The simplest precaution is to make sure you process all data in power of two blocks and handle *remainder* at the end. e.g. If you are CTR'ing a long stream process it in blocks of (say) four kilobytes and handle any remaining incomplete blocks at the end of the stream.

If you do plan on using the *LTC_FAST* mode you have to also define a *LTC_FAST_TYPE* macro which resolves to an optimal sized data type you can perform integer operations with. Ideally it should be four or eight bytes since it must properly divide the size of your block cipher (e.g. 16 bytes for AES). This means sadly if you're on a platform with 57–bit words (or something) you can't use this mode. So sad.

14.7.8 LTC_NO_ASM

When this has been defined the library will not use any inline assembler. Only a few platforms support assembler inlines but various versions of ICC and GCC cannot handle all of the assembler functions.

14.7.9 Symmetric Ciphers, One-way Hashes, PRNGS and Public Key Functions

There are a plethora of macros for the ciphers, hashes, PRNGs and public key functions which are fairly self-explanatory. When they are defined the functionality is included otherwise it is not. There are some dependency issues which are noted in the file. For instance, Yarrow requires CTR chaining mode, a block cipher and a hash function.

Also see technical note number five for more details.

14.7.10 LTC_EASY

When defined the library is configured to build fewer algorithms and modes. Mostly it sticks to NIST and ANSI approved algorithms. See the header file *tomcrypt_custom.h* for more details. It is meant to provide literally an easy method of trimming the library build to the most minimum of useful functionality.

14.7.11 TWOFISH_SMALL and TWOFISH_TABLES

Twofish is a 128-bit symmetric block cipher that is provided within the library. The cipher itself is flexible enough to allow some trade-offs in the implementation. When TWOFISH_SMALL is defined the scheduled symmetric key for Twofish requires only 200 bytes of memory. This is achieved by not pre-computing the substitution boxes. Having this defined will also greatly slow down the cipher. When this macro is not defined Twofish will pre-compute the tables at a cost of 4KB of memory. The cipher will be much faster as a result.

When TWOFISH_TABLES is defined the cipher will use pre-computed (and fixed in code) tables required to work. This is useful when TWOFISH_SMALL is defined as the table values are computed on the fly. When this is defined the code size will increase by approximately 500 bytes. If this is defined but TWOFISH_SMALL is not the cipher will still work but it will not speed up the encryption or decryption functions.

14.7.12 GCM_TABLES

When defined GCM will use a 64KB table (per GCM state) which will greatly speed up the per–packet latency. It also increases the initialization time and is not suitable when you are going to use a key a few times only.

14.7.13 GCM_TABLES_SSE2

When defined GCM will use the SSE2 instructions to perform the $GF(2^x)$ multiply using 16 128–bit XOR operations. It shaves a few cycles per byte of GCM output on both the AMD64 and Intel Pentium 4 platforms. Requires GCC and an SSE2 equipped platform.

14.7.14 LTC_SMALL_CODE

When this is defined some of the code such as the Rijndael and SAFER+ ciphers are replaced with smaller code variants. These variants are slower but can save quite a bit of code space.

14.7.15 LTC_PTHREAD

When this is activated all of the descriptor table functions will use pthread locking to ensure thread safe updates to the tables. Note that it doesn't prevent a thread that is passively using a table from being messed up by another thread that updates the table.

Generally the rule of thumb is to setup the tables once at startup and then leave them be. This added build flag simply makes updating the tables safer.

14.7.16 LTC_ECC_TIMING_RESISTANT

When this has been defined the ECC point multiplier (built–in to the library) will use a timing resistant point multiplication algorithm which prevents leaking key bits of the private key (scalar). It is a slower algorithm but useful for situations where timing side channels pose a significant threat.

14.7.17 Math Descriptors

The library comes with three math descriptors that allow you to interface the public key cryptography API to freely available math libraries. When **GMP_DESC**, **LTM_DESC**, or **TFM_DESC** are defined descriptors for the respective library are built and included in the library as *gmp_desc*, *ltm_desc*, or *tfm_desc* respectively.

In the test demos that use the libraries the additional flags **USE_GMP**, **USE_LTM**, and **USE_TFM** can be defined to tell the program which library to use. Only one of the USE flags can be defined at once.

```
CFLAGS="-DGMP_DESC -DLTM_DESC -DTFM_DESC -DUSE_TFM" \
EXTRALIBS="-lgmp -ltommath -ltfm" make -f makefile.shared install timing
```

That will build and install the library with all descriptors (and link against all), but only use TomsFastMath in the timing demo.

Chapter 15

Optimizations

15.1 Introduction

The entire API was designed with plug and play in mind at the low level. That is you can swap out any cipher, hash, PRNG or bignum library and the dependent API will not require updating. This has the nice benefit that one can add ciphers (etc.) not have to re–write portions of the API. For the most part, LibTomCrypt has also been written to be highly portable and easy to build out of the box on pretty much any platform. As such there are no assembler inlines throughout the code, I make no assumptions about the platform, etc...

That works well for most cases but there are times where performance is of the essence. This API allows optimized routines to be dropped in–place of the existing portable routines. For instance, hand optimized assembler versions of AES could be provided. Any existing function that uses the cipher could automatically use the optimized code without re–writing. This also paves the way for hardware drivers that can access hardware accelerated cryptographic devices.

At the heart of this flexibility is the *descriptor* system. A descriptor is essentially just a C *struct* which describes the algorithm and provides pointers to functions that do the required work. For a given class of operation (e.g. cipher, hash, prng, bignum) the functions of a descriptor have identical prototypes which makes development simple. In most dependent routines all an end developer has to do is register_XXX() the descriptor and they are set.

15.2 Ciphers

The ciphers in LibTomCrypt are accessed through the ltc_cipher_descriptor structure.

```
struct ltc_cipher_descriptor {
   /** name of cipher */
   char *name;

   /** internal ID */
   unsigned char ID;

   /** min keysize (octets) */
   int  min_key_length,
```

```
/** max keysize (octets) */
     max_key_length,

/** block size (octets) */
     block_length,

/** default number of rounds */
     default_rounds;

/** Setup the cipher
   @param key         The input symmetric key
   @param keylen      The length of the input key (octets)
   @param num_rounds  The requested number of rounds (0==default)
   @param skey        [out] The destination of the scheduled key
   @return CRYPT_OK if successful
*/
int  (*setup)(const unsigned char *key,
                               int  keylen,
                               int  num_rounds,
                  symmetric_key *skey);

/** Encrypt a block
   @param pt      The plaintext
   @param ct      [out] The ciphertext
   @param skey    The scheduled key
   @return CRYPT_OK if successful
*/
int (*ecb_encrypt)(const unsigned char *pt,
                         unsigned char *ct,
                         symmetric_key *skey);

/** Decrypt a block
   @param ct      The ciphertext
   @param pt      [out] The plaintext
   @param skey    The scheduled key
   @return CRYPT_OK if successful
*/
int (*ecb_decrypt)(const unsigned char *ct,
                         unsigned char *pt,
                         symmetric_key *skey);

/** Test the block cipher
    @return CRYPT_OK if successful,
            CRYPT_NOP if self-testing has been disabled
*/
int (*test)(void);

/** Terminate the context
   @param skey    The scheduled key
```

```
   */
   void (*done)(symmetric_key *skey);

   /** Determine a key size
       @param keysize    [in/out] The size of the key desired
                                  The suggested size
       @return CRYPT_OK if successful
   */
   int  (*keysize)(int *keysize);

/** Accelerators **/
   /** Accelerated ECB encryption
       @param pt      Plaintext
       @param ct      Ciphertext
       @param blocks  The number of complete blocks to process
       @param skey    The scheduled key context
       @return CRYPT_OK if successful
   */
   int (*accel_ecb_encrypt)(const unsigned char *pt,
                                  unsigned char *ct,
                                  unsigned long  blocks,
                                  symmetric_key *skey);

   /** Accelerated ECB decryption
       @param pt      Plaintext
       @param ct      Ciphertext
       @param blocks  The number of complete blocks to process
       @param skey    The scheduled key context
       @return CRYPT_OK if successful
   */
   int (*accel_ecb_decrypt)(const unsigned char *ct,
                                  unsigned char *pt,
                                  unsigned long  blocks,
                                  symmetric_key *skey);

   /** Accelerated CBC encryption
       @param pt      Plaintext
       @param ct      Ciphertext
       @param blocks  The number of complete blocks to process
       @param IV      The initial value (input/output)
       @param skey    The scheduled key context
       @return CRYPT_OK if successful
   */
   int (*accel_cbc_encrypt)(const unsigned char *pt,
                                  unsigned char *ct,
                                  unsigned long  blocks,
                                  unsigned char *IV,
                                  symmetric_key *skey);

   /** Accelerated CBC decryption
```

```
    @param pt      Plaintext
    @param ct      Ciphertext
    @param blocks  The number of complete blocks to process
    @param IV      The initial value (input/output)
    @param skey    The scheduled key context
    @return CRYPT_OK if successful
*/
int (*accel_cbc_decrypt)(const unsigned char *ct,
                               unsigned char *pt,
                               unsigned long  blocks,
                               unsigned char *IV,
                               symmetric_key *skey);

/** Accelerated CTR encryption
    @param pt      Plaintext
    @param ct      Ciphertext
    @param blocks  The number of complete blocks to process
    @param IV      The initial value (input/output)
    @param mode    little or big endian counter (mode=0 or mode=1)
    @param skey    The scheduled key context
    @return CRYPT_OK if successful
*/
int (*accel_ctr_encrypt)(const unsigned char *pt,
                               unsigned char *ct,
                               unsigned long  blocks,
                               unsigned char *IV,
                                         int  mode,
                               symmetric_key *skey);

/** Accelerated LRW
    @param pt      Plaintext
    @param ct      Ciphertext
    @param blocks  The number of complete blocks to process
    @param IV      The initial value (input/output)
    @param tweak   The LRW tweak
    @param skey    The scheduled key context
    @return CRYPT_OK if successful
*/
int (*accel_lrw_encrypt)(const unsigned char *pt,
                               unsigned char *ct,
                               unsigned long  blocks,
                               unsigned char *IV,
                         const unsigned char *tweak,
                               symmetric_key *skey);

/** Accelerated LRW
    @param ct      Ciphertext
    @param pt      Plaintext
    @param blocks  The number of complete blocks to process
    @param IV      The initial value (input/output)
```

```
    @param tweak   The LRW tweak
    @param skey    The scheduled key context
    @return CRYPT_OK if successful
*/
int (*accel_lrw_decrypt)(const unsigned char *ct,
                               unsigned char *pt,
                               unsigned long  blocks,
                               unsigned char *IV,
                         const unsigned char *tweak,
                               symmetric_key *skey);

/** Accelerated CCM packet (one-shot)
    @param key        The secret key to use
    @param keylen     The length of the secret key (octets)
    @param uskey      A previously scheduled key [can be NULL]
    @param nonce      The session nonce [use once]
    @param noncelen   The length of the nonce
    @param header     The header for the session
    @param headerlen  The length of the header (octets)
    @param pt         [out] The plaintext
    @param ptlen      The length of the plaintext (octets)
    @param ct         [out] The ciphertext
    @param tag        [out] The destination tag
    @param taglen     [in/out] The max size and resulting size
                              of the authentication tag
    @param direction  Encrypt or Decrypt direction (0 or 1)
    @return CRYPT_OK if successful
*/
int (*accel_ccm_memory)(
    const unsigned char *key,    unsigned long keylen,
    symmetric_key       *uskey,
    const unsigned char *nonce,  unsigned long noncelen,
    const unsigned char *header, unsigned long headerlen,
          unsigned char *pt,     unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,    unsigned long *taglen,
                    int  direction);

/** Accelerated GCM packet (one shot)
    @param key        The secret key
    @param keylen     The length of the secret key
    @param IV         The initial vector
    @param IVlen      The length of the initial vector
    @param adata      The additional authentication data (header)
    @param adatalen   The length of the adata
    @param pt         The plaintext
    @param ptlen      The length of the plaintext/ciphertext
    @param ct         The ciphertext
    @param tag        [out] The MAC tag
    @param taglen     [in/out] The MAC tag length
```

```
    @param direction  Encrypt or Decrypt mode (GCM_ENCRYPT or GCM_DECRYPT)
    @return CRYPT_OK on success
*/
int (*accel_gcm_memory)(
    const unsigned char *key,    unsigned long keylen,
    const unsigned char *IV,     unsigned long IVlen,
    const unsigned char *adata,  unsigned long adatalen,
          unsigned char *pt,     unsigned long ptlen,
          unsigned char *ct,
          unsigned char *tag,    unsigned long *taglen,
                    int direction);

/** Accelerated one shot OMAC
    @param key            The secret key
    @param keylen         The key length (octets)
    @param in             The message
    @param inlen          Length of message (octets)
    @param out            [out] Destination for tag
    @param outlen         [in/out] Initial and final size of out
    @return CRYPT_OK on success
*/
int (*omac_memory)(
    const unsigned char *key, unsigned long keylen,
    const unsigned char *in,  unsigned long inlen,
          unsigned char *out, unsigned long *outlen);

/** Accelerated one shot XCBC
    @param key            The secret key
    @param keylen         The key length (octets)
    @param in             The message
    @param inlen          Length of message (octets)
    @param out            [out] Destination for tag
    @param outlen         [in/out] Initial and final size of out
    @return CRYPT_OK on success
*/
int (*xcbc_memory)(
    const unsigned char *key, unsigned long keylen,
    const unsigned char *in,  unsigned long inlen,
          unsigned char *out, unsigned long *outlen);

/** Accelerated one shot F9
    @param key            The secret key
    @param keylen         The key length (octets)
    @param in             The message
    @param inlen          Length of message (octets)
    @param out            [out] Destination for tag
    @param outlen         [in/out] Initial and final size of out
    @return CRYPT_OK on success
    @remark Requires manual padding
*/
```

```
   int (*f9_memory)(
       const unsigned char *key, unsigned long keylen,
       const unsigned char *in,  unsigned long inlen,
             unsigned char *out, unsigned long *outlen);
};
```

15.2.1 Name

The *name* parameter specifies the name of the cipher. This is what a developer would pass to find_cipher() to find the cipher in the descriptor tables.

15.2.2 Internal ID

This is a single byte Internal ID you can use to distinguish ciphers from each other.

15.2.3 Key Lengths

The minimum key length is *min_key_length* and is measured in octets. Similarly the maximum key length is *max_key_length*. They can be equal and both must valid key sizes for the cipher. Values in between are not assumed to be valid though they may be.

15.2.4 Block Length

The size of the ciphers plaintext or ciphertext is *block_length* and is measured in octets.

15.2.5 Rounds

Some ciphers allow different number of rounds to be used. Usually you just use the default. The default round count is *default_rounds*.

15.2.6 Setup

To initialize a cipher (for ECB mode) the function setup() was provided. It accepts an array of key octets *key* of length *keylen* octets. The user can specify the number of rounds they want through *num_rounds* where $num\_rounds = 0$ means use the default. The destination of a scheduled key is stored in *skey*.

Inside the *symmetric_key* union there is a *void *data* which you can use to allocate data if you need a data structure that does not fit with the existing ones provided. Just make sure in your *done()* function that you free the allocated memory.

15.2.7 Single block ECB

To process a single block in ECB mode the ecb_encrypt() and ecb_decrypt() functions were provided. The plaintext and ciphertext buffers are allowed to overlap so you must make sure you do not overwrite the output before you are finished with the input.

15.2.8 Testing

The test() function is used to self–test the *device*. It takes no arguments and returns **CRYPT_OK** if all is working properly. You may return **CRYPT_NOP** to indicate that no testing was performed.

15.2.9 Key Sizing

Occasionally, a function will want to find a suitable key size to use since the input is oddly sized. The keysize() function is for this case. It accepts a pointer to an integer which represents the desired size. The function then has to match it to the exact or a lower key size that is valid for the cipher. For example, if the input is 25 and 24 is valid then it stores 24 back in the pointed to integer. It must not round up and must return an error if the keysize cannot be mapped to a valid key size for the cipher.

15.2.10 Acceleration

The next set of functions cover the accelerated functionality of the cipher descriptor. Any combination of these functions may be set to **NULL** to indicate it is not supported. In those cases the software defaults are used (using the single ECB block routines).

Accelerated ECB

These two functions are meant for cases where a user wants to encrypt (in ECB mode no less) an array of blocks. These functions are accessed through the accel_ecb_encrypt and accel_ecb_decrypt pointers. The *blocks* count is the number of complete blocks to process.

Accelerated CBC

These two functions are meant for accelerated CBC encryption. These functions are accessed through the accel_cbc_encrypt and accel_cbc_decrypt pointers. The *blocks* value is the number of complete blocks to process. The *IV* is the CBC initial vector. It is an input upon calling this function and must be updated by the function before returning.

Accelerated CTR

This function is meant for accelerated CTR encryption. It is accessible through the accel_ctr_encrypt pointer. The *blocks* value is the number of complete blocks to process. The *IV* is the CTR counter vector. It is an input upon calling this function and must be updated by the function before returning. The *mode* value indicates whether the counter is big (mode = CTR_COUNTER_BIG_ENDIAN) or little (mode = CTR_COUNTER_LITTLE_ENDIAN) endian.

This function (and the way it's called) differs from the other two since ctr_encrypt() allows any size input plaintext. The accelerator will only be called if the following conditions are met.

1. The accelerator is present
2. The CTR pad is empty
3. The remaining length of the input to process is greater than or equal to the block size.

The *CTR pad* is empty when a multiple (including zero) blocks of text have been processed. That is, if you pass in seven bytes to AES–CTR mode you would have to pass in a minimum of nine extra bytes before the accelerator could be called. The CTR accelerator must increment the counter (and store it back into the buffer provided) before encrypting it to create the pad.

The accelerator will only be used to encrypt whole blocks. Partial blocks are always handled in software.

Accelerated LRW

These functions are meant for accelerated LRW. They process blocks of input in lengths of multiples of 16 octets. They must accept the *IV* and *tweak* state variables and updated them prior to returning. Note that you may want to disable **LRW_TABLES** in *tomcrypt_custom.h* if you intend to use accelerators for LRW.

While both encrypt and decrypt accelerators are not required it is suggested as it makes lrw_setiv() more efficient.

Note that calling lrw_done() will only invoke the cipher_descriptor[].done() function on the *symmetric_key* parameter of the LRW state. That means if your device requires any (LRW specific) resources you should free them in your ciphers() done function. The simplest way to think of it is to write the plugin solely to do LRW with the cipher. That way cipher_descriptor[].setup() means to init LRW resources and cipher_descriptor[].done() means to free them.

Accelerated CCM

This function is meant for accelerated CCM encryption or decryption. It processes the entire packet in one call. You can optimize the work flow somewhat by allowing the caller to call the setup() function first to schedule the key if your accelerator cannot do the key schedule on the fly (for instance). This function MUST support both key passing methods.

key	uskey	Source of key
NULL	NULL	Error, not supported
non-NULL	NULL	Use key, do a key schedule
NULL	non-NULL	Use uskey, key schedule not required
non-NULL	non-NULL	Use uskey, key schedule not required

This function is called when the user calls ccm_memory().

Accelerated GCM

This function is meant for accelerated GCM encryption or decryption. It processes the entire packet in one call. Note that the setup() function will not be called prior to this. This function must handle scheduling the key provided on its own. It is called when the user calls gcm_memory().

Accelerated OMAC

This function is meant to perform an optimized OMAC1 (CMAC) message authentication code computation when the user calls omac_memory().

Accelerated XCBC-MAC

This function is meant to perform an optimized XCBC-MAC message authentication code computation when the user calls xcbc_memory().

Accelerated F9

This function is meant to perform an optimized F9 message authentication code computation when the user calls f9_memory(). Like f9_memory(), it requires the caller to perform any 3GPP related padding before calling in order to ensure proper compliance with F9.

15.3 One–Way Hashes

The hash functions are accessed through the ltc_hash_descriptor structure.

```
struct ltc_hash_descriptor {
    /** name of hash */
    char *name;

    /** internal ID */
    unsigned char ID;

    /** Size of digest in octets */
    unsigned long hashsize;

    /** Input block size in octets */
    unsigned long blocksize;

    /** ASN.1 OID */
    unsigned long OID[16];

    /** Length of DER encoding */
    unsigned long OIDlen;

    /** Init a hash state
      @param hash   The hash to initialize
      @return CRYPT_OK if successful
    */
    int (*init)(hash_state *hash);

    /** Process a block of data
      @param hash   The hash state
      @param in     The data to hash
      @param inlen  The length of the data (octets)
      @return CRYPT_OK if successful
    */
    int (*process)(         hash_state *hash,
                   const unsigned char *in,
                         unsigned long  inlen);
```

```
    /** Produce the digest and store it
      @param hash   The hash state
      @param out    [out] The destination of the digest
      @return CRYPT_OK if successful
    */
    int (*done)(   hash_state *hash,
                unsigned char *out);

    /** Self-test
      @return CRYPT_OK if successful,
              CRYPT_NOP if self-tests have been disabled
    */
    int (*test)(void);

    /* accelerated hmac callback: if you need to-do
       multiple packets just use the generic hmac_memory
       and provide a hash callback
    */
    int  (*hmac_block)(const unsigned char *key,
                               unsigned long  keylen,
                       const unsigned char *in,
                               unsigned long  inlen,
                               unsigned char *out,
                               unsigned long *outlen);
};
```

15.3.1 Name

This is the name the hash is known by and what find_hash() will look for.

15.3.2 Internal ID

This is the internal ID byte used to distinguish the hash from other hashes.

15.3.3 Digest Size

The *hashsize* variable indicates the length of the output in octets.

15.3.4 Block Size

The *blocksize* variable indicates the length of input (in octets) that the hash processes in a given invocation.

15.3.5 OID Identifier

This is the universal ASN.1 Object Identifier for the hash.

15.3.6 Initialization

The init function initializes the hash and prepares it to process message bytes.

15.3.7 Process

This processes message bytes. The algorithm must accept any length of input that the hash would allow. The input is not guaranteed to be a multiple of the block size in length.

15.3.8 Done

The done function terminates the hash and returns the message digest.

15.3.9 Acceleration

A compatible accelerator must allow processing data in any granularity which may require internal padding on the driver side.

15.3.10 HMAC Acceleration

The hmac_block() callback is meant for single–shot optimized HMAC implementations. It is called directly by hmac_memory() if present. If you need to be able to process multiple blocks per MAC then you will have to simply provide a process() callback and use hmac_memory() as provided in LibTomCrypt.

15.4 Pseudo–Random Number Generators

The pseudo–random number generators are accessible through the ltc_prng_descriptor structure.

```
struct ltc_prng_descriptor {
    /** Name of the PRNG */
    char *name;

    /** size in bytes of exported state */
    int  export_size;

    /** Start a PRNG state
        @param prng   [out] The state to initialize
        @return CRYPT_OK if successful
    */
    int (*start)(prng_state *prng);

    /** Add entropy to the PRNG
        @param in         The entropy
        @param inlen      Length of the entropy (octets)
        @param prng       The PRNG state
        @return CRYPT_OK if successful
    */
```

```
int (*add_entropy)(const unsigned char *in,
                         unsigned long  inlen,
                            prng_state *prng);

/** Ready a PRNG state to read from
    @param prng       The PRNG state to ready
    @return CRYPT_OK if successful
*/
int (*ready)(prng_state *prng);

/** Read from the PRNG
    @param out     [out] Where to store the data
    @param outlen  Length of data desired (octets)
    @param prng    The PRNG state to read from
    @return Number of octets read
*/
unsigned long (*read)(unsigned char *out,
                      unsigned long  outlen,
                         prng_state *prng);

/** Terminate a PRNG state
    @param prng   The PRNG state to terminate
    @return CRYPT_OK if successful
*/
int (*done)(prng_state *prng);

/** Export a PRNG state
    @param out     [out] The destination for the state
    @param outlen  [in/out] The max size and resulting size
    @param prng    The PRNG to export
    @return CRYPT_OK if successful
*/
int (*pexport)(unsigned char *out,
               unsigned long *outlen,
                  prng_state *prng);

/** Import a PRNG state
    @param in      The data to import
    @param inlen   The length of the data to import (octets)
    @param prng    The PRNG to initialize/import
    @return CRYPT_OK if successful
*/
int (*pimport)(const unsigned char *in,
                     unsigned long  inlen,
                        prng_state *prng);

/** Self-test the PRNG
    @return CRYPT_OK if successful,
            CRYPT_NOP if self-testing has been disabled
*/
```

```
    int (*test)(void);
};
```

15.4.1 Name

The name by which find_prng() will find the PRNG.

15.4.2 Export Size

When an PRNG state is to be exported for future use you specify the space required in this variable.

15.4.3 Start

Initialize the PRNG and make it ready to accept entropy.

15.4.4 Entropy Addition

Add entropy to the PRNG state. The exact behaviour of this function depends on the particulars of the PRNG.

15.4.5 Ready

This function makes the PRNG ready to read from by processing the entropy added. The behaviour of this function depends on the specific PRNG used.

15.4.6 Read

Read from the PRNG and return the number of bytes read. This function does not have to fill the buffer but it is best if it does as many protocols do not retry reads and will fail on the first try.

15.4.7 Done

Terminate a PRNG state. The behaviour of this function depends on the particular PRNG used.

15.4.8 Exporting and Importing

An exported PRNG state is data that the PRNG can later import to resume activity. They're not meant to resume *the same session* but should at least maintain the same level of state entropy.

15.5 BigNum Math Descriptors

The library also makes use of the math descriptors to access math functions. While bignum math libraries usually differ in implementation it hasn't proven hard to write *glue* to use math libraries so far. The basic descriptor looks like.

```
/** math descriptor */
typedef struct {
   /** Name of the math provider */
   char *name;

   /** Bits per digit, amount of bits must fit in an unsigned long */
   int  bits_per_digit;

/* ---- init/deinit functions ---- */

   /** initialize a bignum
     @param   a     The number to initialize
     @return  CRYPT_OK on success
   */
   int (*init)(void **a);

   /** init copy
     @param  dst    The number to initialize and write to
     @param  src    The number to copy from
     @return CRYPT_OK on success
   */
   int (*init_copy)(void **dst, void *src);

   /** deinit
      @param   a    The number to free
      @return CRYPT_OK on success
   */
   void (*deinit)(void *a);

/* ---- data movement ---- */

   /** copy
      @param   src   The number to copy from
      @param   dst   The number to write to
      @return CRYPT_OK on success
   */
   int (*copy)(void *src, void *dst);

/* ---- trivial low level functions ---- */

   /** set small constant
      @param a    Number to write to
      @param n    Source upto bits_per_digit (meant for small constants)
      @return CRYPT_OK on success
   */
   int (*set_int)(void *a, unsigned long n);

   /** get small constant
      @param a  Small number to read
      @return   The lower bits_per_digit of the integer (unsigned)
```

```
*/
unsigned long (*get_int)(void *a);

/** get digit n
  @param a  The number to read from
  @param n  The number of the digit to fetch
  @return  The bits_per_digit  sized n'th digit of a
*/
unsigned long (*get_digit)(void *a, int n);

/** Get the number of digits that represent the number
  @param a   The number to count
  @return The number of digits used to represent the number
*/
int (*get_digit_count)(void *a);

/** compare two integers
  @param a   The left side integer
  @param b   The right side integer
  @return LTC_MP_LT if a < b,
          LTC_MP_GT if a > b and
          LTC_MP_EQ otherwise.  (signed comparison)
*/
int (*compare)(void *a, void *b);

/** compare against int
  @param a   The left side integer
  @param b   The right side integer (upto bits_per_digit)
  @return LTC_MP_LT if a < b,
          LTC_MP_GT if a > b and
          LTC_MP_EQ otherwise.  (signed comparison)
*/
int (*compare_d)(void *a, unsigned long n);

/** Count the number of bits used to represent the integer
  @param a   The integer to count
  @return The number of bits required to represent the integer
*/
int (*count_bits)(void * a);

/** Count the number of LSB bits which are zero
  @param a   The integer to count
  @return The number of contiguous zero LSB bits
*/
int (*count_lsb_bits)(void *a);

/** Compute a power of two
  @param a  The integer to store the power in
  @param n  The power of two you want to store (a = 2^n)
  @return CRYPT_OK on success
```

```
   */
   int (*twoexpt)(void *a , int n);

/* ---- radix conversions ---- */

   /** read ascii string
     @param a     The integer to store into
     @param str   The string to read
     @param radix The radix the integer has been represented in (2-64)
     @return CRYPT_OK on success
   */
   int (*read_radix)(void *a, const char *str, int radix);

   /** write number to string
     @param a     The integer to store
     @param str   The destination for the string
     @param radix The radix the integer is to be represented in (2-64)
     @return CRYPT_OK on success
   */
   int (*write_radix)(void *a, char *str, int radix);

   /** get size as unsigned char string
     @param a  The integer to get the size
     @return   The length of the integer in octets
   */
   unsigned long (*unsigned_size)(void *a);

   /** store an integer as an array of octets
     @param src   The integer to store
     @param dst   The buffer to store the integer in
     @return CRYPT_OK on success
   */
   int (*unsigned_write)(void *src, unsigned char *dst);

   /** read an array of octets and store as integer
     @param dst   The integer to load
     @param src   The array of octets
     @param len   The number of octets
     @return CRYPT_OK on success
   */
   int (*unsigned_read)(         void *dst,
                        unsigned char *src,
                        unsigned long  len);

/* ---- basic math ---- */

   /** add two integers
     @param a   The first source integer
     @param b   The second source integer
     @param c   The destination of "a + b"
```

```
  @return CRYPT_OK on success
*/
int (*add)(void *a, void *b, void *c);

/** add two integers
  @param a   The first source integer
  @param b   The second source integer
             (single digit of upto bits_per_digit in length)
  @param c   The destination of "a + b"
  @return CRYPT_OK on success
*/
int (*addi)(void *a, unsigned long b, void *c);

/** subtract two integers
  @param a   The first source integer
  @param b   The second source integer
  @param c   The destination of "a - b"
  @return CRYPT_OK on success
*/
int (*sub)(void *a, void *b, void *c);

/** subtract two integers
  @param a   The first source integer
  @param b   The second source integer
             (single digit of upto bits_per_digit in length)
  @param c   The destination of "a - b"
  @return CRYPT_OK on success
*/
int (*subi)(void *a, unsigned long b, void *c);

/** multiply two integers
  @param a   The first source integer
  @param b   The second source integer
             (single digit of upto bits_per_digit in length)
  @param c   The destination of "a * b"
  @return CRYPT_OK on success
*/
int (*mul)(void *a, void *b, void *c);

/** multiply two integers
  @param a   The first source integer
  @param b   The second source integer
             (single digit of upto bits_per_digit in length)
  @param c   The destination of "a * b"
  @return CRYPT_OK on success
*/
int (*muli)(void *a, unsigned long b, void *c);

/** Square an integer
  @param a    The integer to square
```

```
   @param b    The destination
   @return CRYPT_OK on success
*/
int (*sqr)(void *a, void *b);

/** Divide an integer
   @param a    The dividend
   @param b    The divisor
   @param c    The quotient (can be NULL to signify don't care)
   @param d    The remainder (can be NULL to signify don't care)
   @return CRYPT_OK on success
*/
int (*div)(void *a, void *b, void *c, void *d);

/** divide by two
   @param  a   The integer to divide (shift right)
   @param  b   The destination
   @return CRYPT_OK on success
*/
int (*div_2)(void *a, void *b);

/** Get remainder (small value)
   @param  a    The integer to reduce
   @param  b    The modulus (upto bits_per_digit in length)
   @param  c    The destination for the residue
   @return CRYPT_OK on success
*/
int (*modi)(void *a, unsigned long b, unsigned long *c);

/** gcd
   @param  a     The first integer
   @param  b     The second integer
   @param  c     The destination for (a, b)
   @return CRYPT_OK on success
*/
int (*gcd)(void *a, void *b, void *c);

/** lcm
   @param  a     The first integer
   @param  b     The second integer
   @param  c     The destination for [a, b]
   @return CRYPT_OK on success
*/
int (*lcm)(void *a, void *b, void *c);

/** Modular multiplication
   @param  a     The first source
   @param  b     The second source
   @param  c     The modulus
   @param  d     The destination (a*b mod c)
```

```
      @return CRYPT_OK on success
   */
   int (*mulmod)(void *a, void *b, void *c, void *d);

   /** Modular squaring
      @param  a     The first source
      @param  b     The modulus
      @param  c     The destination (a*a mod b)
      @return CRYPT_OK on success
   */
   int (*sqrmod)(void *a, void *b, void *c);

   /** Modular inversion
      @param  a     The value to invert
      @param  b     The modulus
      @param  c     The destination (1/a mod b)
      @return CRYPT_OK on success
   */
   int (*invmod)(void *, void *, void *);

/* ---- reduction ---- */

   /** setup Montgomery
       @param a  The modulus
       @param b  The destination for the reduction digit
       @return CRYPT_OK on success
   */
   int (*montgomery_setup)(void *a, void **b);

   /** get normalization value
       @param a   The destination for the normalization value
       @param b   The modulus
       @return  CRYPT_OK on success
   */
   int (*montgomery_normalization)(void *a, void *b);

   /** reduce a number
       @param a   The number [and dest] to reduce
       @param b   The modulus
       @param c   The value "b" from montgomery_setup()
       @return CRYPT_OK on success
   */
   int (*montgomery_reduce)(void *a, void *b, void *c);

   /** clean up  (frees memory)
       @param a   The value "b" from montgomery_setup()
       @return CRYPT_OK on success
   */
   void (*montgomery_deinit)(void *a);
```

```
/* ---- exponentiation ---- */

   /** Modular exponentiation
       @param a    The base integer
       @param b    The power (can be negative) integer
       @param c    The modulus integer
       @param d    The destination
       @return CRYPT_OK on success
   */
   int (*exptmod)(void *a, void *b, void *c, void *d);

   /** Primality testing
       @param a     The integer to test
       @param b     The destination of the result (FP_YES if prime)
       @return CRYPT_OK on success
   */
   int (*isprime)(void *a, int *b);

/* ----  (optional) ecc point math ---- */

   /** ECC GF(p) point multiplication (from the NIST curves)
       @param k   The integer to multiply the point by
       @param G   The point to multiply
       @param R   The destination for kG
       @param modulus  The modulus for the field
       @param map Boolean indicated whether to map back to affine or not
                  (can be ignored if you work in affine only)
       @return CRYPT_OK on success
   */
   int (*ecc_ptmul)(     void *k,
                    ecc_point *G,
                    ecc_point *R,
                         void *modulus,
                          int  map);

   /** ECC GF(p) point addition
       @param P    The first point
       @param Q    The second point
       @param R    The destination of P + Q
       @param modulus  The modulus
       @param mp   The "b" value from montgomery_setup()
       @return CRYPT_OK on success
   */
   int (*ecc_ptadd)(ecc_point *P,
                    ecc_point *Q,
                    ecc_point *R,
                         void *modulus,
                         void *mp);

   /** ECC GF(p) point double
```

```
      @param P    The first point
      @param R    The destination of 2P
      @param modulus  The modulus
      @param mp   The "b" value from montgomery_setup()
      @return CRYPT_OK on success
   */
   int (*ecc_ptdbl)(ecc_point *P,
                    ecc_point *R,
                         void *modulus,
                         void *mp);

   /** ECC mapping from projective to affine,
       currently uses (x,y,z) => (x/z^2, y/z^3, 1)
       @param P     The point to map
       @param modulus The modulus
       @param mp    The "b" value from montgomery_setup()
       @return CRYPT_OK on success
       @remark The mapping can be different but keep in mind a
               ecc_point only has three integers (x,y,z) so if
               you use a different mapping you have to make it fit.
   */
   int (*ecc_map)(ecc_point *P, void *modulus, void *mp);

   /** Computes kA*A + kB*B = C using Shamir's Trick
       @param A        First point to multiply
       @param kA       What to multiple A by
       @param B        Second point to multiply
       @param kB       What to multiple B by
       @param C        [out] Destination point (can overlap with A or B)
       @param modulus  Modulus for curve
       @return CRYPT_OK on success
   */
   int (*ecc_mul2add)(ecc_point *A, void *kA,
                      ecc_point *B, void *kB,
                      ecc_point *C,
                           void *modulus);


/* ---- (optional) rsa optimized math (for internal CRT) ---- */

   /** RSA Key Generation
       @param prng     An active PRNG state
       @param wprng    The index of the PRNG desired
       @param size     The size of the key in octets
       @param e        The "e" value (public key).
                       e==65537 is a good choice
       @param key      [out] Destination of a newly created private key pair
       @return CRYPT_OK if successful, upon error all allocated ram is freed
    */
    int (*rsa_keygen)(prng_state *prng,
```

```
                          int  wprng,
                          int  size,
                         long  e,
                      rsa_key *key);

   /** RSA exponentiation
      @param in       The octet array representing the base
      @param inlen    The length of the input
      @param out      The destination (to be stored in an octet array format)
      @param outlen   The length of the output buffer and the resulting size
                      (zero padded to the size of the modulus)
      @param which    PK_PUBLIC for public RSA and PK_PRIVATE for private RSA
      @param key      The RSA key to use
      @return CRYPT_OK on success
   */
   int (*rsa_me)(const unsigned char *in,   unsigned long inlen,
                       unsigned char *out,  unsigned long *outlen, int which,
                       rsa_key *key);
} ltc_math_descriptor;
```

Most of the functions are fairly straightforward and do not need documentation. We'll cover the basic conventions of the API and then explain the accelerated functions.

15.5.1 Conventions

All *bignums* are accessed through an opaque *void ** data type. You must internally cast the pointer if you need to access members of your bignum structure. During the init calls a *void *** will be passed where you allocate your structure and set the pointer then initialize the number to zero. During the deinit calls you must free the bignum as well as the structure you allocated to place it in.

All functions except the Montgomery reductions work from left to right with the arguments. For example, mul(a, b, c) computes $c \leftarrow ab$.

All functions (except where noted otherwise) return **CRYPT_OK** to signify a successful operation. All error codes must be valid LibTomCrypt error codes.

The digit routines (including functions with the *i* suffix) use a *unsigned long* to represent the digit. If your internal digit is larger than this you must then partition your digits. Normally this does not matter as *unsigned long* will be the same size as your register size. Note that if your digit is smaller than an *unsigned long* that is also acceptable as the *bits_per_digit* parameter will specify this.

15.5.2 ECC Functions

The ECC system in LibTomCrypt is based off of the NIST recommended curves over $GF(p)$ and is used to implement EC-DSA and EC-DH. The ECC functions work with the **ecc_point** structure and assume the points are stored in Jacobian projective format.

```
/** A point on a ECC curve, stored in Jacobian format such
    that (x,y,z) => (x/z^2, y/z^3, 1) when interpreted as affine */
```

```
typedef struct {
    /** The x co-ordinate */
    void *x;
    /** The y co-ordinate */
    void *y;
    /** The z co-ordinate */
    void *z;
} ecc_point;
```

All ECC functions must use this mapping system. The only exception is when you remap all ECC callbacks which will allow you to have more control over how the ECC math will be implemented. Out of the box you only have three parameters per point to use (x, y, z) however, these are just void pointers. They could point to anything you want. The only further exception is the export functions which expects the values to be in affine format.

Point Multiply

This will multiply the point G by the scalar k and store the result in the point R. The value should be mapped to affine only if *map* is set to one.

Point Addition

This will add the point P to the point Q and store it in the point R. The *mp* parameter is the b value from the montgomery_setup() call. The input points may be in either affine (with $z = 1$) or projective format and the output point is always projective.

Point Mapping

This will map the point P back from projective to affine. The output point P must be of the form $(x, y, 1)$.

Shamir's Trick

To accelerate EC–DSA verification the library provides a built–in function called ltc_ecc_mul2add(). This performs two point multiplications and an addition in roughly the time of one point multiplication. It is called from ecc_verify_hash() if an accelerator is not present. The acclerator function must allow the points to overlap (e.g., $A \leftarrow k_1 A + k_2 B$) and must return the final point in affine format.

15.5.3 RSA Functions

The RSA Modular Exponentiation (ME) function is used by the RSA API to perform exponentiations for private and public key operations. In particular for private key operations it uses the CRT approach to lower the time required. It is passed an RSA key with the following format.

```
/** RSA PKCS style key */
typedef struct Rsa_key {
    /** Type of key, PK_PRIVATE or PK_PUBLIC */
```

```
    int type;
    /** The public exponent */
    void *e;
    /** The private exponent */
    void *d;
    /** The modulus */
    void *N;
    /** The p factor of N */
    void *p;
    /** The q factor of N */
    void *q;
    /** The 1/q mod p CRT param */
    void *qP;
    /** The d mod (p - 1) CRT param */
    void *dP;
    /** The d mod (q - 1) CRT param */
    void *dQ;
} rsa_key;
```

The call reads the *in* buffer as an unsigned char array in big endian format. Then it performs the exponentiation and stores the output in big endian format to the *out* buffer. The output must be zero padded (leading bytes) so that the length of the output matches the length of the modulus (in bytes). For example, for RSA–1024 the output is always 128 bytes regardless of how small the numerical value of the exponentiation is.

Since the function is given the entire RSA key (for private keys only) CRT is possible as prescribed in the PKCS #1 v2.1 specification.

Index