LibTomCrypt
Version 1.02

Tom St Denis

tomstdenis@gmail.com
http://libtomcrypt.org

April 19, 2005

Open Source. Open Academia. Open Minds.

Tom St Denis,
Phone: 1-613-836-3160
111 Banning Rd
Kanata, Ontario
K2L 1C3
Canada

Contents

Chapter 1

Introduction

1.1 What is the LibTomCrypt?

LibTomCrypt is a portable ISO C cryptographic library that is meant to be a toolset for cryptographers who are designing a cryptosystem. 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 runtime 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 runtime 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.591 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) have the **exact** 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 my library you know how to use Safer+ or RC6 or Serpent or ... 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 runtime. That means your application can support all ciphers/hashes/prngs without changing the source code.

Not only did I strive to make a consistent and simple API to work with but I also strived 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 strived 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 and pseudo–random number generators (PRNG) 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. Similiarly 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

All of the source code except for the following files have been written by the author or donated to the project under a public domain license:

1. rc2.c

'mpi.c" was originally written by Michael Fromberger (sting@linguist.dartmouth.edu) but has since been replaced with my LibTomMath library which is public domain.

"rc2.c" is based on publicly available code that is not attributed to a person from the given source.

The project is hereby released as public domain.

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 authors 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 occured, %s\n", error_to_string(err));
      /* perform error handling */
   }
   /* continue on if no error occured */
}
```

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", "ctype.h" and "ltc_tommath.h" (the bignum library routines).

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	Swaps byte order (32–bits only)

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	$x >> y, 0 \le y \le 63$

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[1] where the output will be stored. For example:

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

[1]Extensive error checking is not in place but it will be in future releases so it is a good idea to follow through with these guidelines.

```
    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.

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.

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 it 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's 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 provide five visible functions which are (given that XXX is the name of the cipher):

```
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 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).

3.1.2 ECB Encryption and Decryption

To encrypt or decrypt a block in ECB mode there are these two function classes

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

void 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] and store the result where you want it. 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. To test a particular cipher against test vectors[4] call the 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:

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

Essentially it will round the input keysize in "keysize" down to the next appropriate key size. This function 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;
}
```

This should indicate a keysize of sixteen bytes is suggested.

3.1.5 Cipher Termination

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

[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.

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

For the software based ciphers within LibTomCrypt this function will not do anything. However, user supplied cipher descriptors may require calls to it for resource management. 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 is below.

```
#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. 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're supposed to be paranoid. It's 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 format:

```
struct _cipher_descriptor {
   char *name;
   unsigned char ID;
   int  min_key_length,
        max_key_length,
        block_length,
        default_rounds;
   int  (*setup)(const unsigned char *key, int keylen, int num_rounds, symmetric_key *skey);
   void (*ecb_encrypt)(const unsigned char *pt, unsigned char *ct, symmetric_key *skey);
   void (*ecb_decrypt)(const unsigned char *ct, unsigned char *pt, symmetric_key *skey);
   int (*test)(void);
   void (*done)(symmetric_key *skey);
   int  (*keysize)(int *keysize);

   void (*accel_ecb_encrypt)(const unsigned char *pt,
                                 unsigned char *ct,
                                 unsigned long blocks, symmetric_key *skey);
   void (*accel_ecb_decrypt)(const unsigned char *ct,
                                 unsigned char *pt,
                                 unsigned long blocks, symmetric_key *skey);
   void (*accel_cbc_encrypt)(const unsigned char *pt,
                                 unsigned char *ct,
                                 unsigned long blocks, unsigned char *IV,
                                 symmetric_key *skey);
   void (*accel_cbc_decrypt)(const unsigned char *ct,
                                 unsigned char *pt,
                                 unsigned long blocks, unsigned char *IV,
                                 symmetric_key *skey);
   void (*accel_ctr_encrypt)(const unsigned char *pt,
                                 unsigned char *ct,
                                 unsigned long blocks, unsigned char *IV,
```

```
                                  int mode, symmetric_key *skey);
   void (*accel_ccm_memory)(
       const unsigned char *key,    unsigned long keylen,
       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);

};
```

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.

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

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	7	16
3DES (EDE mode)	des3_desc	8	21	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

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 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 keysetup, larger code.
defined	undefined	Very slow, 0.2KB of ram.
defined	defined	Faster, 0.2KB of ram, larger code.

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 negative one 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 it otherwise it returns **CRYPT_ERROR**. Consider:

```
#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 only Rijndael only.

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 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 ECB mode for obvious reasons):

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

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.

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

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

3.4.4 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 you simply 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.

3.4.5 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.

3.4.6 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.

3.4.7 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, 128 bits) */
                          0, /* 0 == default # of rounds */
                       &ctr)  /* where to store initialized 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 data to encrypt */
                                   &ctr)  /* previously initialized 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 data to encrypt */
                                   &ctr)  /* previously initialized 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;
}
```

3.5 Encrypt and Authenticate Modes

3.5.1 EAX Mode

LibTomCrypt provides support for a mode called EAX[6] 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 I explain how to use it. EAX is a mode that requires a cipher, CTR and OMAC support and provides encryption and authentication[7]. It is initialized with a random "nonce" that can be shared publicly as well as a "header" which can be fixed and public as well as 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 though 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. "cipher" is the index of the desired cipher in the descriptor table. "key" is the shared secret symmetric key of length "keylen". "nonce" is the random public string of length "noncelen". "header" is the random (or fixed or **NULL**) header for the message of length "headerlen".

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

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

When this function completes "eax" will be initialized such that you can now either have data decrypted or encrypted in EAX mode. Note that 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);
```

The function "eax_encrypt" will encrypt the bytes in "pt" of "length" bytes and store the ciphertext in "ct". Note that "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 that 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" bytes from "header" to the given "eax" stream. 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 upto "taglen" bytes of the message tag in "tag". The function then stores how many bytes of the tag were written out back into "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,  /* the context */
               find_cipher("rijndael"),  /* cipher we want to use */
                                 nonce,  /* our state nonce */
                                    16,  /* none is 16 bytes */
                               "TestApp",  /* example header, identifies this program */
                                     7)  /* length of the header */
       ) != 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[8] . 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

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

initialize two "ocb" states (with different nonces). Also "pt" and "ct" may point to the same location in memory.

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 upto "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,
    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 is "key" with a length of "keylen" octets. 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 copy of CCM cannot handle a header or plaintext 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.

3.5.4 GCM Mode

Galois counter mode is an IEEE proposal for authenticated encryption. Like EAX and OCB 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 sixteen byte block.

A GCM stream is meant to be processed in three modes each one sequential serial. 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.

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 that the GCM protocols provides a "shortcut" for 12–byte IVs where no preprocessing is to be done. If you want to minimize per packet latency it's ideal to only use 12–byte IVs. You can just increment it like a counter for each packet and the CTR [privacy] will be ensured.

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 that once you begin adding AAD to the GCM state you cannot return to adding IV data until the state is 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_add_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;
   }

   /* 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);
```

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:

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

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

This will return **CRYPTO_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 too. 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);
};
```

Similarly "name" is the name of the hash function in ASCII (all lowercase). "hashsize" is the size of the digest output in bytes. The remaining fields are pointers to the functions that do the respective tasks. 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.

You can use the table to indirectly call a hash function that is chosen at runtime. 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 as well:

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

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

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

The "hash" parameter is the location in the descriptor table of the hash (*e.g. the return of find_hash()*). The "*outlen" variable is used to keep track of the output size. You must set it to the size of your output buffer before calling the functions. When they complete succesfully they store the length of

the message digest back in it. The functions are otherwise straightforward. The "hash_filehandle" function assumes that "in" is an file handle opened in binary mode. It will hash to the end of file and not reset the file position when finished.

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;
}
```

The following hashes are provided as of this release:

Name	Descriptor Name	Size of Message Digest (bytes)
WHIRLPOOL	whirlpool_desc	64
SHA-512	sha512_desc	64
SHA-384	sha384_desc	48
SHA-256	sha256_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

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);
```

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 pro-

vided 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 authenication codes or HMAC for short. An HMAC of a message is a keyed authenication 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 chars. 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. "hash" is the index into the descriptor table of the hash you want to use to authenticate the message. "key" is the pointer to the array of chars that make up the key. "keylen" 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. "buf" is the array of octets to send into the HMAC process. "len" is the number of octets to process. Like the hash process routines you can send the data in arbitrarly 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);
```

"hmac" is the HMAC state you are working with. "out" 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 todo. 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. From an API standpoint the OMAC routines work much like the HMAC routines do. 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, unsigned long keylen);
```

The "omac" variable is the state for the OMAC algorithm. "cipher" is the index into the cipher_descriptor table of the cipher[2] you wish to use. "key" and "keylen" 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,

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

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

```
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

```
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

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

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

```
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 omac_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 omac_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;
}
```

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 **CRYPTO_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 **CRYPTO_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 startup 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. Its 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, 10, &prng));
   return 0;
}
```

6.2 PRNG Descriptors

PRNGs have descriptors too (surprised?). 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);
};
```

There is a "int find_prng(char *name)" function as well. Returns -1 if the PRNG is not found, otherwise it returns the position in the prng_descriptor array.

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:

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

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 taskings 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.

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

RC4

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

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

You really shouldn't use RC4 anymore. This isn't because RC4 is weak (though biases are known to exist) just simply 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 that this cipher has several oddities about how it operates. The first time you call add_entropy() that sets the cipher's key. Every other time you call the same function it 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>
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 except that it doesn't 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 beasts that try hard not to stray from pre-determined paths. That makes gathering entropy needed to seed the 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. Its 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 three bytes a second. The callback pointer may be set to **NULL** to avoid using it if you don't 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 setup 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 doesn't work well. Example usage of this function is given below.

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

```
#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;
}
```

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 "v1.5" and "v2.0" algorithms. To simplify matters a little the v2.0 encryption and signature padding algorithms are called OAEP and PSS respectively.

7.2 PKCS #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.

7.2.1 OAEP Encoding

```
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.

7.2.2 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.

7.2.3 PKCS #1 v1.5 Encoding

```
int pkcs_1_v15_es_encode(const unsigned char *msg,    unsigned long msglen,
                               unsigned long  modulus_bitlen,
                               prng_state    *prng,   int           prng_idx,
                               unsigned char *out,    unsigned long *outlen);
```

This will PKCS v1.5 encode the data in "msg" of length "msglen". Pass the length (in bits) of your RSA modulus in "modulus_bitlen". The encoded data will be stored in "out" of length "outlen".

7.2.4 PKCS #1 v1.5 Decoding

```
int pkcs_1_v15_es_decode(const unsigned char *msg,  unsigned long msglen,
                               unsigned long modulus_bitlen,
                               unsigned char *out,  unsigned long outlen,
                               int           *res);
```

This will PKCS v1.5 decode the message in "msg" of length "msglen". It will store the output in "out". Note that the length of the output "outlen" is a constant. This decoder cannot determine the original message length. If the data in "msg" is a valid packet then a 1 is stored in "res", otherwise a 0 is stored.

7.3 PKCS #1 Digital Signatures

7.3.1 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().

7.3.2 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.

7.3.3 PKCS #1 v1.5 Encoding

```
int pkcs_1_v15_sa_encode(const unsigned char *msghash,  unsigned long msghashlen,
                               int            hash_idx, unsigned long modulus_bitlen,
                               unsigned char *out,      unsigned long *outlen);
```

This will PKCS #1 v1.5 signature encode the message hash "msghash" of length "msghashlen". You have to tell this routine which hash produced the message hash in "hash_idx". The encoded hash is stored in "out" of length "outlen".

7.3.4 PKCS #1 v1.5 Decoding

```
int pkcs_1_v15_sa_decode(const unsigned char *msghash, unsigned long msghashlen,
                         const unsigned char *sig,     unsigned long siglen,
                               int           hash_idx, unsigned long modulus_bitlen,
                               int          *res);
```

This will PKCS #1 v1.5 signature decode the data in "sig" of length "siglen" and compare the extracted hash against "msghash" of length "msghashlen". You have to tell this routine which hash produced the message digest in "hash_idx". If the packet is valid and the hashes match "res" is set to 1. Otherwise, it is set to 0.

7.4 RSA Operations

7.4.1 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.

7.4.2 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. "size" is the size in bytes of the RSA modulus desired. "e" is the encryption exponent desired, typical values are 3, 17, 257 and 65537. I suggest you stick with 65537 since its big enough to prevent trivial math attacks and not super slow. "key" is where the 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 that the "rsa_make_key()" function allocates memory at runtime 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 ram allocated itself.

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. The second type is a public RSA key which only includes the modulus and public exponent. It takes the form of a RSAPublicKey.

7.4.3 RSA Exponentiation

To do raw work with the RSA function call:

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

This loads the bignum from "in" as a big endian word in the format PKCS 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 that the output of his function is zero-padded as per PKCS #1 specifications. This allows this routine to interoprate with PKCS #1 padding functions properly.

7.4.4 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.

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

```
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 then RSA encrypt it and store the ciphertext in "out" of length "outlen". The "lparam" and "lparamlen" are the same parameters you would pass to pkcs_1_oaep_encode().

```
int rsa_decrypt_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, int *res,
                          rsa_key       *key);
```

This function will RSA decrypt "in" of length "inlen" then OAEP depad 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 isn't a valid OAEP packet then "res" is set to 0. Otherwise, it is set to 1.

7.4.5 RSA Hash Signatures

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 hash "in" of length "inlen". Next the PSS encoded message will be RSA "signed" and the output is stored in "out" of length "outlen".

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

This will RSA "verify" the signature in "sig" of length "siglen". Next the RSA decoded data is PSS decoded and the extracted hash is compared against the message hash "msghash" of length "msghashlen".

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

```
#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;
   }

   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 */
                            NULL, /* PRNG state */
                        prng_idx, /* prng idx */
                        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 */
}
```

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`

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 $\sim 2^{192}$ 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 Core Functions

Like the DH routines there is a key structure "ecc_key" used by the 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 20, 24, 28, 32, 48 and 65 bytes which correspond to key

sizes of 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 rest of the parameters work like they do in the "dh_make_key()" function. To free the ram allocated by a key call:

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

To import and export a key there are:

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

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

These two work exactly like there DH counterparts. Finally when you share your public key you can make a shared secret with:

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

Which works exactly like the DH counterpart, the "private_key" is your own key and "public_key" is the key the other user sent you. Note that this function stores both x and y co-ordinates of the shared elliptic point. You should hash the output to get a shared key in a more compact and useful form (most of the entropy is in x anyways). Both keys have to be the same size for this to work, to help there is a function to get the size in bytes of a key.

```
int ecc_get_size(ecc_key *key);
```

To test the ECC routines and to get the minimum and maximum key sizes there are these two functions:

```
int ecc_test(void);
void ecc_sizes(int *low, int *high);
```

Which both work like their DH counterparts.

9.3 ECC Packet

Similar to the RSA API there are two functions which encrypt and decrypt symmetric keys using the ECC public key algorithms.

```
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);

int ecc_decrypt_key(const unsigned char *in,  unsigned long  inlen,
                          unsigned char *out, unsigned long *outlen,
                          ecc_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 "ecc_encrypt_key()". The hash chosen must produce a message digest at least as large as the symmetric key you are trying to share.

There are also functions to sign and verify the hash of a message.

```
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);

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

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

9.4 ECC Keysizes

With ECC if you try and 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 a ECC-160 key in effect you have 160-bits of security (e.g. as if you signed with SHA-1).

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 a signature. For example, to have "80-bits of security" with ElGamal you need a group of order 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 where as the DSA signature would require only at least 40 bytes.

The API for the DSA is essentially the same as the other PK algorithms. Except in the case of DSA no encryption or decryption routines are provided.

10.2 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

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);
```

10.3 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).

10.4 Signatures

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.

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.

10.5 Import and 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.

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 DER Support

DER or "Distinguished Encoding Rules" is a subset of the ASN.1 encoding rules that is fully deterministic and ideal for cryptography. In particular ASN.1 specifies an INTEGER type for storing arbitrary sized integers. DER further limits the ASN.1 specifications to a deterministic encoding.

11.1.1 Storing INTEGER types

```
int der_encode_integer(mp_int *num, unsigned char *out, unsigned long *outlen);
```

This will store the integer in "num" to the output buffer "out" of length "outlen". It only stores non–negative numbers. It stores the number of octets used back in "outlen".

11.1.2 Reading INTEGER types

```
int der_decode_integer(const unsigned char *in, unsigned long *inlen, mp_int *num);
```

This will decode the DER encoded INTEGER in "in" of length "inlen" and store the resulting integer in "num". It will store the bytes read in "inlen" which is handy if you have to parse multiple data items out of a binary packet.

11.1.3 INTEGER length

```
int der_length_integer(mp_int *num, unsigned long *len);
```

This will determine the length of the DER encoding of the integer "num" and store it in "len".

11.1.4 Multiple INTEGER types

To simplify the DER encoding/decoding there are two functions two handle multple types at once.

```
int der_put_multi_integer(unsigned char *dst, unsigned long *outlen, mp_int *num, ...);
int der_get_multi_integer(const unsigned char *src, unsigned long *inlen,  mp_int *num, ...);
```

These will handle multiple encodings/decodings at once. They work like their single operand counterparts except they handle a **NULL** terminated list of operands.

```
#include <tomcrypt.h>
int main(void)
{
   mp_int        a, b, c, d;
   unsigned char buffer[1000];
   unsigned long len;
   int           err;

   /* init a,b,c,d with some values ... */

   /* ok we want to store them now... */
   len = sizeof(buffer);
   if ((err = der_put_multi_integer(buffer, &len,
                                    &a, &b, &c, &d, NULL)) != CRYPT_OK) {
      // error
   }
   printf("I stored %lu bytes in buf\n", len);

   /* ok say we want to get them back for fun */
   /* len set previously...otherwise set it to the size of the packet */
   if ((err = der_get_multi_integer(buffer, &len,
                                    &a, &b, &c, &d, NULL)) != CRYPT_OK) {
      // error
   }
   printf("I read %lu bytes from buf\n", len);
}
```

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 users 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 upto "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 invokation 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 upto "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) */
}
```

Chapter 12

Miscellaneous

12.1 Base64 Encoding and Decoding

The library provides functions to encode and decode a RFC1521 base64 coding scheme. This means that it can decode what it encodes but the format used does not comply to any known standard. The characters used in the mappings are:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

Those characters should are supported in virtually any 7-bit ASCII system 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 The Multiple Precision Integer Library (MPI)

The library comes with a copy of LibTomMath which is a multiple precision integer library written by the author of LibTomCrypt. LibTomMath is a trivial

to use ANSI C compatible large integer library which is free for all uses and is distributed freely.

At the heart of all the functions is the data type "mp_int" (defined in tommath.h). This data type is what will hold all large integers. In order to use an mp_int one must initialize it first, for example:

```
#include <tomcrypt.h> /* tomcrypt.h includes mpi.h automatically */
int main(void)
{
   mp_int bignum;

   /* initialize it */
   mp_init(&bignum);

   return 0;
}
```

If you are unfamiliar with the syntax of C the & symbol is used to pass the address of "bignum" to the function. All LibTomMath functions require the address of the parameters. To free the memory of a mp_int use (for example):

```
mp_clear(&bignum);
```

The functions also have the basic form of one of the following:

```
mp_XXX(mp_int *a);
mp_XXX(mp_int *a, mp_int *b, mp_int *c);
mp_XXX(mp_int *a, mp_int *b, mp_int *c, mp_int *d);
```

Where they perform some operation and store the result in the mp_int variable passed on the far right. For example, to compute $c = a + b \pmod{m}$ you would call:

```
mp_addmod(&a, &b, &m, &c);
```

12.2.1 Binary Forms of "mp_int" Variables

Often it is required to store a "mp_int" in binary form for transport (e.g. exporting a key, packet encryption, etc.). LibTomMath includes two functions to help when exporting numbers:

```
int mp_raw_size(mp_int *num);
mp_toraw(&num, buf);
```

The former function gives the size in bytes of the raw format and the latter function actually stores the raw data. All "mp_int" numbers are stored in big endian form (like PKCS demands) with the first byte being the sign of the number. The "rsa_exptmod()" function differs slightly since it will take the input in the form exactly as PKCS demands (without the leading sign byte). All other functions include the sign byte (since its much simpler just to include it). The sign byte must be zero for positive numbers and non-zero for negative numbers. For example, the sequence:

```
00 FF 30 04
```

Represents the integer $255 \cdot 256^2 + 48 \cdot 256^1 + 4 \cdot 256^0$ or 16,723,972.

To read a binary string back into a "mp_int" call:

```
mp_read_raw(mp_int *num, unsigned char *str, int len);
```

Where "num" is where to store it, "str" is the binary string (including the leading sign byte) and "len" is the length of the binary string.

12.2.2 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 candiate 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 singal 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 provided provides and easy to access source of entropy for any application on a *NIX 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 or such can cause algorithms to fail to work as desired.

13.3 Registering Your Algorithms

To avoid linking and other runtime 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.

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 todo.

13.4.2 Assymetric 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

The work factor for ECC keys is much higher since the best attack is still fully exponentional. Given a key of magnitude n it requires $\sqrt{n}$ work. The following table sumarizes the work required:

ECC Key Size (bits)	Work Factor (log_2)
160	80
192	96
224	112
256	128
384	192
521	260.5

Using the above tables the following suggestions for key sizes seems appropriate:

Security Goal	RSA/DH Key Size (bits)	ECC Key Size (bits)
Short term (less than a year)	1024	160
Short term (less than five years)	1536	192
Long Term (less than ten years)	2560	256

13.5 Thread Safety

The library is not 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. Its best to call them from your programs initializtion 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. This is fairly simple with symmetric ciphers and hashes. However, the keyring and PRNG support is something the threads will want to share. The simplest workaround is create semaphores or mutexes around calls to those functions.

Since C does not have standard semaphores this support is not native to Libtomcrypt. Even a C based semaphore is not entire possible as some compilers may ignore the "volatile" keyword or have multiple processors. Provide your host application is modular enough putting the locks in the right place should not bloat the code significantly and will solve all thread safety issues within the library.

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.

14.2 Building a Static Library

The library can be built as a static library which is generally the simplest and most portable method of building the library. With a CC or GCC equipped platform you can issue the following

```
make install_lib
```

Which will build the library and install it in /usr/lib (as well as the headers in /usr/include). The destination directory of the library and headers can be changed by editing "makefile". The variable LIBNAME controls where the library is to be installed and INCNAME controls where the headers are to be installed. A developer can then use the library by including "tomcrypt.h" in their program and linking against "libtomcrypt.a".

A static library can also be built with the Intel C Compiler (ICC) by issuing the following

```
make -f makefile.icc install
```

This will also build "libtomcrypt.a" except that it will use ICC. Additionally Microsoft's Visual C 6.00 can be used by issuing

```
nmake -f makefile.msvc
```

You will have to manually copy "tomcrypt.lib" and the headers to your MSVC lib/inc directories.

14.2.1 MPI Control

If you already have LibTomMath installed you can safely remove it from the build. By commenting the line in the appropriate makefile which starts with

```
MPIOBJECT=mpi
```

Simply place a # at the start and re-build the library. To properly link applications you will have to also link in LibTomMath. Removing MPI has the benefit of cutting down the library size as well potentially have access to the latest mpi.

14.3 Building a Shared Library

LibTomCrypt can also be built as a shared library (.so, .dll, etc...). With non-Windows platforms the assumption of the presence of gcc and "libtool" has been made. These are fairly common on Unix/Linux/BSD platforms. To build a .so shared library issue

```
make -f makefile.shared
```

This will use libtool and gcc to build a shared library "libtomcrypt.la" as well as a static library "libtomcrypt.a" and install them into /usr/lib (and the headers into /usr/include). To link your application you should use the libtool program in "–mode=link".

14.4 tomcrypt_cfg.h

The file "tomcrypt_cfg.h" is what lets you control various high level macros which control the behaviour of the library.

ARGTYPE

This lets you control how the _ARGCHK macro will behave. The macro is used to check pointers inside the functions against NULL. There are three 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 it will resolve to a empty macro and no error checking will be performed.

Endianess

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 and x86-64 running GCC as well as x86-32 running MSVC.

14.5 The Configure Script

There are also options you can specify from the configure script or "tomcrypt_custom.h".

14.5.1 X memory routines

At the top of tomcrypt_custom.h are four macros denoted as XMALLOC, XCALLOC, XREALLOC and XFREE which resolve to the name of the respective functions. 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. By default the library uses the standard C routines.

14.5.2 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.

14.5.3 NO_FILE

During the build if 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.

14.5.4 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.

14.5.5 LTC_TEST

When this has been defined the various self–test functions (for ciphers, hashes, prngs, etc) are included in the build. When this has been undefined the tests are removed and if called will return CRYPT_NOP.

14.5.6 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.

14.5.7 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 tradeoffs 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.

14.5.8 GCM_TABLES

When defined GCM will use a 64KB table (per GCM state) which will greatly lower up the per–packet latency. It also increases the initialization time.

14.5.9 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.

14.5.10 LTC_FAST

This mode (autodetected 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.

14.6 MPI Tweaks

14.6.1 RSA Only Tweak

If you plan on only using RSA with moduli in the range of 1024 to 2560 bits you can enable a series of tweaks to reduce the library size. Follow these steps

1. Undefine MDSA, MECC and MDH from tomcrypt_custom.h
2. Undefine LTM_ALL from tommath_superclass.h
3. Define SC_RSA_1 from tommath_superclass.h
4. Rebuild the library.

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 or PRNG and dependent API will not require updating. This has the nice benefit that I can add ciphers not have to re–write large 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 time is of the essence. This API also allows optimized routines to be dropped in–place of the existing portable routines. For instance, hand optimized assembler versions of AES could be provided and 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 work. For a given class of operation (e.g. cipher, hash, prng) the functions have identical prototypes which makes development simple. In most dependent routines all a developer has to do is register_XXX() the descriptor and they're 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
   */
   void (*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
   */
   void (*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);
   /** Determine a key size
       @param keysize    [in/out] The size of the key desired and 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
   */
   void (*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
*/
void (*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
*/
void (*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
*/
void (*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
*/
void (*accel_ctr_encrypt)(const unsigned char *pt,
                                unsigned char *ct, unsigned long blocks,
                                unsigned char *IV, int mode, 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 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
```

```
   */
   void (*accel_ccm_memory)(
       const unsigned char *key,    unsigned long keylen,
       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 length is the same)
       @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)
   */
   void (*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);

};
```

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 distingish 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".

This is where things get tricky. Currently there is no provision to allocate memory during initialization since there is no "cipher done" function. So you have to either use an existing member of the symmetric_key union or alias your own structure over top of it provided symmetric_key is not smaller.

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.

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 fallbacks 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 ($mode = 1$) or little ($mode = 0$) 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 CCM

This function is meant for accelerated CCM 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.

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.

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 DER identifier */
    unsigned char DER[64];
    /** Length of DER encoding */
    unsigned long DERlen;
    /** 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);
};
```

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 invokation.

15.3.5 DER Identifier

This is the DER identifier (including the SEQUENCE header). This is used solely for PKCS #1 style signatures.

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.

15.4 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 of the PRNG state
        @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.

Index