Tiny Crypto Library
"LibTomCrypt"
Version 0.66

Tom St Denis
Algonquin College

tomstdenis@yahoo.com
http://libtomcrypt.sunsite.dk

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

September 24, 2002

Contents

Chapter 1

Introduction

1.1 What is the Tiny Crypto Library?

Since I really could care less for a name you can call this library either "Tiny Crypto Library" or "libtomcrypt.a".

So what is "libtomcrypt.a"? It is a portable ANSI C cryptographic library that supports symmetric ciphers, one-way hashes, pseudo-random number generators and public key crypto (via RSA,DH or ECC/DH). It is designed to compile out of the box with the GNU C Compiler (GCC) version 2.95.3 (and higher) and with MSVC version 6.00 and higher.

The library is designed so new ciphers/hashes/prngs can be added and the existing API (and helper API functions) will be able to use the new designs automatically. There exist self-check functions for each cipher and hash to ensure that they compile and execute to the design specifications. The library also performs extensive parameter error checking and will give verbose error messages.

1.1.1 What the library IS for?

The library typically serves as a basis for other protocols and message formats. For example, it should be possible to take the RSA routines out of this library, apply the appropriate message padding and get PKCS compliant RSA routines. The goal is to provide these low level core functions.

The library also serves well as a toolkit for applications where they don't wish to be OpenPGP, PKCS, etc. compliant. Often one simply wants the ciphers, hashes, prngs and PK algorithms available, the actual message format is not important. This can makes the library versatile since it allows you to make up your own message format that suits the task at hand. To serve this end the library provide functions such as "rsa_encrypt()", "dh_encrypt()" and "ecc_encrypt()" which all perform packet PK encryption (there are related decryption routines available). The resulting output does not follow any known

standard format but should be compatible with any other application that uses this library.

Also by providing base core functions the library is well suited for students that need access to a crypto function to test a design or theory.

1.1.2 What the library IS NOT for?

The library is not designed to be in anyway an implementation of PKCS or OpenPGP standards. The library is not designed to be compliant with any known form of API or programming hierarchy. It is not a port of any other library and it is not platform specific (like the MS CSP). So if you're looking to drop in some buzzword compliant crypto this library is not for you. The library has been written from scratch to provide basic functions.

Why not?

You may be asking why I didn't choose to go all out and support standards like P1363, PKCS and the whole lot. The reason is quite simple too much money gets in the way. I just recently tried to access the P1363 draft documents and was denied (it requires a password). If people are supposed to support these standards they had better make them more accessible.

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 my core functions there is a central descriptor table 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.

1.3 License

All of the code except for MPI, the Zlib compression code and the AES/Serpent code has been written from scratch by Tom St Denis and as such is under the officially binding "Tom Doesn't Care About Licenses" or TDCAL license. This entitles the user to use this library for *whatever* they want without any form of repayment to the author whatsoever. Similarly MPI is public domain software copyrighted by Michael Fromberger and may be freely used. The Serpent and AES code are copyrighted by Brian Gladman and available for any purpose provided the copyright message and rights remains intact.

The "Zlib" compression code is copyrighted[1] "freeware" code. It may be freely distributed, modified and used provided the copyright and origins of the code are not removed or altered. Unlike the rest of the code, the authors express their disapproval of the term "public domain" and request that the code be treated as copyrighted freely distributable "freeware".

The net effect of all the varying licenses results in these simple rules:

1. The code is free to distribute, modify and use in commercial projects.

2. You may not remove or alter any copyright notice or header disclosing authors names and original code origins.

1.4 Building the library

To build the library on a GCC equipped platform simply type "make" at your command prompt. It will build the library file "libtomcrypt.a". If you are on a non-x86 platform comment out the appropriate line in "makefile". To build the test executable just take off the ".exe". I hope to have a makefile for *NIX platforms shortly.

Some users have noted that they must first convert the CR/LF nature of the files to the *NIX LF format before they will compile correctly. I will include an updated makefile in future releases to address this. For now just change the files manually before compiling. To install the library copy all of the ".h" files into your "#include" path and the single libtomcrypt.a file into your library path.

With MSVC you can build the library with "make -f makefile.vc".

1.4.1 Too big?

As I have been developing this library I've seen it grow from a small 40kb to a huge > 400kb it is now. Now you may not want to have all this big pile of code in your application.

To make it easy to save space I have written a series of #define switches at the top of "mycrypt_cfg.h". Just comment out the switch for the algorithm you do **NOT** want and rebuild the library. This lets you remove ciphers, hashes,

[1] Written by Jean-loup Gailly and Mark Adler.

prngs, modes of symmetric operation, RSA, Diffie-Hellman, ECC and the GF math routines at will. You can customize the library to suit your direct needs.

Since the DH and ECC key settings are fixed in the library (that makes the key generation routines more efficient and less error prone) they occupy code space (static data). You can also via "#define" switches tell the compiler which key settings you want to support.

As a side bonus this lets you remove RC5 and RC6 which is a good idea if you release your application in the US since RSA holds patents on both RC5 and RC6. Note that if you do not actually use any of the particular ciphers/prngs/hashes they will not get linked into the application. The build time switches are mostly for if you intend to include the library on your small memory device as say some sort of shared library or if your compiler cannot handle some of the code (e.g. 64-bit long long's). With all versions of GCC above 2.95.3 the library will build regardless of what platform provided the library doesn't exhaust the platforms memory space.

1.5 Thanks

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

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

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 either **CRYPT_OK** or **CRYPT_ERROR** always without exception. If there is an error the character pointer **crypt_error**[1] will be set to point to a string literal of the error. This provides verbose error reporting that is good for debugging code. For example:

```
void somefunc(void)
{
   /* call a cryptographic function */
   if (some_crypto_function(...) == CRYPT_ERROR) {
      printf("A crypto error occured, %s\n", crypt_error);
      /* 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, therefore, the library is thread safe. Another potential pitfall is if multiple errors occur simultaneously. In this case the functions will still return

[1] Note that it is lower case.

CRYPT_ERROR like they should but the "crypt_error" variable will not apply to the specific error. Therefore, it is a good idea to treat the "crypt_error" as suggestion in multi-threaded applications.

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

```
#include <mycrypt.h>
int main(void) {
    return 0;
}
```

The header file "mycrypt.h" also includes "stdio.h", "string.h", "stdlib.h", "time.h", "ctype.h", "mpi.h" (the bignum library routines) and "zlib.h" (the Zlib compression code).

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 the byte order of x.

There are 32-bit cyclic rotations as well:

ROL(x, y)	**unsigned long** x, **unsigned long** y	$x << y$
ROR(x, y)	**unsigned long** x, **unsigned long** y	$x >> y$

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

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

[2]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.

```
    unsigned char buffer[1024];
    unsigned long x;

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

    /* lets export the key, set x to the size of the output buffer */
    x = sizeof(buffer);
    if (rsa_export(buffer, &x, PK_PUBLIC, &key) == CRYPT_ERROR) {
       printf("Export error: %s\n", crypt_error);
       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 not store anything in either "buffer" or "x" and simply return **CRYPT_ERROR**. If the function suceeds it stores the length of the output back into "x" so that the calling application will know how many bytes used.

2.4 Functions that need a PRNG

Certain functions such as "rsa_make_key()" require a PRNG to function since they must make up a random key to be secure. These functions do not setup the PRNG themselves so it is the responsibility of the calling application to initialize the PRNG before calling them.

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 all in a plain vanilla ECB block mode. Its 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 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. 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. It returns **CRYPT_ERROR** if the parameters are invalid such as an incorrect key size or number of rounds. If the cipher is setup correctly it returns **CRYPT_OK**.

If the function returns successfully the variable "skey" will have a scheduled key stored in it. Its 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 for the key so when you are done with a key you can simply discard it (e.g. they can be on the stack). Generally it is a good idea to wipe the key from memory first using (for example):

```
zeromem(&skey, sizeof(skey));
```

Wiping the key prevents other functions that read the stack improperly from revealing parts of the key.

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

```
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. To test a particular cipher against test vectors[2] call:

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

This function will return **CRYPT_OK** if the cipher matches the test vectors from the designs. It returns **CRYPT_ERROR** if it fails to meet the test vectors. Finally 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. If the input keysize is invalid it will return **CRYPT_ERROR**. For example:

```
#include <mycrypt.h>
int main(void)
{
   int keysize;

   /* first register twofish */
   if (register_cipher(&twofish_desc) == -1) {
      printf("Error registering Twofish: %s\n", crypt_error);
      return -1;
   }

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

This should indicate a keysize of sixteen bytes is suggested. An example snippet that encodes a block with Blowfish in ECB mode is below.

[1] The size of which depends on which cipher you are using.
[2] As published in their design papers.

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

   /* first register Blowfish */
   if (register_cipher(&blowfish_desc) == -1) {
      printf("Error registering Blowfish: %s\n", crypt_error);
      return -1;
   }

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

   /* schedule the key */
   if (blowfish_setup(key, 8, 0, &skey) == CRYPT_ERROR) {
      printf("Setup error: %s\n", crypt_error);
      return -1;
   }

   /* encrypt the block */
   blowfish_ecb_encrypt(pt, ct, &skey);

   /* decrypt the block */
   blowfish_ecb_decrypt(ct, pt, &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. Its 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 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 long min_key_length, max_key_length,
                 block_length, default_rounds;
   int  (*setup)      (const unsigned char *key, int keylength,
                       int num_rounds, symmetric_key *skey);
   void (*ecb_encrypt)(const unsigned char *pt, unsigned char *ct,
                       symmetric_key *key);
   void (*ecb_decrypt)(const unsigned char *ct, unsigned char *pt,
                       symmetric_key *key);
   int  (*test)       (void);
   int  (*keysize)    (int *desired_keysize);
};
```

Where "name" is the lower case ASCII version of the name. The fields "min_key_length", "max_key_length" and "block_length" are all the number of bytes not bits. 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 (bytes)	Key Range (bytes)	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
Safer K64	safer_k64_desc	8	8	6 .. 13
Safer SK64	safer_sk64_desc	8	8	6 .. 13
Safer K128	safer_k128_desc	8	16	6 .. 13
Safer SK128	safer_sk128_desc	8	16	6 .. 13
Serpent	serpent_desc	16	16 .. 32	32
Rijndael (AES)	rijndael_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

3.3.1 Notes

For the 64-bit SAFER famliy of ciphers (e.g K64, SK64, K128, SK128) the ecb_encrypt() and ecb_decrypt() functions are the same. So if you want to use those functions directly just call safer_ecb_encrypt() or safer_ecb_decrypt() respectively.

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.

Note that "Twofish" has additional configuration options that take place at build time. These options are found in the file "mycrypt_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.

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 <mycrypt.h>
int main(void)
{
   unsigned char key[8];
   symmetric_key skey;

   /* you must register a cipher before you use it */
   if (register_cipher(&blowfish_desc)) == -1) {
      printf("Unable to register Blowfish cipher: %s\n", crypt_error);
      return -1;
   }

   /* generic call to function (assuming the key in key[] was already setup) */
   if (cipher_descriptor[find_cipher("blowfish")].setup(key, 8, 0, &skey) == CRYPT_ERROR) {
      printf("Error setting up Blowfish: %s\n", crypt_error);
      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 <mycrypt.h>
int main(void)
{
   /* register the cipher */
   if (register_cipher(&rijndael_desc) == -1) {
      printf("Error registering Rijndael\n");
      return -1;
   }

   /* use Rijndael */

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

   return 0;
}
```

This snippet is a small program that registers only Rijndael only. Note you must register your ciphers before using the PK code since all of the PK code (RSA, DH and ECC) rely heavily on the descriptor tables.

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. This mode 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 Implementation

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[3] 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 **CRYPT_ERROR**. To actually encrypt or decrypt the following routines are provided:

```
void XXX_encrypt(const unsigned char *pt, unsigned char *ct,
                 symmetric_XXX *XXX);
void XXX_decrypt(const unsigned char *ct, unsigned char *pt,
                 symmetric_XXX *XXX);

void YYY_encrypt(const unsigned char *pt, unsigned char *ct,
                 int len, symmetric_YYY *YYY);
void YYY_decrypt(const unsigned char *ct, unsigned char *pt,
                 int len, symmetric_YYY *YYY);
```

Where "XXX" is one of (ecb, cbc) and "YYY" is one of (ctr, ofb, cfb). In the CTR, OFB and CFB cases "len" is the size of the buffer (as number of chars) to encrypt or decrypt.

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. When you are done working with either mode you should wipe the memory (using "zeromem()") to help prevent the key from leaking. For example:

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

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

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

   /* somehow fill out key and IV */

   /* start up CTR mode */
   if (ctr_start(find_cipher("twofish"), IV, key, 16, 0, &ctr) == CRYPT_ERROR) {
      printf("ctr_start error: %s\n", crypt_error);
      return -1;
   }

   /* somehow fill buffer than encrypt it */
   ctr_encrypt(buffer, buffer, sizeof(buffer), &ctr);

   /* make use of ciphertext... */

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

   return 0;
}
```

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:

```
void XXX_process(hash_state *md, const unsigned char *in, unsigned long len);
```

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:

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

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

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

To test a hash function call:

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

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

```
#include <mycrypt.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 */
    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 *);
    void (*process)(hash_state *, const unsigned char *, unsigned long);
    void (*done)   (hash_state *, unsigned char *);
    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 <mycrypt.h>
int main(void)
{
   unsigned char buffer[100], hash[32];
   int idx, x;
   hash_state md;

   /* register hashes .... */
   if (register_hash(&md5_desc) == -1) {
      printf("Error registering MD5: %s\n", crypt_error);
      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;
}
```

There are two helper functions as well:

```
int hash_memory(int hash, const unsigned char *data,
                unsigned long len, unsigned char *dst,
                unsigned long *outlen);
```

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

Both functions return **CRYPT_OK** on success, otherwise they return **CRYPT_ERROR**. The "hash" parameter is the location in the descriptor table of the 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. To perform the above hash with md5 the following code could be used:

```
#include <mycrypt.h>
int main(void)
{
   int idx;
   unsigned long len;
   unsigned char out[16];

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

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

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

The following hashes are provided as of this release:

Name	Descriptor Name	Size of Message Digest (bytes)
SHA-512	sha512_desc	64
SHA-384	sha384_desc	48
SHA-256	sha256_desc	32
TIGER-192	tiger_desc	24
SHA-1	sha1_desc	20
MD5	md5_desc	16
MD4	md4_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);
```

4.2.1 Notice

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

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

4.3 Hash based Message Authenication Codes

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. The function returns **CRYPT_ERROR** on error or **CRYPT_OK** otherwise.

To send octets of a user message through the HMAC system you must use the provided function:

```
void hmac_process(hmac_state *hmac, const unsigned char *buf,
                  unsigned long len);
```

"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:

```
void hmac_done(hmac_state *hmac, unsigned char *hash);
```

"hmac" is the HMAC state you are working with. "hash" is the array of octets where the HMAC code should be stored. You must ensure that your destination array is the right size (or just make it of size MAXBLOCKSIZE to be sure). There are two utility functions provided to make using HMACs easier todo.

```
int hmac_memory(int hash, const unsigned char *key, unsigned long keylen,
                const unsigned char *data, unsigned long len,
                unsigned char *dst);
```

This will produce an HMAC code for the array of octets in "data" of length "len". 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"dst". The function returns **CRYPT_ERROR** on error or **CRYPT_OK** otherwise. 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 *dst);
```

"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. "keylen" is the length of the key. "dst" is the array of octets where the result should be stored. The function returns **CRYPT_ERROR** on error or **CRYPT_OK** otherwise.

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

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

Which simply returns **CRYPT_ERROR** on error or **CRYPT_OK** otherwise. Some example code for using the HMAC system is given below.

```
#include <mycrypt.h>
int main(void)
{
   int idx;
   hmac_state hmac;
   unsigned char key[16], dst[20];

   /* register SHA-1 */
   if (register_hash(&sha1_desc) == -1) {
      printf("Error registering SHA1: %s\n", crypt_error);
      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 (hmac_init(&hmac, idx, key, 16) == CRYPT_ERROR) {
      printf("Error setting up hmac: %s\n", crypt_error);
      return -1;
   }

   /* process a few octets */
   hmac_process(&hmac, "hello", 5);

   /* get result (presumably to use it somehow...) */
   hmac_done(&hmac, dst);

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

Chapter 5

Pseudo-Random Number Generators

5.1 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 but not seed it. Returns **CRYPTO_ERROR** if there is an error. 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 (see section 5.3). To add entropy to the PRNG call:

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

Which returns the number of bytes read from the PRNG.

5.1.1 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.

5.1.2 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 <mycrypt.h>
int main(void)
{
   prng_state prng;
   unsigned char buf[10];
   /* start it */
   if (yarrow_start(&prng) == CRYPT_ERROR) {
      printf("Start error: %s\n", crypt_error);
   }
   /* add entropy */
   if (yarrow_add_entropy("hello world", 11, &prng) == CRYPT_ERROR) {
      printf("Add_entropy error: %s\n", crypt_error);
   }
   /* ready and read */
   if (yarrow_ready(&prng) == CRYPT_ERROR) {
      printf("Ready error: %s\n", crypt_error);
   }
   printf("Read %lu bytes from yarrow\n", yarrow_read(buf, 10, &prng));
   return 0;
}
```

5.2 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 (*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 *);
};
```

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);
```

Currently only Yarrow (yarrow_desc) and the secure RNG (sprng_desc) are provided.

5.3 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[1]. 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));
```

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

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()". This returns **CRYPT_ERROR** on error, otherwise it returns **CRYPT_OK**. Note that you don't have to double the input to this function, that is, if you want a 128-bit state PRNG just pass 128 in the "bits" parameter. 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.

```
#include <mycrypt.h>
int main(void)
{
   prng_state prng;

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

   /* setup the PRNG */
   if (rng_make_prng(128, find_prng("yarrow"), &prng, NULL) == CRYPT_ERROR) {
      printf("Error setting up PRNG, %s\n", crypt_error);
      return -1;
   }
   return 0;
}
```

5.4 PRNGs Included

The library currently includes two different PRNG systems, the first is "yarrow" which is a derivative of the "yarrow" PRNG[2]. The second is "sprng" which uses the "rng_get_bytes()" function as a source.

You never need to start, seed or ready the "sprng" PRNG, you can use it right away. If you call a function that needs a PRNG you can pass "prng_find('sprng')" as the index and NULL as the prng state. For example, below is a snippet that makes an ECC key with this PRNG:

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

[2]It uses a cipher in CTR mode like Yarrow but has a different reseed mechanism that is simpler.

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

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

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

Chapter 6

RSA Routines

6.1 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 re-

sult 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.

6.2 Core Functions

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. This routine returns **CRYPTO_ERROR** if it fails to make a RSA key. All keys must be at least 128 bytes and no more than 512 bytes in size (thats 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 three types of RSA keys. The types are **PK_PRIVATE_OPTIMIZED**, **PK_PRIVATE** and **PK_PUBLIC**. The first two are private keys where the "optimized" type uses the Chinese Remainder Theorem to speed up decryption/signatures. By default all new keys are of the "optimized" type. The non-optimized private type is provided for backwards compatibility as well as to save space since the optimized key requires about four times as much memory.

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, rsa_key *key);
```

This loads the bignum from "in" as a big endian word, 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). This function returns **CRYPT_ERROR** on error.

6.3 Packet Routines

The remaining RSA functions are non-standard but should (to the best of my knowledge) be secure if used correctly. To encrypt a buffer of memory in a hybrid fashion call:

```
int rsa_encrypt(const unsigned char *in, unsigned long len,
                unsigned char *out, unsigned long *outlen,
                prng_state *prng, int wprng, int cipher,
                rsa_key *key);
```

This will encrypt the message with the cipher specified by "cipher" under a random key made by a PRNG specified by "wprng" and RSA encrypt the symmetric key with "key". This stores all the relevant information in "out" and sets the length in "outlen". You must ensure that "outlen" is set to the buffer size before calling this. This returns **CRYPT_ERROR** on error.

The rsa_encrypt() function will use up to a 256-bit symmetric key (limited by the max key length of the cipher being used). To decrypt packets made by this routine call:

```
int rsa_decrypt(const unsigned char *in, unsigned long len,
                unsigned char *out, unsigned long *outlen,
                rsa_key *key);
```

Which works akin to rsa_encrypt(). "in" is the ciphertext and "out" is where the plaintext will be stored. Similarly to sign/verify there are:

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

int rsa_verify(const unsigned char *sig,
               const unsigned char *msg,
               unsigned long inlen, int *stat,
               rsa_key *key);
```

The verify function sets "stat" to 1 if it passes or to 0 if it fails. The "sig" parameter is the output of the rsa_sign() function and "msg" is the original msg that was signed. An important fact to note is that with the padding scheme used in "rsa_sign()" you cannot use the SHA-384 or SHA-512 hash function with 1024 bit RSA keys. This is because the padding makes the values too large to fit in the space allowed. You can use SHA-384 with 1160 and above bit RSA keys. You can use SHA-512 with 1544 and above bit RSA keys.

There are times where you may want to encrypt a message to multiple recipients via RSA public keys. The simplest way to accomplish this is to make up your own symmetric key and then RSA encrypt the symmetric key using all of the recipients public keys. To facilitate this task two functions[1] are available:

```
int rsa_encrypt_key(const unsigned char *inkey, unsigned long inlen,
                          unsigned char *outkey, unsigned long *outlen,
                          prng_state *prng, int wprng, rsa_key *key);
```

[1] Donated by Clay Culver.

```
int rsa_decrypt_key(const unsigned char *in,  unsigned long len,
                          unsigned char *outkey, unsigned long *keylen,
                          rsa_key *key);
```

The "rsa_encrypt_key()" function accepts a symmetric key (limited to 32 bytes) as input in "inkey". "inlen" is the size of the input key in bytes. The function will then "rsa_pad()" the key and encrypt it using the RSA algorithm. It will store the result in "outkey" along with the length in "outlen". The "rsa_decrypt_key()" function performs the opposite. The "in" variable is where the RSA packet goes and it will store the original symmetric key in the "outkey" variable along with its length in "keylen". Both functions return **CRYPT_ERROR** on error, otherwise they return **CRYPT_OK**.

To import/export RSA keys as a memory buffer (e.g. to store them to disk) call:

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

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

The "type" parameter is **PK_PUBLIC**, **PK_PRIVATE** or **PK_PRIVATE_OPTIMIZED** to export either a public or private key. The latter type will export a key with the optimized parameters. Both functions return **CRYPT_ERROR** on error, otherwise they return **CRYPT_OK**. To free the memory used by an RSA key call:

```
void rsa_free(rsa_key *key);
```

Note that if the key fails to "rsa_import()" you do not have to free the memory allocated for it.

6.4 Remarks

It is important that you match your RSA key size with the function you are performing. The internal padding for both signatures and encryption triple the size of the plaintext you send to rsa_exptmod(). This means to encrypt or sign a message of N bytes with rsa_exptmod() you must have a modulus of 1+3N bytes. Note that this doesn't affect the length of the plaintext you pass into functions like rsa_encrypt(). This restriction applies only to data that is passed through RSA directly.

The following table gives the size requirements for various hashes.

Name	Size of Message Digest (bytes)	RSA Key Size (bits)
SHA-512	64	1544
SHA-384	48	1160
SHA-256	32	776
TIGER-192	24	584
SHA-1	20	488
MD5	16	392
MD4	16	392

The symmetric ciphers will use at a maximum a 256-bit key which means at the least a 776-bit RSA key is required to use all of the symmetric ciphers with the RSA routines. It is suggested that you make keys that are at a minimum 1024 bits in length. If you want to use any of the large size message digests (SHA-512 or SHA-384) you will have to use a larger key.

Chapter 7

Diffie-Hellman Key Exchange

7.1 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.

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

The "keysize" is the size of the modulus you want in bytes. Currently support sizes are 64 to 512 bytes which correspond to key sizes of 512 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 512 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 routine returns **CRYPT_ERROR** if there was an error, otherwise it returns **CRYPT_OK**

As for Diffie-Hellman key sizes its recommended that you use at least a 768-bit key. Since a 512-bit key has never been broken (its much harder than 512-bit RSA to attack) a 512-bit key setting is supported. You can use it if you want I just suggest you don't.

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, dh_key *key);
```

These two functions work just like the "rsa_export()" and "rsa_import()" functions except these work with Diffie-Hellman keys. They both return **CRYPT_ERROR** on error, otherwise they return **CRYPT_OK**. 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". It returns **CRYPT_ERROR** on error, otherwise it returns **CRYPT_OK**. 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.

7.2.1 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;
   dh_key mykey, theirkey;

   /* make up our private key */
   if (dh_make_key(128, prng, wprng, &mykey) == CRYPT_ERROR)
      return CRYPT_ERROR;

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

   if (mode == 0) {
      /* mode 0 so we send first */
      if (send(sock, buf, x, 0) != x) {
         crypt_error = "Error writing socket in establish_secure_socket()."
         res = CRYPT_ERROR;
         goto done2;
      }

      /* get their key */
      if (recv(sock, buf2, sizeof(buf2), 0) <= 0) {
         crypt_error = "Error reading socket in establish_secure_socket()."
         res = CRYPT_ERROR;
         goto done2;
      }
   } else {
      /* mode >0 so we send second */
      if (recv(sock, buf2, sizeof(buf2), 0) <= 0) {
         crypt_error = "Error reading socket in establish_secure_socket()."
         res = CRYPT_ERROR;
         goto done2;
      }

      if (send(sock, buf, x, 0) != x) {
         crypt_error = "Error writing socket in establish_secure_socket()."
         res = CRYPT_ERROR;
         goto done2;
      }
   }

   if (dh_import(buf2, &theirkey) == CRYPT_ERROR) {
      res = CRYPT_ERROR;
```

```
      goto done2;
   }

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

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

7.2.2 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()".

7.3 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_ERROR** if the bases and primes in the library are malformed, otherwise it returns **CRYPT_OK**. 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.

7.4 DH Packet

There are routines to perform the work similar to that of "rsa_encrypt()" and "rsa_decrypt()" for DH keys as well. The encrypt routine will make up a random key, attach the public key to the message and used the shared secret to encrypt the message with a cipher you choose (and hash the shared secret into a symmetric key with a hash you choose). The encrypt function is a bit long to call but its worth it.

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

Where "in" is the plaintext and "out" is where the ciphertext will go. Make sure you set the "outlen" value before calling. The "key" is the public DH key of the user you want to encrypt to not your private key. It will randomly make up a Diffie-Hellman key, export the public copy, hash the shared key with the hash you specify and use the message digest in a cipher you specify to encrypt the message. On error it returns **CRYPTO_ERROR**, otherwise it returns **CRYPTO_OK**. To decrypt one of these packets call:

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

Where "in" is the ciphertext and len is the length of the ciphertext. "out" is where the plaintext should be stored and "outlen" is the length of the output (you must first set it to the size of your buffer).

To facilate encrypting to multiple parties the follow two functions are provided:

```
int dh_encrypt_key(const unsigned char *inkey, unsigned long keylen,
                         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 len,
                         unsigned char *outkey, unsigned long *keylen,
                         dh_key *key);
```

Where "inkey" 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.

To sign with a Diffie-Hellman key call:

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

Where "in" is the message to size of length "inlen" bytes. "out" is where the signature is placed and "outlen" is the length of the signature (you must first set it to the size of your buffer). The function returns **CRYPT_ERROR** on error, otherwise it returns **CRYPT_OK**. To verify call:

```
int dh_verify(const unsigned char *sig,
              const unsigned char *msg,
              unsigned long inlen, int *stat, dh_key *key);
```

Where "sig" is the output of "dh_sign()" and "msg" is the message of length "inlen". It stores a zero in "stat" if the signature is invalid otherwise it puts a one in there. The function returns **CRYPT_ERROR** on error, otherwise it returns **CRYPT_OK**.

Chapter 8

Elliptic Curve Cryptography

8.1 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:

$$y^2 = x^3 - 3x + b \pmod{p} \tag{8.1}$$

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.

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

The "keysize" is the size of the modulus in bytes desired. Currently directly supported values are 24, 32, 48 and 65 which correspond to key sizes of 192, 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. It returns **CRYPT_ERROR** on error, otherwise it returns **CRYPT_OK**. 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, 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.

8.3 ECC Packet

There are routines to perform the work similar to that of "rsa_encrypt()" and "rsa_decrypt()" for ECC keys as well. The encrypt routine will make up a random key, attach the public key to the message and used the shared secret to encrypt the message with a cipher you choose (and hash the shared secret into a symmetric key with a hash you choose). The encrypt function is a bit long to call but its worth it.

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

Where "in" is the plaintext and "out" is where the ciphertext will go. Make sure you set the "outlen" value before calling. The "key" is the public ECC key of the user you want to encrypt too. On error it returns **CRYPTO_ERROR**, otherwise it returns **CRYPTO_OK**. To decrypt one of these packets call:

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

Which returns **CRYPT_ERROR** on error. Similar to the DH code there are two functions to facilate multi-party code. They work exactly like the DH code and are given as:

```
int ecc_encrypt_key(const unsigned char *inkey, unsigned long keylen,
                          unsigned char *out,  unsigned long *len,
                          prng_state *prng, int wprng, int hash,
                          ecc_key *key);

int ecc_decrypt_key(const unsigned char *in,  unsigned long len,
                          unsigned char *outkey, unsigned long *keylen,
                          ecc_key *key);
```

You can sign messages with the ECC routines as well, to sign a message call:

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

Where "in" is the message to sign and "out" is where the signature will go. "hash" is the index into the descriptor table of which hash function you want to use (e.g. use "find_hash()"). You must set "outlen" to the size of the output buffer before calling. This function returns **CRYPTO_ERROR** on error, otherwise it returns **CRYPT_OK**. To verify a signature call:

```
int ecc_verify(const unsigned char *sig, const unsigned char *msg,
               unsigned long inlen, int *stat, ecc_key *key);
```

Where "sig" is the signature from "ecc_sign()" and "msg" is the input message. It sets "stat" to 0 if the signature is invalid and it sets "stat" to 1 if its valid. This function returns **CRYPTO_ERROR** on error, otherwise it returns **CRYPT_OK**.

Chapter 9

$GF(2^w)$ Math Routines

The library provides a set of polynomial-basis $GF(2^w)$ routines to help facilitate algorithms such as ECC over such fields. Note that the current implementation of ECC in the library is strictly over the integers only. The routines are simple enough to use for other purposes outside of ECC.

At the heart of all of the GF routines is the data type "gf_int'. It is simply a type definition for an array of L 32-bit words. You can configure the maximum size L of the "gf_int" type by opening the file "mycrypt.h" and changing "LSIZE". Note that if you set it to n then you can only multiply upto two $\frac{n}{2}$ bit polynomials without an overflow. The type "gf_intp" is associated with a pointer to an "unsigned long" as required in the algorithms.

There are no initialization routines for "gf_int" variables and you can simply use them after declaration. There are five low level functions:

```
void gf_copy(gf_intp a, gf_intp b);
void gf_zero(gf_intp a);
int gf_iszero(gf_intp a);
int gf_isone(gf_intp a);
int gf_deg(gf_intp a);
```

There are all fairly self-explanatory. "gf_copy(a, b)" copies the contents of "a" into "b". "gf_zero()" simply zeroes the entire polynomial. "gf_iszero()" tests to see if the polynomial is all zero and "gf_isone()" tests to see if the polynomial is equal to the multiplicative identity. "gf_deg()" returns the degree of the polynomial or -1 if its a zero polynomial.

There are five core math routines as well:

```
void gf_shl(gf_intp a, gf_intp b);
void gf_shr(gf_intp a, gf_intp b);
void gf_add(gf_intp a, gf_intp b, gf_intp c);
void gf_mul(gf_intp a, gf_intp b, gf_intp c);
void gf_div(gf_intp a, gf_intp b, gf_intp q, gf_intp r);
```

Which are all fairly obvious. "gf_shl(a,b)" multiplies the polynomial "a" by x and stores it in "b". "gf_shl(a,b)" divides the polynomial "a" by x and stores it in "b". "gf_add(a,b,c)" adds the polynomial "a" to "b" and stores the sum in "c". Similarly for "gf_mul(a,b,c)". The "gf_div(a,b,q,r)" function divides "a" by "b" and stores the quotient in "q" and the remainder in "r".

There are six number theoretic functions as well:

```
void gf_mod(gf_intp a, gf_intp m, gf_intp b);
void gf_mulmod(gf_intp a, gf_intp b, gf_intp m, gf_intp c);
void gf_invmod(gf_intp A, gf_intp M, gf_intp B);
void gf_sqrt(gf_intp a, gf_intp m, gf_intp b);
void gf_gcd(gf_intp A, gf_intp B, gf_intp c);
int gf_is_prime(gf_intp a);
```

Which all work similarly except for "gf_mulmod(a,b,m,c)" which computes $c = ab \pmod{m}$. The "gf_is_prime()" function returns one if the polynomial is primitive, otherwise it returns zero.

Finally to read/store a "gf_int" in a binary string use:

```
int gf_size(gf_intp a);
void gf_toraw(gf_intp a, unsigned char *dst);
void gf_readraw(gf_intp a, unsigned char *str, int len);
```

Where "gf_size()" returns the size in bytes required for the data. "gf_toraw(a,b)" stores the polynomial in "b" in binary format (endian neutral). "gf_readraw(a,b,c)" reads the binary string in "b" back. Note that the length you pass it must be the same as returned by "gf_size()" or it will not load correctly.

Chapter 10

Miscellaneous

10.1 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);
```

10.2 Data Compression

The library provides the entire suite of Zlib compression functions. The interested user is encouraged to goto the actual Zlib website:

`http://www.gzip.org/zlib/`

The library also provides two wrapper functions for memory buffer compression which have the same calling convention as the other cryptographic functions. They are:

```
int pack_buffer(const unsigned char *in,  unsigned long inlen,
                      unsigned char *out, unsigned long *outlen)

int unpack_buffer(const unsigned char *in,  unsigned long inlen,
                        unsigned char *out, unsigned long *outlen)
```

The "pack" routine will compress the data from "in" and store the data stream in "out". The "unpack" routine will decompress the data from "in" and store the original data in "out". In both cases you must set the size of your output buffer in "outlen" before calling the function. They return **CRYPT_ERROR** on error, otherwise they return **CRYPT_OK**.

10.3 The Multiple Precision Integer Library (MPI)

The library comes with a copy of Michael Fromberger's[1] multiple precision integer library MPI. MPI is a trivial to use ANSI C compatible large integer library. It is free for all uses and is distributed freely.

At the heart of all MPI functions is the data type "mp_int" (defined in mpi.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 <mycrypt.h> /* mycrypt.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 MPI functions require the address of the parameters. To free the memory of a mp_int use (for example):

[1] Michael J. Fromberger, sting@linguist.Thayer.dartmouth.edu

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

All MPI functions 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);
```

The simplest way to get a complete listing of all MPI functions is to open "mpi.h" and look. They're all fairly well documented.

10.3.1 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.). MPI 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.

10.3.2 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 6542 primes. Second it will perform sixteen 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 $\left(\frac{1}{4}\right)^{16} = 2^{-32}$. Even if 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 MPI library[2] 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 will return **CRYPT_ERROR** on error and **CRYPT_OK** otherwise. 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 ($2 \leq len \leq 1024$). 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. It returns **CRYPT_ERROR** on error, otherwise it returns **CRYPT_OK** and "N" will contain a probable prime.

[2] As written by Michael Fromberger

Chapter 11

Programming Guidelines

11.1 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.

11.2 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 integer which is either **CRYPT_OK** or **CRYPT_ERROR**. You should detect errors whenever possible.

11.3 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.

11.4 Key Sizes

11.4.1 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.

11.4.2 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

$$e^{1.923 \cdot ln(n)^{\frac{1}{3}} \cdot ln(ln(n))^{\frac{2}{3}}} \tag{11.1}$$

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)
192	96
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)	768	192
Short term (less than five years)	1024	192
Long Term (less than ten years)	2560	256