A Tiny Crypto Library, LibTomCrypt Version 0.93

Tom St Denis
Algonquin College

tomstdenis@iahu.ca
http://libtomcrypt.org

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

January 25, 2004

 This book has been formatted for B5 [176x250] paper using the LaTeX *book* macro package.

Open Source. Open Academia. Open Minds.

Tom St Denis,
Ontario, Canada

Contents

Chapter 1

Introduction

1.1 What is the LibTomCrypt?

LibTomCrypt is a portable ANSI C cryptographic library that supports symmetric ciphers, one-way hashes, pseudo-random number generators, public key cryptography (via RSA,DH or ECC/DH) and a plethora of support routines. 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 in win32.

The library has been successfully tested on quite a few other platforms ranging from the ARM7TDMI in a Gameboy Advanced to various PowerPC processors and even the MIPS processor in the PlayStation 2. Suffice it to say the code is portable.

The library is designed so new ciphers/hashes/PRNGs can be added at runtime 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 published design specifications. The library also performs extensive parameter error checking and will give verbose error messages when possible.

Essentially the library saves the time of having to implement the ciphers, hashes, prngs yourself. Typically implementing useful cryptography is an error prone business which means anything that can save considerable time and effort is a good thing.

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. Similarly SSL protocols could be formed on top of the low-level symmetric cipher functions. The goal of this package is to provide these low level core functions in a robust and easy to use fashion.

The library also serves well as a toolkit for applications where they don't need to be OpenPGP, PKCS, etc. compliant. Included are fully operational public key routines for encryption, decryption, signature generation and verification. These routines are fully portable but are not conformant to any known set of standards. They are all based on established number theory and cryptography.

1.1.2 What the library IS NOT for?

The library is not designed to be in anyway an implementation of the SSL, PKCS, P1363 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 library this is not for you. The library has been written from scratch to provide basic functions as well as non-standard higher level functions.

This is not to say that the library is a "homebrew" project. All of the symmetric ciphers and one-way hash functions conform to published test vectors. The public key functions are derived from publicly available material and the majority of the code has been reviewed by a growing community of developers.

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. When I tried to access the P1363 draft documents and was denied (it requires a password) I realized that they're just a business anyways. See what happens is a company will sit down and invent a "standard". Then they try to sell it to as many people as they can. All of a sudden this "standard" is everywhere. Then the standard is updated every so often to keep people dependent. Then you become RSA. 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 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.

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 todo 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*) is 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
2. safer.c

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

"rc2.c" is based on publicly available code that is not attributed to a person from the given source. "safer.c" was written by Richard De Moliner (demoliner@isi.ee.ethz.ch) and is public domain.

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 the makefile script. 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 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".

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 "nmake -f makefile.msvc". This will produce a "tomcrypt.lib" file which is the core library. Copy the header files into your MSVC include path and the library in the lib path (typically under where VC98 is installed).

1.6 Building against the library

In the recent versions the build steps have changed. The build options are now stored in "mycrypt_custom.h" and no longer in the makefile. If you change a build option in that file you must re-build the library from clean to ensure the build is intact. The perl script "config.pl" will help setup the custom header and a custom makefile if you want one (the provided "makefile" will work with custom configs).

1.7 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
11. Andrew Tyler
12. Sky Schulz
13. Christopher Imes

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 errno);
```

An example of handling an error is:

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

   /* call a cryptographic function */
   if ((errno = some_crypto_function(...)) != CRYPT_OK) {
      printf("A crypto error occured, %s\n", error_to_string(errno));
```

```
      /* 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 "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" and "mpi.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 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[1] where the output will be stored. For example:

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

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

    /* lets export the key, set x to the size of the output buffer */
    x = sizeof(buffer);
    if ((errno = rsa_export(buffer, &x, PK_PUBLIC, &key)) != CRYPT_OK) {
       printf("Export error: %s\n", error_to_string(errno));
       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 an error code. 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. These functions do not setup the PRNG themselves so it is the responsibility of the calling function

[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.

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 of the ciphers are written as ECB interfaces since it allows the rest of the API to grow in a modular fashion.

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.

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 off the heap so when you are done with a key you can simply discard it (e.g. they can be on the stack).

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. For the encrypt function "pt"[2] is the input and "ct" is the output. For the decryption function its the opposite. To test a particular cipher against test vectors[3] call:

```
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. 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. This function return **CRYPT_OK** if the key size specified is acceptable. For example:

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

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

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

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

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

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

   /* schedule the key */
   if ((errno = blowfish_setup(key, 8, 0, &skey)) != CRYPT_OK) {
      printf("Setup error: %s\n", error_to_string(errno));
      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	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
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
AES	aes_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

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.

TWOFISH_SMALL	TWOFISH_TABLES	Speed and Memory (per key)
undefined	undefined	Very fast, 4.2KB of ram.
undefined	defined	As above, faster keysetup, larger code (1KB more).
defined	undefined	Very slow, 0.2KB of ram.
defined	defined	Somewhat 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 <mycrypt.h>
int main(void)
{
   unsigned char key[8];
   symmetric_key skey;
   int errno;

   /* 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 ((errno = cipher_descriptor[find_cipher("blowfish")].setup(key, 8, 0, &skey)) !=
      printf("Error setting up Blowfish: %s\n", error_to_string(errno));
      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)
{
   int errno;

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

   /* use Rijndael */

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

   return 0;
}
```

This snippet is a small program that registers only Rijndael only. Note you must register 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. 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 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[4] 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. To actually encrypt or decrypt the following routines are provided:

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

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

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

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:

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

   /* 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 ((errno = ctr_start(find_cipher("twofish"), IV, key, 16, 0, &ctr)) != CRYPT_OK) {
      printf("ctr_start error: %s\n", error_to_string(errno));
      return -1;
   }

   /* somehow fill buffer than encrypt it */
   if ((errno = ctr_encrypt(buffer, buffer, sizeof(buffer), &ctr)) != CRYPT_OK) {
      printf("ctr_encrypt error: %s\n", error_to_string(errno));
      return -1;
   }

   /* make use of ciphertext... */

   /* 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[5] in a manner similar to the way it was intended to be used.

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. 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 an invalid 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".

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**. It will still initialize the EAX "H" value to the correct value.

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);
```

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

```
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.

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.

Also note that 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).

```
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.

3.5.2 OCB Mode

LibTomCrypt provides support for a mode called OCB[6] in a mode somewhat similar to as it was meant to be used.

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 for AES).

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

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

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

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);
```

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

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).

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.

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 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:

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

```
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:

```
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 <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 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 *);
    int  (*process)(hash_state *, const unsigned char *, unsigned long);
    int  (*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[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 *data,
                unsigned long len, unsigned char *dst,
                unsigned long *outlen);

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

int hash_filehandle(int hash, FILE *in,
                    unsigned char *dst, 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 <mycrypt.h>
```

```
int main(void)
{
   int idx, errno;
   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 ((errno = hash_memory(idx, "hello world", 11, out, &len)) != CRYPT_OK) {
      printf("Error hashing data: %s\n", error_to_string(errno));
      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
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);
```

4.2.1 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. 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 *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:

```
int hmac_done(hmac_state *hmac, unsigned char *hashOut,
              unsigned long *outlen);
```

"hmac" is the HMAC state you are working with. "hashOut" 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 *data, unsigned long len,
                unsigned char *dst, unsigned long *dstlen);
```

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" and the length in "dstlen". The value of "dstlen" 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 *dst, unsigned long *dstlen);
```

"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". "dst" 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 <mycrypt.h>
int main(void)
{
   int idx, errno;
   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 ((errno = hmac_init(&hmac, idx, key, 16)) != CRYPT_OK) {
      printf("Error setting up hmac: %s\n", error_to_string(errno));
      return -1;
   }

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

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

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

4.4 OMAC Support

OMAC[2], 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[3] 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 *buf, unsigned long len);
```

This will send "len" bytes from "buf" through the active OMAC state "state". Returns **CRYPT_OK** if the function succeeds. When you are done with the message you can call

```
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

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

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

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

This will compute the OMAC 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 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 <mycrypt.h>
int main(void)
{
   int idx, errno;
   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 ((errno = omac_init(&omac, idx, key, 16)) != CRYPT_OK) {
      printf("Error setting up omac: %s\n", error_to_string(errno));
      return -1;
   }

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

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

   /* 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 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 (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];
   int errno;

   /* start it */
   if ((errno = yarrow_start(&prng)) != CRYPT_OK) {
      printf("Start error: %s\n", error_to_string(errno));
   }
   /* add entropy */
   if ((errno = yarrow_add_entropy("hello world", 11, &prng)) != CRYPT_OK) {
      printf("Add_entropy error: %s\n", error_to_string(errno));
   }
   /* ready and read */
   if ((errno = yarrow_ready(&prng)) != CRYPT_OK) {
```

```
      printf("Ready error: %s\n", error_to_string(errno));
   }
   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);
```

PRNGs Provided

Currently Yarrow (yarrow_desc), RC4 (rc4_desc) and the secure RNG (sprng_desc) are provided as PRNGs within the library.

RC4 is provided with a PRNG interface because it is a stream cipher and not well suited for the symmetric block cipher interface. You provide the key for RC4 via the rc4_add_entropy() function. By calling rc4_ready() the key will be used to setup the RC4 state for encryption or decryption. The rc4_read() function has been modified from RC4 since it will XOR the output of the RC4 keystream generator against the input buffer you provide. The following snippet will demonstrate how to encrypt a buffer with RC4:

```
#include <mycrypt.h>
int main(void)
{
   prng_state prng;
   unsigned char buf[32];
   int errno;

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

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

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

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));
```

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.

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

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

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

```
   }

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

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

5.3.1 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 <mycrypt.h>
int main(void)
{
   ecc_key mykey;
   int errno;

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

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

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. 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 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 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 this function does not perform padding on the input (as per PKCS). So if you send in "0000001" you will get "01" back (when you do the opposite operation). Make sure you pad properly which usually involves setting the msb to a non-zero value.

6.3 Packet Routines

To encrypt or decrypt a symmetric key using RSA the following functions are provided. The idea is that you make up a random symmetric key and use that to encode your message. By RSA encrypting the symmetric key you can send it to a recipient who can RSA decrypt it and symmetrically decrypt the message.

```
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);
```

This function is used to RSA encrypt a symmetric to share with another user. The symmetric key and its length are passed as "inkey" and "inlen" respectively. The symmetric key is limited to a range of 8 to 32 bytes (*64 to 256 bits*). The RSA encrypted packet is stored in "outkey" and will be of length "outlen" bytes. The value of "outlen" must be originally set to the size of the output buffer.

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

This function will decrypt an RSA packet to retrieve the original symmetric key encrypted with rsa_encrypt_key(). Similarly to sign or verify a hash of a message the following two messages are provided. The idea is to hash your message then use these functions to RSA sign the hash.

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

int rsa_verify_hash(const unsigned char *sig, unsigned long siglen,
                    const unsigned char *hash, int *stat, rsa_key *key);
```

For "rsa_sign_hash" the input is intended to be the hash of a message the user wants to sign. The output is the RSA signed packet which "rsa_verify_hash" can verify. For the verification function "sig" is the RSA signature and "hash" is the hash of the message. The integer "stat" is set to non-zero if the signature is valid or zero otherwise.

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, unsigned long inlen, rsa_key *key);
```

The "type" parameter is **PK_PUBLIC**, **PK_PRIVATE** or **PK_PRIVATE_OPTIMIZI** to export either a public or private key. The latter type will export a key with the optimized parameters. 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. This means to encrypt or sign a message of N bytes 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 the internal RSA routines directly 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. 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. Or to be simple just make 2048-bit or larger keys. None of the hashes will have problems with such key sizes.

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(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.

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, errno, inlen;
   dh_key mykey, theirkey;

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

   /* export our key as public */
   x = sizeof(buf);
   if ((errno = dh_export(buf, &x, PK_PUBLIC, &mykey)) != CRYPT_OK) {
      res = errno;
      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 ((errno = dh_import(buf2, inlen, &theirkey)) != CRYPT_OK) {
      res = errno;
      goto done2;
   }

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

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

7.4 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 *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 inlen,
                         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.

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 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(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.

8.3 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 *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 inlen,
                          unsigned char *outkey, unsigned long *keylen,
                          ecc_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 "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.

8.4 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 9

Digital Signature Algorithm

9.1 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.

9.2 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);
```

9.3 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).

9.4 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.

9.5 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 10

Public Keyrings

10.1 Introduction

In order to simplify the usage of the public key algorithms a set of keyring routines have been developed. They let the developer manage asymmetric keys by providing load, save, export, import routines as well as encrypt, decrypt, sign, verify routines in a unified API. That is all three types of PK systems can be used within the same keyring with the same API.

To define types of keys there are four enumerations used globaly:

```
enum {
   NON_KEY=0,
   RSA_KEY,
   DH_KEY,
   ECC_KEY
};
```

To make use of the system the developer has to know how link-lists work. The main structure that the keyring routines use is the "pk_key" defined as:

```
typedef struct Pk_key {
    int     key_type,             /* PUBLIC, PRIVATE, PRIVATE_OPTIMIZED */
            system;               /* RSA, ECC or DH ?   */

    char    name[MAXLEN],         /* various info's about this key */
            email[MAXLEN],
            description[MAXLEN];
```

```
    unsigned long ID;             /* CRC32 of the name/email/description together */

    _pk_key key;

    struct Pk_key  *next;         /* linked list chain */
} pk_key;
```

The list is chained via the "next" member and terminated with the node of the list that has "system" equal to **NON_KEY**.

10.2 The Keyring API

To initialize a blank keyring the function "kr_init()" is used.

```
int kr_init(pk_key **pk);
```

You pass it a pointer to a pointer of type "pk_key" where it will allocate ram for one node of the keyring and sets the pointer.

Now instead of calling the PK specific "make_key" functions there is one function that can make all three types of keys.

```
int kr_make_key(pk_key *pk, prng_state *prng, int wprng,
                int system, int keysize, const char *name,
                const char *email, const char *description);
```

The "name", "email" and "description" parameters are simply little pieces of information that you can tag along with a key. They can each be either blank or any string less than 256 bytes. "system" is one of the enumeration elements, that is **RSA_KEY**, **DH_KEY** or **ECC_KEY**. "keysize" is the size of the key you desire which is regulated by the individual systems, for example, RSA keys are limited in keysize from 128 to 512 bytes.

To find keys along a keyring there are two functions provided:

```
pk_key *kr_find(pk_key *pk, unsigned long ID);

pk_key *kr_find_name(pk_key *pk, const char *name);
```

The first searches by the 32-bit ID provided and the latter checks the name against the keyring. They both return a pointer to the node in the ring of a match or **NULL** if no match is found.

To export or import a single node of a keyring the two functions are provided:

```
int kr_export(pk_key *pk, unsigned long ID, int key_type,
              unsigned char *out, unsigned long *outlen);

int kr_import(pk_key *pk, const unsigned char *in);
```

The export function exports the key with an ID provided and of a specific type much like the normal PK export routines. The "key_type" is one of **PK_PUBLIC** or **PK_PRIVATE**. In this function with RSA keys the type **PK_PRIVATE_OPTIMIZED** is the same as the **PK_PRIVATE** type. The import function will read in a packet and add it to the keyring.

To load and save whole keyrings from disk:

```
int kr_load(pk_key **pk, FILE *in, symmetric_CTR *ctr);

int kr_save(pk_key *pk, FILE *out, symmetric_CTR *ctr);
```

Both take file pointers to allow the user to pre-append data to the stream. The "ctr" parameter should be setup with "ctr_start" or set to NULL. This parameter lets the user encrypt the keyring as its written to disk, if it is set to NULL the data is written without being encrypted. The load function assumes the list has not been initialized yet and will reset the pointer given to it.

There are the four encrypt, decrypt, sign and verify functions as well

```
int kr_encrypt_key(pk_key *pk, unsigned long ID,
                   const unsigned char *in, unsigned long inlen,
                   unsigned char *out, unsigned long *outlen,
                   prng_state *prng, int wprng, int hash);

int kr_decrypt_key(pk_key *pk, const unsigned char *in,
                   unsigned char *out, unsigned long *outlen);
```

The kr_encrypt_key() routine is designed to encrypt a symmetric key with a specified users public key. The symmetric key is then used with a block cipher to encode the message. The recipient can call kr_decrypt_key() to get the original symmetric key back and decode the message. The hash specified must produce a message digest longer than symmetric key provided.

```
int kr_sign_hash(pk_key *pk, unsigned long ID,
                 const unsigned char *in, unsigned long inlen,
                 unsigned char *out, unsigned long *outlen,
                 prng_state *prng, int wprng);
```

```
int kr_verify_hash(pk_key *pk, const unsigned char *in,
                   const unsigned char *hash, unsigned long hashlen,
                   int *stat);
```

Similar to the two previous these are used to sign a message digest or verify one. This requires hashing the message first then passing the output in.

To delete keys and clear rings there are:

```
int kr_del(pk_key **_pk, unsigned long ID);
int kr_clear(pk_key **pk);
```

"kr_del" will try to remove a key with a given ID from the ring and "kr_clear" will completely empty a list and free the memory associated with it. Below is small example using the keyring API:

```
#include <mycrypt.h>
int main(void)
{
   pk_key *kr;
   unsigned char buf[4096], buf2[4096];
   unsigned long len;
   int errno;

   /* make a new list */
   if ((errno = kr_init(&kr)) != CRYPT_OK) {
      printf("kr_init: %s\n", error_to_string(errno));
      exit(-1);
   }

   /* add a key to it */
   register_prng(&sprng_desc);
   if ((errno = kr_make_key(kr, NULL, find_prng("sprng"), RSA_KEY, 128,
               "TomBot", "tomstdenis@yahoo.com", "test key")) == CRYPT_OK) {
      printf("kr_make_key: %s\n", error_to_string(errno));
      exit(-1);
   }

   /* export the first key */
   len = sizeof(buf);
   if ((errno = kr_export(kr, kr->ID, PK_PRIVATE, buf, &len)) != CRYPT_OK) {
      printf("kr_export: %s\n", error_to_string(errno));
```

```
        exit(-1);
    }

    /* ... */
}
```

Chapter 11

$GF(2^w)$ 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 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 <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 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 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. Most options are placed in the makefile and others are in "mycrypt_cfg.h". All are used when the library is built from scratch.

For GCC platforms the file "makefile" is the makefile to be used. On MSVC platforms "makefile.vc" and on PS2 platforms "makefile.ps2".

14.2 mycrypt_cfg.h

The file "mycrypt_cfg.h" is what lets you control what functionality you want to remove from the library. By default, everything the library has to offer it built.

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 the system will automatically detect GCC or MSVC on a windows platform as well as GCC on a PS2 platform.

14.3 The Configure Script

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

X memory routines

The makefiles must define three macros denoted as XMALLOC, XCALLOC 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.

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.

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_ERROR. This should help resolve any linker errors stemming from a lack of file I/O on embedded platforms.

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.

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.

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.

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.