Mulle kybernetiK presents http://www.mulle-kybernetik.com

The MulleCipher Cryptography Framework v1.1

Containing

Digests

MD-5
SHA-1
CRC32 (not by default)

Ciphers

BlowFish
TwoFish

Thanks to Andy Lee and Derrel Piper for lots of useful suggestions for version 1.0.

Where is the documentation ?

It's currently mostly in the headers, some could be in the source. Please read our copyright notices and

disclaimers. For a sensitive framework like this, this is extra important! OK lets not be too unfriendly, here is some code that shows how to create a MD5 digest from a NSString

```
   #import <MulleCipher/MulleCipher.h>
   ...
   NSData   *plaintext;
   NSData   *digest;

   plaintext = [@"The Fragile Art Of Existence"
dataUsingEncoding:NSISOLatin1StringEncoding];
   digest    = [data md5Digest];
```

Encryption is really easy now.

Encryption is also easy, lets use Blowfish to encrypt that same data. (The key in a real application would be a random number of appropriate keySize):

```
   MulleSymmetricCipher     *cipher;
   MulleSymmetricCipherKey  *cryptKey;
   NSDictionary             *ciphered;
   NSString                 *text;
   NSData                   *plaintext;

   plaintext = [@"The Fragile Art Of Existence"
dataUsingEncoding:NSISOLatin1StringEncoding];
   password = [@"is kept alive by"
dataUsingEncoding:NSISOLatin1StringEncoding];
   cipher   = [MulleSymmetricCipher cipherWithName:@"Blowfish"];

   ciphered = [cipher crypt:plaintext
                   password:password];
```

decrpytion than goes like this:

```
   plaintext = [cipher decrypt:ciphered
                      password:password];
   text = [NSString stringWithData:plaintext
                     usingEncoding:NSISOLatin1StringEncoding];
```

The other way to encrypt

Another way to encrypt data is this slightly longer version, where you generate the encryption and decryption keys manually:

```
   MulleSymmetricCipher     *cipher;
   MulleSymmetricCipherKey  *cryptKey;
   id                       plaintext;
   NSData                   *ciphertext;
   unsigned int             plaintextLen;

   plaintext = [@"The Fragile Art Of Existence"
dataUsingEncoding:NSISOLatin1StringEncoding];
   password = [@"is kept alive by"
dataUsingEncoding:NSISOLatin1StringEncoding];
```

```
   cipher   = [MulleSymmetricCipher cipherWithName:@"Blowfish"];
   cryptKey = [[cipher keyClass] keyWithData:password
                                forDirection:MulleCipherForEncryption];
   plaintextLen = [plaintext length];  // remember original length
   ciphertext = [data encryptedData:plaintext
                            withKey:cryptKey];
```

decrpytion than goes like this

```
   cryptKey = [[cipher keyClass] keyWithData:password
                                forDirection:MulleCipherForDecryption];
   plaintext = [ciphertext decryptedData:ciphertext
                                 withKey:cryptKey];
   plaintext = [plaintext subdataWithRange:NSMakeRange( 0,
plaintextLen)];   // truncate
   text = [NSString stringWithData:plaintext
                     usingEncoding:NSISOLatin1StringEncoding];
```

The chief advantage here is that you can reuse the cryptKey for subsequent encryptions. Since creating a key is a potentially expensive operation, this may improve performance quite a bit. **Due to the nature of the Blowfish and Twofish algorithms the decrypted data may be enlarged by a small amount of random bytes at the end.** This is because Blowfish and Twofish encrypt blocks of memory, that are 8 respectively 32 bytes long. If the tail bytes of a plaintext do not fit perfectly into such a block, the remaining bytes will be padded with random bytes. These random bytes will appear in the decrypted plaintext. Therefore the receiver of a ciphertext should also get the size of the original plaintext with the ciphertext and truncate the data accordingly.

Storing the Plaintext length in the data

If you don't want to use the new methods `crypt:password: decrypt:password: methods` - introduced in v1.0, you may use as an alternative the methods `encryptedDataWithFooter:withKey: and decryptedDataWithFooter:withKey:` to append information about the size of the encrypted plaintext. Every ciphertext encrypted with one of the new ...WithFooter methods, will have (currently) 8 bytes appended, that encode the length of the plaintext.

Using these methods, you would modify the code above lile this

```
   ciphertext = [data encryptedDataWithFooter:plaintext
                                      withKey:cryptKey];
```

and

```
   plaintext = [ciphertext decryptedDataWithFooter:ciphertext
                                           withKey:cryptKey];
```

Although this may sound convenient, it means that a 100% proper decrypt of the encoded message can only be done with a program also using the MulleCipher size encoding scheme. A different encoder will find some (more) "garbage" at the end of its plaintext message.

You may check if a data contains a footer by asking `hasFooterAppended:` with the ciphertext. If there is no footer you may still use the withFooter routines which will silently assume that there is no padding in the ciphertext.

How do I use the Framework ?

You compile it and link it to your program, or - maybe preferable - add the desired subproject to your existing code. Both subprojects are independent of each other.
To compile and install, the following shell commands should work for Mac OS X and Mac OS X Server. Be sure to say install_all to have the debug frameworks available.

```
bash$ gnutar xfz MulleCipher.source.tgz
bash$ cd MulleCipher
bash$ su
Password:
bash$ make install_all
```

How do I use the Framework with the new Project Builder?

Start Project Builder and IMPORT the framework. Everything should work out ok.

What is a Digest ?

A digest is sort of a digital fingerprint of some binary data. A digest promises to produce for a unique amount of data a unique digest, which is just a short string of binary data itself.

The claim may sound dubious. If you were to use an digest algorithm like MD5 that provides digests of 16 bytes length, and you are inputting data of 17 bytes length, you can mathemetically expect that there are 256 value combinations of those 17 bytes, where there MUST be duplicate digests. The idea behind such a digest is though, that it would be computationally infeasible to create such a duplicate. Imagine having to try all possible combinations by brute force! You would not make it in your lifetime even with a million supercomputers. (1)
So it should be extremely unlikely, that two amounts of data produce the same digest.

One use for a digest could be to compare the contents of a local MP3 file with a file on an internet server, instead of having to download that file, to then find out that it the same lame remix you already have. Instead you'd just be comparing your digest with his digest.

The MulleCipher framework provides two digest types, one is called MD5, which is fairly popular in Internet applications and one is called SHA1, which is used for example in JAR files. SHA1 produces longer digests and is considered more secure, i.e. less likely to produce duplicates.

I have run a few test cases with MD5 and SHA-1 and the implementations appear to work correctly on a PowerPC with a Gnu Compiler.

Historically crc32 has been used for somewhat of the same purpose, validating and error checking hunks of data. The 4 byte data length of the crc32 and the algorithm itself does not lend itself to digital fingerprinting though. A version of this algorithm has also been included. It is a more lightweight algorithm and should be faster in all circumstances.

What is a Cipher ?

A cipher is a way to encrypt binary data with a certain key, so that is very hard, neigh almost impossible to recreate the original values without the key. A symmetric cipher has a single key, that is used for encryption and decryption. PGP, which uses two keys, a public and a private, is not a symmetric cipher. The two ciphers cotained in MulleCipher are Blowfish and Twofish - both brainchilds of Bruce Schneier -

are symmetric ciphers.

Twofish is considered the stronger encryption of the two and was a top 5 finalist for DES replacement in the NIST competition this year (2000 ?).

Both implementations seem to work, but I have not tested that their output is compatible with their Intel implementations.

To use this technology properly it, you need a little background on the subject, else the NSA or some other friendly agency will crack your ciphertext in no time. So get a copy of Bruce Schneiers "AppliedCryptography", if you want to use this code "real seriously".

Public Key encryption

Is not a part of the MulleCipher framework.

How safe are Blowfish and Twofish ?

Answering these questions means writing a book about it. Recurring theme: get "Applied Cryptography".

Some Material in the Net

SHA-1 http://security.isu.edu/isl/fip180-1.html
MD5 http://www.faqs.org/rfcs/rfc1321.html
BlowFish, TwoFish http://www.counterpane.com
Mersenne Twister: http://www.math.keio.ac.jp/~matumoto/emt.html

(1)
With a billion supercomputers, though there might be hope! Figure it like this for MD-5: (note: 2^32 is ca. 4 billion (US), 4 Milliarden (German))

In 16 bytes (128 bits) you have 2^(16*8) possible bit patterns. (256*256*256*256*256*256*256*256*256*256*256*256*256*256*256*256*). Now if you were able to try lets say 2^32 patterns with one machine (pretty fine machine!) in one second, it would still take you 2^(12*8) seconds. If you employ 2^32 machines in parallel, the time left is still 2^32 seconds. If you consider a 50% chance to hit the duplicate early, that reduces to 2^31 seconds. Now that is only 70 years!
SHA-1 demands a cryogenic solution...