

# LibCdsaCrypt – A High Level CDSA-based Crypto Library

Copyright © 2003 Apple Computer Inc.  
All Rights Reserved  
Last Update 2/4/2003

## 1.0 Summary

This document describes libCdsaCrypt, a library intended to facilitate the use of Security.framework's Common Data Security Architecture (CDSA), specifically in the areas of symmetric encryption and message digest calculation. The libCdsaCrypt package contains the source for libCdsaCrypt, a static library, and the source for several example programs demonstrating the use of libCdsaCrypt.

## 2.0 CDSA Overview

The CDSA API is a complex and comprehensive cryptographic architecture which provides a standardized interface to just about any known or conceivable crypto device, module, or function. Unfortunately, as a result, the CDSA API and its accompanying documentation are rather challenging to the casual programmer, who may wish to just perform some common cryptographic operation and move on to something else. The libCdsaCrypt library is intended to provide:

- A high-level, first-cut library, written in C, with which a large percentage of cryptographic operations can be performed,
- Source code for the library for use by those who wish to extend libCdsaCrypt's functionality, and
- Source code for example programs which demonstrate the use of every function in libCdsaCrypt.

One basic CDSA concept which must be understood is the **handle**. Many CDSA-based resources, the details of which are opaque at the public CDSA API, are represented by handles. One type of handle, denoted by the type `CSSM_CSP_HANDLE` (all data types are defined in `<Security/cssmtype.h>`), represents an application's "connection" to a Cryptographic Service Provider, or CSP. A CSP is conceptually a plugin or bundle which is loaded into the application's address space on demand. All cryptographic operations are performed by a CSP. Although there are multiple CSPs in MacOS X, the libCdsaCrypt package only uses one, so you never have to specify "which CSP". (A description of the various CSPs in the system and what they are used for is beyond the scope of this document.) You do, however, have to attach to the (sole) CSP before using any other function in libCdsaCrypt. This attachment is performed by the `cdsaCspAttach()` function, described in section 3.1 (like all other libCdsaCrypt functions, this function is declared in `libCdsaCrypt.h`).

Another type of handle which some users of libCdsaCrypt might optionally use is the Cryptographic Context Handle, represented by a `CSSM_CC_HANDLE`, and colloquially called a "context handle". A context handle represents the state associated with a cryptographic operation which spans more than one function call. Actually, all crypto operations at the CDSA layer span multiple function calls, and thus require the use of a context handle; one benefit of libCdsaCrypt is that many crypto operations can be performed with the application knowing about or maintaining context handles.

### 3.0 libCdsaUtils Functions

One thing must be understood by those who consider using libCdsaCrypt. This library serves to simplify access to the underlying CDSA functionality. As a result, many details of the CDSA API are hidden from view at the libCdsaCrypt level. This also means that many functions and options of CDSA are also hidden from view. If there is something crypto-related you're trying to do, and libCdsaCrypt just doesn't quite do exactly what you want, then you will have to access CDSA directly, where you can almost certainly find a way to do exactly what you want. The source for libCdsaCrypt is available for customization and inspection; the intent here is to get you started and to point the way.

Perhaps the best way to study the usage of the functions in libCdsaCrypt is to examine the real-world example programs described in section 4.0 of this document. Usage of every function in the library can be found in the Examples.

All of the functions described in this section are declared in the header file libCdsaCrypt.h. Data types common to CDSA all start with the prefix "CSSM\_" and are declared in the system header file `<Security/cssmtype.h>`. Error returns from all CDSA functions and all functions in libCdsaCrypt are of type `CSSM_RETURN`; values for this type are found in `<Security/cssmerr.h>`. The values declared in that file are, however, not very readable by humans, so you'll probably need to use the `cssmPerror()` function, declared in `<Security/cssmapple.h>`.

### 3.1 Initialization

As mentioned above, you must attach to the CSP before performing any cryptographic operations using this library. This is performed as follows:

```
CSSM_RETURN          crtn;
CSSM_CSP_HANDLE      cspHandle;

crtn = cdsaCspAttach(&cspHandle);
if(crtn) {
    cssmPerror("Attach to CSP", crtn);
    /* handle the error; this one is actually fatal */
}
```

Most subsequent calls to libCdsaCrypt will take the `CSSM_CSP_HANDLE` you obtain in this way as an argument.

When you are all finished with libCdsaCrypt, release this CSP handle by calling `cdsaCspDetach()`.

### 3.2 Symmetric Key Generation

Symmetric encryption is the sort of encryption in which the same key is used to encrypt and decrypt data. The "key" is the secret data which enables one to decrypt. Therefore, the entity (person or program) which is doing the encryption has to share some secret data – the key – with the entity doing the decryption. Exactly how the two entities share this secret without compromising the secret is a subject far beyond the scope of this document.

Keys are represented in the CDSA world by a type `CSSM_KEY`, a pointer to which is a `CSSM_KEY_PTR`. There are many ways to create a `CSSM_KEY`; one way provided by libCdsaCrypt is the `cdsaDeriveKey()` function. This function takes as its input some raw key material – typically a

passphrase, but it could be any data at all – and some other parameters, and uses an industry standard technique to repeatably derive a `CSSM_KEY`. (The industry standard is defined in PKCS5.) All encryption and decryption operations in `libCdsaCrypt` take a `CSSM_KEY_PTR` as an argument, and unless you know how to generate all of the information in a `CSSM_KEY` yourself, you'll be best served by using the `cdsaDeriveKey()` call.

The `cdsaDeriveKey()` function is declared as follows:

```
CSSM_RETURN cdsaDeriveKey(  
    CSSM_CSP_HANDLE    cspHandle,  
    const void          *rawKey,  
    size_t              rawKeyLen,  
    CSSM_ALGORITHMS     keyAlg,  
    uint32               keySizeInBits,  
    CSSM_KEY_PTR        key);
```

Where

`cspHandle` is the CSP handle you obtained from `cdsaCspAttach()`.

`rawKey` and `rawKeyLen` are the raw key material (e.g., a passphrase)

`keyAlg` is cryptographic algorithm associated with the key.

`keySizeInBits` is the size of the key to generate.

`key` is a pointer to a `CSSM_KEY` struct which you have allocated. It may be completely uninitialized upon entry to this function.

Note that the size of the raw key material you provide to this function does not have any relation to the size of the key being generated. You can use this function to derive a 1000-bit RC5 key from a 4-byte passphrase; you could use it to derive a 64-bit DES key from a thousand bytes of data.

You do have to be aware of the allowable (or required) key sizes for a given algorithm. Currently supported key algorithms, along with their allowable key sizes, are

<code>CSSM_ALGID_DES</code>	64 bits
<code>CSSM_ALGID_3DES_3KEY</code>	192 bits
<code>CSSM_ALGID_RC5</code>	8-2040 bits
<code>CSSM_ALGID_RC2</code>	8-1024 bits
<code>CSSM_ALGID_AES</code>	128 bits
<code>CSSM_ALGID_ASC</code>	8-512 bits
<code>CSSM_ALGID_RC4</code>	8-4096 bits

Note: if you need advice on which algorithm to choose, use AES (`CSSM_ALGID_AES`). For one thing, this algorithm is a well-accepted standard; it was accepted by FIPS as the U.S. Government standard in October 2000. It has received massive peer review and scrutiny in the Security community and is widely believed to be very secure. Also, MacOS X has a very fast implementation of AES – about 20 Mbytes per second, encrypting, on a 800 MHz G4. (The implementation used by `libCdsaCrypt` uses 128-bit keys and blocks; the CSP actually allows for other keys and block sizes.) The MacOS X DiskCopy application uses AES when encryption is enabled. There is really no reason to use any other algorithm these days unless you need to interoperate with other code which is hard-coded to some other algorithm.

When you are finished with a `CSSM_KEY` (generated by `cdsaDeriveKey()`), free its associated resources using `cdsaFreeKey()`.

### 3.3 Symmetric Encryption and Decryption

At this point a “feature” of `libCdsaCrypt`’s encryption routines needs to be discussed. As mentioned earlier, a benefit of using `libCdsaCrypt` is that many complications of the low-level CDSA API are glossed over. In fact many options available at the CDSA layer are also missing at the `libCdsaCrypt` layer, and a significant number of those missing options are right here in the area of symmetric encryption. The `libCdsaCrypt` encrypt and decrypt routines do not provide you with any means of specifying block size, or padding, or initialization vectors, or chaining mode. If these terms mean nothing to you, great! This stuff will work just fine. If these things concern you, be advised that padding and chaining mode are selected as appropriate for most normal uses of the algorithm of the specified `CSSM_KEY`. All block ciphers are performed with Cipher Block Chaining mode and normal PKCS5/7 padding. An initialization vector of all zeroes is used. (Stream ciphers use none of these options.) The upshot is: the default values selected for padding and chaining provide a cipher which can be used without regard to block size or block boundaries, and in general are as secure as the given algorithm can be (less the fact that a constant IV is used). If you need control over these parameters, then you need to either customize `libCdsaCrypt` or else program using the CDSA API.

Once you have obtained a `CSSM_KEY`, you are ready to perform actual encryption and/or decryption. There are two general styles of these functions provided in `libCdsaCrypt`. The first style is called “one-shot”, and involves one single function call. This style is used when you have all of the data you wish to encrypt or decrypt at one time.

The other style is called “staged” encryption or decryption. In this case, data to process may be available in portions here and there, for example when reading from a large disk file or from a network. In this style, one first makes an initialization call, then performs an arbitrary number of actual encrypt or decrypt calls; on the last one of these calls, a ‘final’ argument is set true to indicate that the operation is to be completed. When performing staged operations, the caller does not have to be concerned with, or even know about or understand, things like block boundaries, alignment, or anything related to how much data is passed in to each particular encrypt or decrypt operation. The caller does, however, have to maintain the state of the staged operation between calls, in the form of a `CSSM_CC_HANDLE` which is created during the initialization call.

One data type you need to familiarize yourself with to use any of the following functions is `CSSM_DATA`, which is a wrapper for a pointer and a length:

```
typedef struct cssm_data {
    uint32 Length; /* in bytes */
    uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR;
```

This is the “CDSA way” to represent almost any untyped data.

The one-shot functions are very simple and look like this:

```
CSSM_RETURN cdsaEncrypt(
    CSSM_CSP_HANDLE    cspHandle,
    const CSSM_KEY      *key,
    const CSSM_DATA     *plainText,
    CSSM_DATA_PTR       cipherText);
```

```

CSSM_RETURN cdsaDecrypt(
    CSSM_CSP_HANDLE    cspHandle,
    const CSSM_KEY      *key,
    const CSSM_DATA      *cipherText,
    CSSM_DATA_PTR        plainText);

```

Where `cspHandle`, of course, is obtained from `cdsaCspAttach()`, and `key` is obtained from `cdsaDeriveKey()`. The encrypt operations takes `plainText` as input and returns `cipherText`; the decrypt operation is the reverse. The caller has to free the returned data referent (`cipherText->Data` in the encrypt case, `plaintext->Data` in the decrypt case) via the standard C `free()` function when finished with it. (This memory is `malloc()`d by the CSP during the function call.)

When performing staged operations, one first calls this function:

```

CSSM_RETURN cdsaStagedEncDecrInit(
    CSSM_CSP_HANDLE    cspHandle,
    const CSSM_KEY      *key,
    StagedOpType        opType,
    CSSM_CC_HANDLE      *ccHandle);

```

Where

`cspHandle` is the CSP handle you obtained from `cdsaCspAttach()`.

`key` is the `CSSM_KEY` prepared in `cdsaDeriveKey()`.

`opType` is one of the (self-explanatory) following enums:

```

typedef enum {
    SO_Encrypt,
    SO_Decrypt
} StagedOpType;

```

`ccHandle` is returned by the function and is used in subsequent encrypt or decrypt calls.

Then one of the following two functions is called an arbitrary number of times:

```

CSSM_RETURN cdsaStagedEncrypt (
    CSSM_CC_HANDLE      ccHandle,
    CSSM_BOOL            final,
    const CSSM_DATA      *plainText,
    CSSM_DATA_PTR        cipherText);
CSSM_RETURN cdsaStagedDecrypt (
    CSSM_CC_HANDLE      ccHandle,
    CSSM_BOOL            final,
    const CSSM_DATA      *cipherText,
    CSSM_DATA_PTR        plainText);

```

On the last such call, the final argument must be `CSSM_TRUE`; on all previous calls the final argument must be `CSSM_FALSE`. As with the one-shot operations, the caller must `free()` the returned data referent after each call. On the final call, the input data (`plainText` for encrypt, `cipherText` for decrypt) can be `NULL`, or its referent (e.g., `plaintext->Data`) can be `NULL`. The `ccHandle` which was created in `cdsaStagedEncDecrInit()` is freed during the final call.



### 3.4 Digest operations

As with symmetric encryption, digest operations can be performed in “one-shot” mode or in staged mode. Digest operations do not require `CSSM_KEYs`. They take an arbitrary size of data as input and yield a small (16 – 20 bytes) output upon completion. Usage of the digest functions declared in `libCdsaCrypt.h` is straightforward and should be clear by examining the function declarations, the associated comments, and examining the `DigestTool` example program (see section 4.2).

The supported digest algorithms are `CSSM_ALGID_MD5` and `CSSM_ALGID_SHA1`.

### 3.5 Asymmetric Key Generation

Asymmetric cryptography involves the use of two keys – a public key and a private key. Data which is encrypted with a public key can only be decrypted by the corresponding private key. Digital signatures generated using the private key can be verified using the corresponding public key.

One function is available in `libCdsaUtils` to generate an asymmetric key pair:

```
CSSM_RETURN  cdsaGenerateKeyPair(  
    CSSM_CSP_HANDLE      cspHandle,  
    CSSM_ALGORITHMS      keyAlg,  
    uint32                keySizeInBits,  
    CSSM_KEY_PTR          publicKey,  
    CSSM_KEY_PTR          privateKey);
```

Where

`cspHandle` is the CSP handle you obtained from `cdsaCspAttach()`.

`keyAlg` is cryptographic algorithm associated with the key. Currently the only supported algorithms are `CSSM_ALGID_RSA` and `CSSM_ALGID_DSA`.

`keySizeInBits` is the size of the key to generate.

`publicKey` and `privateKey` are pointers to `CSSM_KEY` structs which you have allocated. They may be completely uninitialized upon entry to this function.

Note that unlike when generating (deriving) symmetric keys, asymmetric keys do not derive from a user-supplied passphrase. Each key pair is unique. For purposes of public key exchange, the `CSSM_KEY.KeyData.Data` blob of the public key is the material one can safely distribute, in the clear.

To perform encryption and decryption using the public key and private key, respectively, you can use the same encryption routines described in section 3.3. PKCS1 padding is performed when using RSA keys. Encryption and decryption are not possible when using DSA keys.

See section 4.3 for more information on the use of asymmetric keys.

Note that in the “real world”, contents of private keys are not normally kept in the clear. On MacOS X, private keys are typically kept in a keychain which is protected by a user’s passphrase. Generation and maintenance of private keys using the Keychain mechanism is outside the scope of this document.

### 3.5 Diffie-Hellman Key Generation and Key Exchange

Diffie-Hellman Key Exchange is the first public key cryptosystem ever invented (at least it's the first that was publicly disclosed) and is still in use today. It allows two parties to create a unique symmetric key which they can both use to encrypt or decrypt, by exchanging data with each other in the clear; no third party can guess or reproduce the created symmetric key.

In general, Diffie-Hellman Key Exchange occurs in three phases:

1. Algorithm parameters are calculated and agreed upon. These parameters are shared between any and all parties, in the clear. Security of Diffie-Hellman Key Exchange is not compromised in any way by the public sharing of these parameters. Calculation of these parameters is a time-consuming process; it takes about 90 seconds on an 800 Mhz G4 to calculate algorithm parameters for 1024-bit Diffie-Hellman keys. Therefore these parameters are not calculated with great frequency. At the level of CDSA and of libCdsaCrypt, these algorithm parameters are opaque data blobs, represented by a `CSSM_DATA` or a pointer to one. Such a blob is referred to as a Parameter Block.
2. Each party generates a key pair – one public key, one private key. Your public key is published in the clear for use by anyone wishing to perform a key exchange operation with you.
3. The actual Diffie-Hellman key exchange uses one person's private key and another person's public key to calculate a "secret", known only to the two parties. This secret is typically used as the key material in symmetric encryption operations.

LibCdsaCrypt provides the following two functions to accomplish the above three steps. The first function generates a key pair, and optionally, a Parameter Block:

```
CSSM_RETURN cdsaDhGenerateKeyPair(  
    CSSM_CSP_HANDLE cspHandle,  
    CSSM_KEY_PTR     publicKey,  
    CSSM_KEY_PTR     privateKey,  
    uint32            keySizeInBits,  
    const CSSM_DATA *inParams,  
    CSSM_DATA_PTR     outParams);
```

Where

`cspHandle` Is the CSP handle obtained from `cdsaCspAttach()`.

`publicKey` and `privateKey` are pointers to uninitialized caller-supplied `CSSM_KEY` structs.

`keySizeInBits` Is the size of the Diffie-Hellman keys to be generated.

`inParams` and `outParams` are pointers to Diffie-Hellman Algorithm Parameter blocks.

Exactly one of (`inParams`, `outParams`) must be non-NULL when calling this function. That is, you either pass an existing Parameter Block when you call this function, or you are asking this function to generate one for you and return it to you along with the key pair. Note that a single Diffie-Hellman key pair, generated with its own unique algorithm parameters, is useless. At least two different key pairs must be generated using the same algorithm parameters for Diffie-Hellman key exchange to occur.

The following function performs the actual Diffie-Hellman key exchange, taking as its input one private key and one public key, and creating as its output a symmetric key with the specified algorithm and size.

```
CSSM_RETURN cdsaDhKeyExchange(  
    CSSM_CSP_HANDLE cspHandle,
```

```

CSSM_KEY_PTR    myPrivateKey,
const void      *theirPubKey,
uint32          theirPubKeyLen,
CSSM_KEY_PTR    derivedKey,
uint32          deriveKeySizeInBits,
CSSM_ALGORITHMS derivedKeyAlg);

```

Where

`cspHandle` Is the CSP handle obtained from `cdsaCspAttach()`.

`myPrivateKey` Is the private key obtained from `cdsaDhGenerateKeyPair()`.

`theirPubKey` and `theirPubKeyLen` specify the “other person’s” public key in the form of an untyped data blob. This is actually obtained from the `publicKey->KeyData` field when the “other person” does their own `cdsaDhKeyExchange()`.

`derivedKey` is a pointer to a uninitialized caller-supplied `CSSM_KEY` struct.

`deriveKeySizeInBits` and `derivedKeyAlg` define the symmetric key being derived, which is returned in `*derivedKey`. This key is typically used in subsequent symmetric encryption operations. See section 4.4 for an example.

### 3.6 Signature Generation and Verification

One use – probably the most common use – of asymmetric encryption is that of digital signatures. “Signing” refers to the process in which a block of data to be signed and a private key are used to generate a digital signature block; when “verifying” a signature, the data to be signed, the signature, and the corresponding public key are used. Signature verification provides a means of ensuring that a block of data has not been modified since it was signed, and that it was in fact signed by an entity which has access to the private key corresponding to the public key you’re verifying with.

As with encryption and digest generation, `libCdsaCrypt` provided both a simplified one-shot sign and verify, and a set of functions for performing staged operations.

The one-shot functions are look like this:

```

CSSM_RETURN cdsaSign(
    CSSM_CSP_HANDLE    cspHandle,
    const CSSM_KEY      *key,
    CSSM_ALGORITHMS     sigAlg,
    const CSSM_DATA     *dataToSign,
    CSSM_DATA_PTR       signature);

CSSM_RETURN cdsaVerify(
    CSSM_CSP_HANDLE    cspHandle,
    const CSSM_KEY      *key,
    CSSM_ALGORITHMS     sigAlg,
    const CSSM_DATA     *dataToSign,
    const CSSM_DATA     *signature);

```

Where

`cspHandle` Is the CSP handle obtained from `cdsaCspAttach()` .

`key` is a private or public key, for signing and verifying, respectively.

`sigAlg` is the signature algorithm. Examples are `CSSM_ALGID_SHA1WithRSA` for use with RSA keys and `CSSM_ALGID_SHA1WithDSA` for use with DSA keys.

`dataToSign` is the data over which digital signature generation or verification is to be performed.

`signature` is the digital signature. For `cdsaSign()` , this data is allocated by the CSP and must be freed via `free()` by the caller. For `cdsaVerify()` , this data is an input parameter.

When performing staged signing or verifying operations, one first calls this function:

```
CSSM_RETURN cdsaStagedSignVerifyInit(  
    CSSM_CSP_HANDLE    cspHandle,  
    const CSSM_KEY      *key,  
    CSSM_ALGORITHMS     sigAlg,  
    StagedOpType        opType,  
    CSSM_CC_HANDLE      *ccHandle);
```

Where

`cspHandle` is the CSP handle you obtained from `cdsaCspAttach()` .

`key` is a private or public key, for signing and verifying, respectively.

`sigAlg` is the signature algorithm. Examples are `CSSM_ALGID_SHA1WithRSA` for use with RSA keys and `CSSM_ALGID_SHA1WithDSA` for use with DSA keys.

`opType` is one of `{SO_Sign, SO_Verify}`.

`ccHandle` is returned by the function and is used in subsequent encrypt or decrypt calls.

Then one of the following two functions is called an arbitrary number of times:

```
CSSM_RETURN cdsaStagedSign (  
    CSSM_CC_HANDLE      ccHandle,  
    const CSSM_DATA      *dataToSign,  
    CSSM_DATA_PTR        signature);  
CSSM_RETURN cdsaStagedVerify (  
    CSSM_CC_HANDLE      ccHandle,  
    const CSSM_DATA      *dataToVerify,  
    const CSSM_DATA      *signature);
```

On the last such call, the signature argument must be non-NULL; on all previous calls the signature argument be NULL. This signature argument is an output for `cdsaStagedSign` and input for `cdsaStagedVerify`. As with the one-shot operation, the caller must `free()` the returned signature referent after the final call to `cdsaStagedSign`. On the final call, the input data (`dataToSign` for sign, `dataToVerify` for verify) can be NULL, or its referent (e.g., `dataToSign->Data`) can be NULL. The `ccHandle` which was created in `cdsaStagedSignVerifyInit()` is freed during the final call.

## 4.0 Example programs

The Examples directory in the libCdsaCrypt module contains a number of programs which provide more-or-less real-world examples of usage of every function in libCdsaCrypt. All of the examples are UNIX command-line tools which operate on files (e.g., encrypt file1 to file2, calculate the digest of file3 and place the digest in file4, etc.). All can be run with no arguments to get usage information. A brief discussion of each program follows.

### 4.1 CryptTool and StagedCrypt

These programs demonstrate the operation of symmetric key derivation and encryption. CryptTool uses the one-shot encrypt/decrypt style; StagedCrypt used the staged style. Functionally – from an external point of view – they produce exactly the same results. Both take as command line arguments a command ('e' or 'd', for encrypt or decrypt, respectively), a passphrase which will be used to derive a key, a key size (in bits), the name of an input file, the name of an output file, and an optional algorithm specifier (the default algorithm is AES).

For example to encrypt file foo, placing the ciphertext in file bar, with the passphrase “reallySecure”, using DES (not the default of AES), you would run

```
# CryptTool e reallySecure 128 foo bar a=d
```

You could have run StagedCrypt with the same argument and gotten the same result in file bar.

To decrypt, you'd run

```
# CryptTool d reallySecure 128 bar foo a=d
```

The source directories for each of these programs contains a script which does a quick “create plaintext file, encrypt it, display the contents of the ciphertext, decrypt, and compare the result with the original file”. The scripts are named runCrypt and runStaged and run with no arguments.

### 4.2 DigestTool

This program calculates the digest of a file, placing the digest in another file. An optional command line argument selects staged model the default is one-shot; both styles of digest calculation are demonstrated in the source. Another optional argument selects the MD5 algorithm (the default is SHA1).

To calculate the digest of file foo using SHA1, placing the digest in file bar:

```
# DigestTool foo bar
```

### 4.3 RsaTool

This program performs operations using asymmetric keys with algorithm CSSM\_ALGID\_RSA and CSSM\_ALGID\_DSA. Supported operations include:

- Generate key pair, write keys to files
- Encrypt a file using specified public key

- Decrypt a file using specified private key
- Generate a digital signature for a file using specified private key, write result to a file. Both one-shot and staged operations are available.
- Verify a digital signature using specified data, public key, and signature

This program illustrates the use of the `cdsaEncrypt()` and `cdsaDecrypt()` functions when using RSA keys, and it shows a simple (one-shot) use of the CDSA signature-related functions. See the `rt_sign()` and `rt_verify()` functions for examples of signature generation and verification.

Some sample usages of RsaTool follow. User-entered commands are in **bold**.

- Generate a key pair, place result in `rsakey_{pub,priv}.der`

```
# RsaTool g k=rsakey
...wrote 74 bytes to rsakey_pub.der
...wrote 345 bytes to rsakey_priv.der
```

- Create a file `p.txt`, then encrypt file `p.txt` using `rsakey_pub.der`, place result in `c.txt`

```
# cat > p.txt
some plaintext to sign or encrypt
# RsaTool e k=rsakey p=p.txt c=c.txt
...wrote 64 bytes to c.txt
```

- Decrypt file `c.txt` using `rsakey_priv.der`, place result in `r.txt`. Display result.

```
# RsaTool d k=rsakey c=c.txt p=r.txt
...wrote 64 bytes to r.txt
# cat r.txt
some plaintext to sign or encrypt
```

- Sign file `p.txt`, using `rsakey_priv.der`, place result in `sig`

```
# RsaTool s k=rsakey p=p.txt s=sig
...wrote 64 bytes to sig
```

- Verify file `p.txt`, using `rsakey_pub.der`, and existing signature file `sig`. Used the staged verify option ('g').

```
# RsaTool v k=rsakey p=p.txt s=sig g
...signature verifies OK
```

- Modify file `p.txt`, verify signature again

```
# cat >> p.txt
append
# RsaTool v k=rsakey p=p.txt s=sig
sigVerify: CSP_VERIFY_FAILED
```

## 4.4 DiffieHellman

The DiffieHellman program illustrates the three phases of Diffie-Hellman key exchange – generation of algorithm parameters, key pair generation, and key exchange. The derived symmetric keys are then used to perform encryption and decryption as they would be used in the real world. In accordance with the requirements of the `cdsaDhGenerateKeyPair` function (see section 3.5), this program either generates a parameter block and writes it to a file, or it takes an existing parameter block from a file and uses this parameter block to generate Diffie-Hellman key pairs.

The sequence is as follows:

- Generate a D-H key pair, optionally using existing algorithm parameters obtained from a file. Call this key pair "myPublic" and "myPrivate".
- Generate another D-H key pair using the same algorithm parameters as used (or generated) in the previous step. Call this key pair "theirPublic" and "theirPrivate".
- Perform a D-H key exchange operations using myPrivate and theirPublic, resulting in symmetric key myDerive.
- Perform a D-H key exchange operations using myPublic and theirPrivate, resulting in symmetric key theirDerive.
- Encrypt some plaintext using myDerive.
- Decrypt the resulting ciphertext using theirDerive.
- Ensure that the result of the decryption yields the original plaintext.
- If the D-H algorithm parameter block was not originally obtained from an existing file, write the (newly generated) parameter block to a file.

Usage is simple: there is one command line argument, either “i=inFileName” or “o=outFileName”. A sample set of runs looks like this:

```
# DiffieHellman o=dhblob
Generating Diffie-Hellman parameters and key pairs...
Performing Diffie-Hellman key exchange...
Verifying key exchange...
...wrote 84 bytes to dhblob
Diffie-Hellman test complete.
# DiffieHellman i=dhblob
Generating Diffie-Hellman key pairs...
Performing Diffie-Hellman key exchange...
Verifying key exchange...
Diffie-Hellman test complete.
```

In this case, the first run of DiffieHellman resulted in the calculation of algorithm parameters, and took a significant length of time (as much as 20 seconds). The second run used the algorithm parameter block calculated in the first run (and stored in file dhblob), and hence executed very quickly (about a quarter of a second).