Chapter 4: Implementing Perfect Questionable Encryptions[*]

Adam L. Young and Moti M. Yung

Abstract

The notion of a questionable encryption scheme was first put forth in *Malicious Cryptography* [10]. In this chapter we present an implementation of the *perfect questionable encryption* scheme that appears in [9] that is based on our MyCrypt '05 paper [11]. A questionable encryption (q.e.) scheme is a generalization of a PKCS that includes an additional key generation algorithm that outputs a *fake* public key and a corresponding *witness of non-encryption*. The perfect q.e. scheme is based on the Paillier PKCS that produces a private key (that serves as a *witness of encryption*) and corresponding public key. The scheme includes a predicate F that takes as input a public key and corresponding witness. The output of F is a single bit that proves whether the public key is real or fake. A perfect q.e. scheme has the property that it is information theoretically impossible for everyone including the key pair owner to decrypt ciphertexts produced using the fake public key. Also, fake public keys are computationally indistinguishable from real public keys. It has been shown that a q.e. scheme is ideal for implementation in cryptoviruses, cryptoworms, and cryptotrojans. The use of a q.e. scheme in malware to "encrypt" data prior to transmission casts doubt on the occurrence of data theft, since prosecutors cannot directly prove that such malware encrypts data even when all code, all core dumps, and all packets that are transmitted by the malware are entered into evidence. The attacker retains the witness and can reveal it at any time. Questionable encryptions also serve as a desirable alternative to *oblivious transfer* in certain applications.

[*]If this file was obtained from a publicly accessible website other than the website www.cryptovirology.com then (1) the entity or entities that made it available are in violation of our copyright and (2) the contents of this file should therefore not be trusted. Please obtain the latest version directly from the official Cryptovirology Labs website at: http://www.cryptovirology.com.

1 Introduction

Consider a malware analyst that is given a malicious software program and is asked what it does. Suppose that the program collects data from the host system. Suppose further that it passes this data as plaintext to an asymmetric encryption function along with what appears to be a public key. The output ciphertext is transmitted over the public Internet (e.g., steganographically encoded into images that are posted to a public bulletin board). Is the malware stealing data? Can the analyst conclude this to be so? These questions are at the very heart of the notion of a *questionable encryption scheme*.

The notion of a questionable encryption was first investigated in [10]. This informal work formed the basis for further work in more formal settings. The notion of a *perfect* questionable encryption scheme was covered in [11] and the notion of a *computational* questionable encryption scheme appears in [12]. These papers point out the similarities and differences between questionable encryptions and: oblivious transfer, deniable encryption, and all-or-nothing disclosure of secrets.

The initial observation was that a public key can be constructed in such a way that it appears to be a valid public key, i.e., algebraically correct, but that has a fundamental flaw, a flaw that is so bad, in fact, that it is intractable for *everyone* to decrypt data encrypted with it. This gave rise to the idea that a fake key pair generator could be devised to output a fake public key and a corresponding small value, called a witness of non-encryption that *proves* that the public key is so flawed that the ciphertexts that it yields are intractable to decrypt. The scheme, called a questionable encryption scheme, not only hides plaintexts in a cryptographic sense, but also hides whether or not data is being encrypted at all.

The definition of a questionable encryption scheme satisfies the following properties. Informally, a fake public key must appear to all probabilistic polynomial time algorithms like a real public key (computational indistinguishability). Also, it must be intractable to produce a "public" key and both a witness of encryption and witness of non-encryption for it. In other words, once a user generates a key pair he or she is committed to the public key being either real or fake but not both.

The computational indistinguishability property is worth emphasizing since it is easily misunderstood. *It is entirely intractable* to determine if the plaintext is being encrypted vs. being destroyed *even while watching* the asymmetric encryption function encrypt using the public key (that is real or fake). There is no way to "trap" the encryption call and learn whether or not encryption is actually taking place.

But what use could a provably flawed yet normal-looking public key be? Consider again the malware that appears to the analyst to be enciphering data. Suppose that the public key is provably fake but that the witness is not included in the malware. If the malware author mounts an attack and is taken to court then the malware author can take the Fifth. If convicted of data theft the attacker can appeal and reveal the witness of non-encryption. This retroactively proves that the malware could not have possibly been used to steal data. The result is practically significant because it implies that when malware is properly designed (i.e., using cryptovirology) it is fundamentally intractable to prove data theft directly.

This attack can be expanded into a larger attack scenario. Suppose that hundreds of different malware strains are released that implement questionable encryptions. Suppose that some contain real public keys and others contain fake public keys. On occasion a malware author anonymously publishes his or her witness of non-encryption long after the malware is released. This would set a strong real-world precedent that malware that appears to asymmetrically encrypt and transmit data does not always do so.

No legal analyses will be attempted in this chapter. Our aim is to present an implementation of a questionable encryption scheme and shed light on its scientific aspects. Our implementation is based on the perfect q.e. scheme that will appear in IEEE Transactions on Information Forensics and Security [9]. This journal article is based on [11] that utilizes the Paillier public key cryptosystem [5]. We remark that the journal version expands upon [11]. However, the same basic idea remains unchanged. A perfect questionable encryption scheme has the property that it is information theoretically *impossible* to decipher data encrypted using a fake public key (in a computational q.e. scheme indecipherability is based on a *computational* argument).

This chapter goes beyond [9] by *showing* how to implement perfect questionable encryptions in a very computationally efficient manner. This chapter and the associated appendix (that contains the corresponding source code) therefore constitutes *fundamental research* in modern cryptovirology.

We chose to use the OpenSSL cryptographic library that is freely available as open source. At this time OpenSSL does not implement the Paillier public key cryptosystem. So, it was necessary for us to implement it ourselves. This chapter therefore serves as a tutorial on programming an efficient implementation of Paillier as well as implementing perfect questionable encryptions.

The example program is called `pquesencr.exe`. It implements a hybrid cryptosystem based on Paillier and Blowfish [8] in cipher block chaining

(CBC) mode. This hybrid cryptosystem is used to encrypt and decrypt files. However, the ‘perfect’ property is lost when a symmetric cipher such as Blowfish is used. This is because, whereas Paillier constitutes a perfect questionable encryption scheme, Paillier combined with Blowfish in a hybrid encryption scheme does not: a computationally unbounded adversary can brute-force a symmetric cipher such as Blowfish.

It is straightforward to extract the pieces of this program that perform Paillier encryption. The Paillier PKCS can be used in electronic code book (ECB) mode. This chapter therefore justifiably covers the implementation of a perfect questionable encryption scheme.

Our focus is on presenting a practical scheme. We combined Paillier with Blowfish to achieve the benefits of hybrid encryption. It should be noted that we by no means eliminate the properties of a questionable encryption scheme by using Paillier with Blowfish. Our use of Blowfish makes the overall questionable encryption scheme secure under a computational argument (that of breaking Blowfish). If the distinction here is not clear then we urge the reader to consult the scientific literature on the subject.

2 Creating the Program

The program consists of the following ANSI C source files: `pquesencr.c`, `kleptoutil.c`, `pqext.c`, and `pqemisc.c`. It also consists of the following header files: `pquesencr.h`, `kleptoutil.h`, `pqext.h`, and `pqemisc.h`. Compilation also requires OpenSSL. When installed, the needed OpenSSL library is typically included by supplying `−lcrypto` as an argument to `gcc`.

The source code given in this chapter covers all of the functions contained in `pquesencr.c`. We constructed a simple makefile for these files and compiled them using `gcc` in MinGW. The process produces the DOS command-line program `pquesencr.exe`.

3 Running the Program

For illustrative purposes, 128-bit primes `p` and `q` are used in the examples we give.[1] This is implemented by setting the #`define` **kover2** in `pqemisc.h` to be 128. By today’s standards primes of at least 512 bits would be needed for security, so this #`define` should be at least 512 in practice.

[1]Using 512-bit primes produces hexadecimal strings that are very long since Paillier uses $\mathtt{n}^2 = \mathtt{p}^2\mathtt{q}^2$.

```
"pquesencr" Copyright (c) 2005-2006 by
Moti Yung and Adam L. Young. All rights reserved.
This program is based on the notion of "questionable
encryptions" that were introduced in Section 6.6.2 of
the book:
  "Malicious Cryptography: Exposing  Cryptovirology"
  by Adam Young & Moti Yung, Wiley, 2004.
This program is also based on both:
  1) A. Young, M. Yung, "On Fundamental Limitations of
  Proving Data Theft", To appear in IEEE Transactions
  on Information Forensics and Security.

  2) A. Young, M. Yung, "Questionable Encryption and Its
  Applications", 1st International Conference on
  Cryptology in Malaysia---MyCrypt '05, E. Dawson,
  S. Vaudenay (Eds.), LNCS 3715, pp. 210-221, Springer, 2005.

RAND_status() returned 1.
The primes p and q will each be 128 bits in length.
Questionable Encryption functions:
Type (a) to generate a REAL key pair.
        This overwrites "pubkey.txt" and "privkey.txt".
Type (b) to generate a FAKE key pair
        This overwrites "pubkey.txt" and "privkey.txt".
Type (c) to test the witness in the file "privkey.txt"
        using function F.
Type (d) to hybrid "encrypt" the file "plaintext.txt"
        using Paillier and Blowfish.
Type (e) to hybrid  decrypt  the file "ciphertext.bin"
        using Paillier and Blowfish.
Type (f) to conduct 100 iterations of
        encryption-decryption stress testing.
Type (g) to conduct 100 iterations of
        witness of encryption stress testing.
Type (h) to conduct 100 iterations of
        witness of non-encryption stress testing.
```

The commands are relatively self-explanatory. We will cover commands (a) through (e) in this chapter. Commands (f) through (h) are used for software testing and will not be covered in this chapter.

3.1 Working with Real Key Pairs

Command (a) generates a *real* Paillier key pair. In this particular implementation it generates a key pair in which `p` and `q` are both safe primes. This is to ensure that n is hard to factor.

The public key parameters are $(\mathtt{n}, \mathtt{g}, \mathtt{n}^2)$. These are written to the ASCII text file `pubkey.txt` that is created (or overwritten if it already exists). The value $\mathtt{n}^2$ is computed once and for all to speed up the encryption process.

There are 9 private key parameters in total. These are `p`, `q`, $\mathtt{p}^2$, $\mathtt{q}^2$,... and they will be described later. At this point it suffices to say that these additional precomputed values serve to speed up Paillier decryption significantly. The private parameters are written to the ASCII text file `privkey.txt`.

```
Enter command (a-h) : a
Generating a real Paillier key pair
n   = C1114C9F7E95C45CB6F77781E63EEED447DB77EC867119394F83AA7F46
98F79D
g   = 8A1BB19F2AF2720AA500971537A145D2C675D3A6888EFE746B0D9216C0
FE080AA5F53C6489C102CE136D03D55DB6F5969A90E93DFCB4D6848CA0F50617
920557
n^2 = 919B16B3C0FBB87F496A2BC7616DBBFD1C02BFC17E91FFD11E5E90B344
0AFEA3B71E9318095635EC73A9111FFE50A07B17C04524C276E05498C83496D5
F05649
p   = C8ADDC935EF96AEA068DE12E49DC878B
q   = F64A47F4CD2BC3089705292D13F0A277
p^2 = 9D501EBA21A1B43B0FF9AD2712A826AFC26727256652FF9819F08F3A84
ABE579
q^2 = ECF2D7D828106C07493EE41A0A669461CAD821C0D06D90D1D2D6D624B0
3AD351
p^{-1} mod 2^w = 8BA78A5B2DDF415970B2E545907B6823
q^{-1} mod 2^w = A9D8954A4B3B08E663D2A97A19F7D747
h_p = 5ABDE2A8D26DD00B6B4AB23D796863E7
h_q = A147F314412F74C6E6D2E265BED5873C
q^{-1} mod p   = 8DBC22F5FBA522A71D6EEC79B4619EF3
Wrote composite n to file "pubkey.txt".
Wrote g to the file "pubkey.txt".
Wrote n^2 to the file "pubkey.txt".
Wrote primes (p,q) to the file "privkey.txt".
Wrote p^2 and q^2 to the file "privkey.txt".
Wrote (p^-1 mod 2^w,q^-1 mod 2^w) to "privkey.txt".
Wrote h_p and h_q to the file "privkey.txt".
Wrote q^{-1} mod p to the file "privkey.txt".
```

```
--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

The last portion of the output indicates the presence or absence of memory leaks. The code for this utilizes OpenSSL's memory leak checking functionality. As long as there is no output between these two lines, there should not be any memory leaks in the implementation.

To determine the nature of the public key, command (c) is used. This command requires the public and private key files. Command (c) executes function F as defined in [9].

```
Enter command (a-h) : c
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
n   = C1114C9F7E95C45CB6F77781E63EEED447DB77EC867119394F83AA7F46
98F79D
g   = 8A1BB19F2AF2720AA500971537A145D2C675D3A6888EFE746B0D9216C0
FE080AA5F53C6489C102CE136D03D55DB6F5969A90E93DFCB4D6848CA0F50617
920557
n^2 = 919B16B3C0FBB87F496A2BC7616DBBFD1C02BFC17E91FFD11E5E90B344
0AFEA3B71E9318095635EC73A9111FFE50A07B17C04524C276E05498C83496D5
F05649
About to read in private key values from the file "privkey.txt".

Private key values loaded successfully.
p   = C8ADDC935EF96AEA068DE12E49DC878B
q   = F64A47F4CD2BC3089705292D13F0A277
p^2 = 9D501EBA21A1B43B0FF9AD2712A826AFC26727256652FF9819F08F3A84
ABE579
q^2 = ECF2D7D828106C07493EE41A0A669461CAD821C0D06D90D1D2D6D624B0
3AD351
p^{-1} mod 2^w = 8BA78A5B2DDF415970B2E545907B6823
q^{-1} mod 2^w = A9D8954A4B3B08E663D2A97A19F7D747
h_p = 5ABDE2A8D26DD00B6B4AB23D796863E7
h_q = A147F314412F74C6E6D2E265BED5873C
q^{-1} mod p   = 8DBC22F5FBA522A71D6EEC79B4619EF3
This key pair is REAL.

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

The output shown above indicates that the key pair is a real Paillier key pair. This is as it should be since the key pair was created using command (a). The other possible responses are that the key pair is fake or that an error occurred.

Now that the real key pair has been generated and tested, we are in a position to encrypt data with the public key. The program `pquesencr.exe` will encrypt the file `plaintext.txt` provided that it is available (the key pair files, the plaintext, and the ciphertext files should be stored in the same directory as `pquesencr.exe`). This is accomplished by selecting command (d) after the key pair has been stored.

```
Enter command (a-h) : d
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
Looking for file "plaintext.txt" so that it can be hybrid "encry
pted".
File "plaintext.txt" successfully opened for reading.
key[] = 0xAC7E0886CB581DE9647E81839B27953E
 iv[] = 0x896128FE30F6DDDE
m = AB9A170C27A1274EFDD74D62429B4F23E95279B83817E64E91D58CB86087
EAC
r = 2728D60FEA2D4B8CA59B02D3047EA8DD10546042BA2C4B2ECF5A7C93FE27
1A6B
c = 294AA3D8838D4FE1FA97E4735626DFD2D14DE184C491C871F4BB51123D78
8934FB9B461F4B27FB947B0DE6BBDF6DC3AFD871F59B5F68C22BB8305C92A315
6555
Ciphertext written to file "ciphertext.bin".

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

The ciphertext is stored in the data file `ciphertext.bin`. This is a binary file that can be examined using a disk utility or hex editor.

Since Paillier is a probabilistic cryptosystem, the asymmetric ciphertext will differ with each encryption with overwhelming probability. Also, the program chooses a random Blowfish key and initialization vector with each encryption. So, the CBC ciphertext will differ with each encryption of the same plaintext file with overwhelming probability.

Selecting command (e) causes the ciphertext file to be decrypted. The resulting plaintext is written to `plaintext2.txt`.

```

Enter command (a-h) : e
About to read in private key values from the file "privkey.txt".

Private key values loaded successfully.
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
Looking for file "ciphertext.bin" so that it can be hybrid decry
pted.
File "ciphertext.bin" successfully opened for reading.
c = 294AA3D8838D4FE1FA97E4735626DFD2D14DE184C491C871F4BB51123D78
8934FB9B461F4B27FB947B0DE6BBDF6DC3AFD871F59B5F68C22BB8305C92A315
6555
m = AB9A170C27A1274EFDD74D62429B4F23E95279B83817E64E91D58CB86087
EAC
key[] = 0xAC7E0886CB581DE9647E81839B27953E
 iv[] = 0x896128FE30F6DDDE
Plaintext written to file "plaintext2.txt".

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------

```

As long as the public key file and private key file are preserved, the file `plaintext2.txt` will be identical to the original plaintext file. This completes the description of using *real* public keys.

3.2 Working with Fake Key Pairs

Selecting command (b) causes a *fake* Paillier key pair to be generated. When the fake public key is used to asymmetrically encrypt data, the resulting ciphertext cannot be decrypted at all, even when `p` is known.

```

Enter command (a-h) : b
Generating a fake Paillier key pair
n   = EBCC600687235C48A4F278D2567D3899DD90FFC92E1E4CACA74B783DFF
781205
g   = 689C2BB776B0B00E099C31A80D864A857F7250A781A8B26654D23D083B
D97EF624A22FB9E33EEA366956704890153F0CED34FFADB940B750229F44B923
B22285
n^2 = D930DB752A87314D94D12B974B18491D0491CEF6B451C57898CD795D30
220975C52EC0EB8F3BD4A0680F077E54BA1123C68B38972953B51963FBB298DB
```

```
F4B419
p   = F525F7AC4C27C267EBC522460C45CF93
q   = F63C740C3A07550573027DFAD3502707
p^2 = EAC1B1B151300C84408AE0CBB80EF33CF5330BBE947F91409C252EE355
8D0E69
q^2 = ECD83D4E19DC6002767AEEC260BE414E170E9A0E593264B493324919F4
532231
p^{-1} mod 2^w = 740D20069ECA1E32954A4A99BF96A69B
q^{-1} mod 2^w = 5D7D03B72DDABE726848CA7FB2FA96B7
h_p = 0
h_q = 0
q^{-1} mod p   = 788D77A9E81599569541BCB19E1302BB
Wrote composite n to file "pubkey.txt".
Wrote g to the file "pubkey.txt".
Wrote n^2 to the file "pubkey.txt".
Wrote primes (p,q) to the file "privkey.txt".
Wrote p^2 and q^2 to the file "privkey.txt".
Wrote (p^-1 mod 2^w,q^-1 mod 2^w) to "privkey.txt".
Wrote h_p and h_q to the file "privkey.txt".
Wrote q^{-1} mod p to the file "privkey.txt".

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

The 'public key' in `pubkey.txt` can be placed in malware and be used to 'encrypt' data. If push comes to shove the file `privkey.txt` can be presented. The values contained therein will prove that no data is enciphered when the 'public key' is supplied to the Paillier encryption algorithm.

To demonstrate that the public key is fake, command (c) is used. This causes the verification function F to be executed on values taken from `pubkey.txt` and `privkey.txt`.

```

Enter command (a-h) : c
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
n   = EBCC600687235C48A4F278D2567D3899DD90FFC92E1E4CACA74B783DFF
781205
g   = 689C2BB776B0B00E099C31A80D864A857F7250A781A8B26654D23D083B
D97EF624A22FB9E33EEA366956704890153F0CED34FFADB940B750229F44B923
B22285
n^2 = D930DB752A87314D94D12B974B18491D0491CEF6B451C57898CD795D30
220975C52EC0EB8F3BD4A0680F077E54BA1123C68B38972953B51963FBB298DB
```

```
F4B419
About to read in private key values from the file "privkey.txt".

Private key values loaded successfully.
p   = F525F7AC4C27C267EBC522460C45CF93
q   = F63C740C3A07550573027DFAD3502707
p^2 = EAC1B1B151300C84408AE0CBB80EF33CF5330BBE947F91409C252EE355
8D0E69
q^2 = ECD83D4E19DC6002767AEEC260BE414E170E9A0E593264B493324919F4
532231
p^{-1} mod 2^w = 740D20069ECA1E32954A4A99BF96A69B
q^{-1} mod 2^w = 5D7D03B72DDABE726848CA7FB2FA96B7
h_p = 0
h_q = 0
q^{-1} mod p   = 788D77A9E81599569541BCB19E1302BB
This key pair is FAKE.

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

Since a fake key was generated using command (b), as expected the output indicates that the key pair is fake. The implementation checks various values from the key files. However, the actual implementation of F only takes $(\mathtt{p}, \mathtt{n}, \mathtt{g})$ as input.

The fake public key nonetheless produces normal looking asymmetric ciphertexts. Therefore, it leads to normal looking hybrid encryptions. Data is encrypted with the fake public key by executing command (d).

```

Enter command (a-h) : d
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
Looking for file "plaintext.txt" so that it can be hybrid "encry
pted".
File "plaintext.txt" successfully opened for reading.
key[] = 0x4119967DEDAF118A7EED0669259E8126
 iv[] = 0x71461A0DC4A69B8F
m = 5EDE5AA7D01FCC7419883E5DCA57307826819E256906ED7E8A11AFED7D96
1941
r = 4CD222E2E6543FF203561F6C594059459810F82177BB795718EEBD9017F4
0568
c = 74E8672757CFB306EED92E30D2E9EA9E4B1DCDA47CBE0A0AB2785EB99EA4
13354D0D228A20B5CC12F85C4D8ACFC6F0097A4B05873158735FF593DE339DF6
```

```
76DA
Ciphertext written to file "ciphertext.bin".

--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

Not surprisingly, the resulting ciphertext file cannot be decrypted. Any attempt to do so leads to an error.

```
Enter command (a-h) : e
About to read in private key values from the file "privkey.txt".

Private key values loaded successfully.
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
Attempting decryption with a fake key pair.
We will continue, but decryption will fail.
Looking for file "ciphertext.bin" so that it can be hybrid decry
pted.
File "ciphertext.bin" successfully opened for reading.
ERROR: Key pair might be fake.
```

4 The Function main()

The function `main` prints programming credits to the screen. It makes sure that the data type `int` is 4 bytes in length. This is required since an integer is written to the ciphertext file and it must be read in correctly for decryption. OpenSSL is then seeded using the Microsoft Cryptographic API call `CryptGenRandom` that is called within `SeedTheRNG`. The user is prompted to enter a command that is then executed.

```
int main(void)
{
int isReal=0,command;

PrintCredits();
if (sizeof(isReal) != 4)
   TerminateWithError("ERROR: int size is not 4 bytes\n");
EnableLeakChecking();
SeedTheRNG();
for (;;)
   {
   PrintCommandOptions();
   command = getchar();
   if ((command >= 'a') && (command <= 'h'))
      break;
   }
switch(command)
   {
   case 'a':
      isReal = 1;
      GenAndStoreQEPair(PUBKEY_FILE,PRIVKEY_FILE,isReal);
      break;
   case 'b':
      GenAndStoreQEPair(PUBKEY_FILE,PRIVKEY_FILE,isReal);
      break;
   case 'c':
      ExecuteFunctionF(PRIVKEY_FILE,PUBKEY_FILE);
      break;
   case 'd':
      QuesEncrHybridEncrFile(PLAINTEXT_FILE,PUBKEY_FILE,
         CIPHERTEXT_FILE);
      break;
   case 'e':
      QuesEncrHybridDecrFile(CIPHERTEXT_FILE,PRIVKEY_FILE,
         PUBKEY_FILE,PLAINTEXT_FILE_2);
      break;
   case 'f':
      StressTestQE();
      break;
   case 'g':
      StressTestWitOfEnc();
      break;
   case 'h':
      StressTestWitOfNonEnc();
      break;
   }
PrintLeakList();
return 0;
}
```

5 Key Pair Generation

GenAndStoreQEPair() is used to generate real and fake Paillier key pairs. When `isReal` is 0 a fake key pair is generated and when `isReal` is 1 a real key pair is generated. This particular implementation makes p and q safe primes.

The function writes the public and private key values to the public and private key files respectively. It frees the structures for the keys and then returns control to the caller.

```
void GenAndStoreQEPair(const char *pubkeyFlName,
      const char *privkeyFlName,int isReal)
{
paillierprivkey privkey;
paillierpubkey pubkey;

PaillierPrivKeyNew(&privkey);
PaillierPubKeyNew(&pubkey);
if (isReal == 1)
   printf("Generating a real Paillier key pair\n");
if (isReal == 0)
   printf("Generating a fake Paillier key pair\n");
GenerateQuesEncrKeyPair(&privkey,&pubkey,isReal);
PrintPublicKeyStructure(&pubkey);
PrintPrivateKeyStructure(&privkey);
WriteKeyPairToFiles(privkeyFlName,pubkeyFlName,
   &privkey,&pubkey);
PaillierPrivKeyFree(&privkey);
PaillierPubKeyFree(&pubkey);
}
```

The value $\texttt{nsquared} = \texttt{n}^2$ is computed once and for all and is stored in the public key file. This enables fast Paillier encryption since the modulus `n` will not have to be squared at run-time.

```
typedef struct {
BIGNUM *n,*g,*nsquared;
   } paillierpubkey;
```

The private key data structure consists of 9 values. The first two values are the primes `p` and `q` that form the public modulus $\texttt{n} = \texttt{pq}$. The values psquared and qsquared are p^2 and q^2, respectively. The remaining 5 values will be covered in the functions that compute them.

```
typedef struct {
BIGNUM *p,*q,*psquared,*qsquared;
BIGNUM *pinvmod2tow,*qinvmod2tow,*hsubp,*hsubq,*qInv;
   } paillierprivkey;
```

GenerateQuesEncrKeyPair() generates a real or fake key pair randomly. It then populates both the public key data structure and the private key data structure with the appropriate values.

```
void GenerateQuesEncrKeyPair(paillierprivkey *privkey,
      paillierpubkey *pubkey,int isReal)
{
int safe = 1;

BN_CTX *ctx = BN_CTX_new();
BN_generate_prime(privkey->p,kover2,safe,NULL,NULL,
      NULL,NULL);
BN_generate_prime(privkey->q,kover2,safe,NULL,NULL,
      NULL,NULL);
BN_sqr(privkey->psquared,privkey->p,ctx);
BN_sqr(privkey->qsquared,privkey->q,ctx);
BN_mul(pubkey->n,privkey->p,privkey->q,ctx);
BN_sqr(pubkey->nsquared,pubkey->n,ctx);
if (isReal)
   GenerateRealPaillierGenerator(pubkey->g,privkey);
else
   GenerateFakePaillierGenerator(pubkey->g,privkey);
ComputePInvMod2ToWConstant(privkey->pinvmod2tow,privkey->p);
ComputePInvMod2ToWConstant(privkey->qinvmod2tow,privkey->q);
ComputeHConstant(privkey->hsubp,pubkey->g,privkey->p,
   privkey->psquared,privkey->pinvmod2tow,ctx);
ComputeHConstant(privkey->hsubq,pubkey->g,privkey->q,
   privkey->qsquared,privkey->qinvmod2tow,ctx);
ComputeQInv(privkey->qInv,privkey->q,privkey->p,ctx);
BN_CTX_free(ctx);
}
```

GenerateRealPaillierGenerator() generates and returns a real Paillier generator g using values stored in privkey. The generator g is found by generating a random value `gsubp` $\in \mathbb{Z}^*_{p^2}$ where the order of `gsubp` is divisible by `p`. Also, a random value `gsubq` $\in \mathbb{Z}^*_{q^2}$ is generated where the order of `gsubq` is divisible by `q`. The multiplicative inverse of q^2 mod p^2 is computed and stored in `inv`. This is then passed to FastChineseRemaindering() along with `gsubp` and `gsubq` to compute `g`. FastChineseRemaindering() is faster than typical implementations of the Chinese Remainder Theorem. GenerateRealPaillierGenerator() ensures that `n` divides the order of `g` evenly.

```
void GenerateRealPaillierGenerator(BIGNUM *g,
      paillierprivkey *privkey)
{
BIGNUM *tmp,*inv,*pmin1,*qmin1,*gsubp,*gsubq;

BN_CTX *ctx = BN_CTX_new();
tmp = BN_new();inv = BN_new();
pmin1 = BN_new();qmin1 = BN_new();
gsubp = BN_new();gsubq = BN_new();
BN_sub(pmin1,privkey->p,BN_value_one());
BN_sub(qmin1,privkey->q,BN_value_one());
for (;;)
   {
   BN_rand_range(gsubp,(BIGNUM *) privkey->psquared);
   if (IsInZnSquaredstar(gsubp,privkey->psquared,ctx))
      {
      BN_mod_exp(tmp,gsubp,pmin1,privkey->psquared,ctx);
      if (BN_are_not_equal(tmp,BN_value_one()))
         break;
      }
   }
for (;;)
   {
   BN_rand_range(gsubq,(BIGNUM *) privkey->qsquared);
   if (IsInZnSquaredstar(gsubq,privkey->qsquared,ctx))
      {
      BN_mod_exp(tmp,gsubq,qmin1,privkey->qsquared,ctx);
      if (BN_are_not_equal(tmp,BN_value_one()))
         break;
      }
   }
BN_mod(tmp,privkey->qsquared,privkey->psquared,ctx);
BN_mod_inverse(inv,tmp,privkey->psquared,ctx);
FastChineseRemaindering(g,gsubp,privkey->psquared,gsubq,
   privkey->qsquared,inv,ctx);
BN_clear_free(tmp);BN_clear_free(inv);
BN_clear_free(pmin1);BN_clear_free(qmin1);
BN_clear_free(gsubp);BN_clear_free(gsubq);
BN_CTX_free(ctx);
}
```

GenerateFakePaillierGenerator() generates and returns a fake Paillier generator `g` using values stored in `privkey`. The multiplicative inverse of q^2 mod p^2 is computed and stored in `inv`. This is then passed to FastChineseRemaindering() along with `gsubp` and `gsubq` to compute `g`. This process ensures that the order of `g` divides $\lambda(n)$. Here λ is Carmichael's function.

```
void GenerateFakePaillierGenerator(BIGNUM *g,
      paillierprivkey *privkey)
{
BIGNUM *tmp,*inv,*gsubp,*gsubq;

BN_CTX *ctx = BN_CTX_new();
tmp = BN_new();inv = BN_new();
gsubp = BN_new();gsubq = BN_new();
for (;;)
   {
   BN_rand_range(gsubp,(BIGNUM *) privkey->psquared);
   if (IsInZnSquaredstar(gsubp,privkey->psquared,ctx))
      {
      BN_mod_exp(gsubp,gsubp,privkey->p,privkey->psquared,ctx);
      break;
      }
   }
for (;;)
   {
   BN_rand_range(gsubq,(BIGNUM *) privkey->qsquared);
   if (IsInZnSquaredstar(gsubq,privkey->qsquared,ctx))
      {
      BN_mod_exp(gsubq,gsubq,privkey->q,privkey->qsquared,ctx);
      break;
      }
   }
BN_mod(tmp,privkey->qsquared,privkey->psquared,ctx);
BN_mod_inverse(inv,tmp,privkey->psquared,ctx);
FastChineseRemaindering(g,gsubp,privkey->psquared,gsubq,
   privkey->qsquared,inv,ctx);
BN_clear_free(tmp);BN_clear_free(inv);
BN_clear_free(gsubp);BN_clear_free(gsubq);
BN_CTX_free(ctx);
}
```

ComputePInvMod2ToWConstant() computes `pinvmod2tow` to be $p^{-1} \bmod 2^w$ where w is the length of p in bits. Therefore, `pinvmod2tow` in the private key data structure is set to this value. Similarly, `qinvmod2tow` in the private key data structure is set to $q^{-1} \bmod 2^w$ where w is the length of q in bits. The output of this function is used to speed up the computation of the function L that is used in the Paillier cryptosystem. The function L is defined as $\mathrm{L}(u, n) = (u-1)/n$.

```
void ComputePInvMod2ToWConstant(BIGNUM *pinvmod2tow,
      const BIGNUM *p)
{
BN_CTX *ctx = BN_CTX_new();
BIGNUM *twotow = BN_new();
BN_set_word(twotow,1);
BN_lshift(twotow,twotow,BN_num_bits(p));
BN_mod_inverse(pinvmod2tow,p,twotow,ctx);
BN_free(twotow);
BN_CTX_free(ctx);
}
```

ComputeHConstant() computes `hsubp` $= h_p$ as follows.

$$h_p = \mathrm{L}(g^{p-1} \bmod p^2, p)^{-1} \bmod p$$

Therefore, `hsubp` in the private key data structure is set to this value. Similarly, `hsubq` $= h_q$ in the private key data structure is set to be,

$$h_q = \mathrm{L}(g^{q-1} \bmod q^2, q)^{-1} \bmod q$$

The function L is computed using FastFunctionL(). Section 8 covers the way in which h_p and h_q are used.

```
void ComputeHConstant(BIGNUM *hsubp,const BIGNUM *g,
      const BIGNUM *p,const BIGNUM *psquared,
      const BIGNUM *pinvmod2tow,BN_CTX *ctx)
{
```

```
BN_CTX_start(ctx);
BIGNUM *tmp = BN_CTX_get(ctx);
BIGNUM *pmin1 = BN_CTX_get(ctx);
BN_sub(pmin1,p,BN_value_one());
BN_mod(tmp,g,psquared,ctx);
BN_mod_exp(tmp,tmp,pmin1,psquared,ctx);
FastFunctionL(hsubp,tmp,p,pinvmod2tow,ctx);
BN_mod_inverse(hsubp,hsubp,p,ctx);
BN_clear(tmp);BN_clear(pmin1);
BN_CTX_end(ctx);
}
```

FastFunctionL() computes $y = \mathrm{L}(u, n) = (u-1)/n$. However, it must be given `ninvmod2tow` which is the multiplicative inverse of n modulo 2^w. Here w is the length of n in bits. The use of `ninvmod2tow` permits the computation of $\mathrm{L}(u, n)$ without having to do any divisions. The computation is equivalent to using the function LFast in which,

$$\mathrm{LFast}(\mathtt{u}, \mathtt{n}, \mathtt{ninvmod2tow}) = (\mathtt{u} - 1) * \mathtt{ninvmod2tow} \bmod 2^{|n|}$$

BN_mask_bits(a,n) truncates a to an n bit number. For example, the result of the call BN_mask_bits(a,128) is

4379092B38E9EB0DD8984323E0E1CF6

when a is equal to,

1C01D04379092B38E9EB0DD8984323E0E1CF6

```
void FastFunctionL(BIGNUM *y,const BIGNUM *u,const BIGNUM *n,
      const BIGNUM *ninvmod2tow,BN_CTX *ctx)
{
BN_CTX_start(ctx);
BIGNUM *tmp = BN_CTX_get(ctx);
BN_sub(tmp,u,BN_value_one());
int w = BN_num_bits(n);
BN_mask_bits(tmp,w);
BN_mul(y,tmp,ninvmod2tow,ctx);
BN_mask_bits(y,w);
BN_clear(tmp);
BN_CTX_end(ctx);
}
```

Adopting the variable name qInv from PKCS #1, we set qInv = q^{-1} mod p. ComputeQInv() computes and returns gInv. The value qInv is stored in the private key data structure and is used to speed up Paillier decryption.

```
void ComputeQInv(BIGNUM *qInv,const BIGNUM *q,
      const BIGNUM *p,BN_CTX *ctx)
{
BN_mod(qInv,q,p,ctx);
BN_mod_inverse(qInv,qInv,p,ctx);
}
```

6 Verification Function F

ExecuteFunctionF() loads the public and private keys. It then checks each for internal consistency and checks their consistency with one another. If an inconsistency is found then the function terminates prematurely. Otherwise, it determines the nature of the key pair by calling FunctionF() and printing out the result. Finally, it frees the data structures used to store the public and private key values.

```
int ExecuteFunctionF(const char *privkeyFlName,
      const char *pubkeyFlName)
{
paillierpubkey pubkey;
paillierprivkey privkey;

BIGNUM *tmp = BN_new();
BN_CTX *ctx = BN_CTX_new();
PaillierPubKeyNew(&pubkey);
LoadPublicKey(&pubkey,pubkeyFlName);
PrintPublicKeyStructure(&pubkey);
if (PublicKeyIsInconsistent(&pubkey))
   TerminateWithError("Public key file is inconsistent.\n");
PaillierPrivKeyNew(&privkey);
LoadPrivateKey(&privkey,privkeyFlName);
PrintPrivateKeyStructure(&privkey);
if (PrivateKeyIsInconsistent(&privkey,pubkey.g))
   TerminateWithError("Private key file is inconsistent.\n");
BN_mul(tmp,privkey.p,privkey.q,ctx);
if (BN_are_not_equal(pubkey.n,tmp))
```

```
   TerminateWithError("Key files are inconsistent.\n");
int retval = FunctionF(privkey.p,pubkey.n,pubkey.g);
if (retval < 0)
   TerminateWithError("Function F returned a value < 0.\n");
if (retval == 1)
   printf("This key pair is REAL.\n\n");
else
   printf("This key pair is FAKE.\n\n");
PaillierPubKeyFree(&pubkey);
PaillierPrivKeyFree(&privkey);
BN_free(tmp);BN_CTX_free(ctx);
return retval;
}
```

FunctionF() returns 0 for a fake Paillier key. It returns 1 for a real Paillier key. It returns -1 on error. Assuming that the p and q are primes, they are the correct bit length, and so forth, the function proceeds to focus on the value g. Using the private key, the function DetermineMembershipOfG() determines whether g is real or fake.

```
int FunctionF(const BIGNUM *p,const BIGNUM *n,
      const BIGNUM *g)
{
int returnvalue = -1;
BIGNUM *nsquared,*q,*rem;

if (BN_num_bits(p) >= BN_num_bits(n))
   return -1;
BN_CTX *ctx = BN_CTX_new();
nsquared = BN_new();q = BN_new();rem = BN_new();
for (;;)
   {
   BN_div(q,rem,n,p,ctx);
   if (BN_is_not_zero(rem))
      break;
   if (BN_num_bits(p) != kover2)
      break;
   if (BN_num_bits(q) != kover2)
      break;
   if (BN_is_prime(p,160,NULL,ctx,NULL) <= 0)
      break;
   if (BN_is_prime(q,160,NULL,ctx,NULL) <= 0)
      break;
   BN_sqr(nsquared,n,ctx); /* now check g */
   if (!(IsInZnSquaredstar(g,nsquared,ctx)))
      break;
   returnvalue = DetermineMembershipOfG(p,q,n,nsquared,g);
   break;
   }
BN_free(nsquared);
BN_clear_free(q);BN_clear_free(rem);
BN_CTX_free(ctx);
return returnvalue;
}
```

DetermineMembershipOfG() returns 1 if g is a *real* Paillier generator. It returns 0 if g is a *fake* Paillier generator. It returns -1 on error.

If the order of g divides $t = \lambda(n)$ evenly then DetermineMembershipOfG returns 0 (fake g). If the order of g is a non-zero multiple of n then DetermineMembershipOfG returns 1 (real g). As noted in [5], when gcd(L(g^t mod n^2), n) = 1 it follows that the order of g is a non-zero multiple of n.

```
int DetermineMembershipOfG(const BIGNUM *p,
      const BIGNUM *q,const BIGNUM *n,
      const BIGNUM *nsquared,const BIGNUM *g)
{
int returnvalue = -1;
BIGNUM *t,*tmp,*pmin1,*qmin1,*quot,*rem,*gtot;

BN_CTX *ctx = BN_CTX_new();
t = BN_new();tmp = BN_new();pmin1 = BN_new();
qmin1 = BN_new();quot = BN_new();
rem = BN_new();gtot = BN_new();
BN_sub(pmin1,p,BN_value_one());
BN_sub(qmin1,q,BN_value_one());
BN_lcm(t,pmin1,qmin1,ctx);
BN_mod_exp(gtot,g,t,nsquared,ctx);
if (BN_is_one(gtot))
   returnvalue = 0;
else
   {
   BN_sub(tmp,gtot,BN_value_one());
   BN_div(quot,rem,tmp,n,ctx);
   BN_gcd(tmp,quot,n,ctx);
   if (BN_is_one(tmp))
```

```
      returnvalue = 1;
   }
BN_clear_free(t);BN_clear_free(tmp);
BN_clear_free(pmin1);BN_clear_free(qmin1);
BN_clear_free(quot);BN_clear_free(rem);
BN_clear_free(gtot);
BN_CTX_free(ctx);
return returnvalue;
}
```

7 The Encryption Source Code

The function QuesEncrHybridEncrFile() encrypts the plaintext file using Paillier and Blowfish in CBC mode. The format of the output file is,

$$(\texttt{n}, \texttt{c}, \texttt{iv}, \texttt{CBCciphertextlen}, \texttt{CBCciphertext})$$

The ciphertext file is a binary file.

The inclusion of `n` allows the decryption program to make sure it is applying the correct Paillier private key to the ciphertext file. Because of the possibility of leading zeros, the Paillier ciphertext `c` is written out as an array. The value `iv` is the Blowfish CBC initialization vector. `CBCciphertextlen` is the length in bytes of the CBC ciphertext denoted by `CBCciphertext`.

```
void QuesEncrHybridEncrFile(const char *ptextFl,
      const char *pubkeyFlName,const char *ctextFl)
{
unsigned char *CBCptextinput,*CBCciphertext,iv[8];
unsigned char cArray[NSQUARED_SIZE_IN_BYTES];
int CBCinputplaintextlen,CBCctextlen,blocksize = 8;
size_t count = 0,fileLength;
FILE *inFl,*outFl;
paillierpubkey pubkey;

BIGNUM *c = BN_new();
PaillierPubKeyNew(&pubkey);
LoadPublicKey(&pubkey,pubkeyFlName);
LoadFilesForEncryption(ptextFl,ctextFl,&inFl,&outFl);
fileLength = GetFileLength(ptextFl);
if ((CBCptextinput = OPENSSL_malloc(fileLength)) == NULL)
   TerminateWithError("ERROR: OPENSSL_malloc() failed.\n");
do {
   count += fread(&CBCptextinput[count],1,1,inFl);
   if (ferror(inFl))
      TerminateWithError("ERROR: fread() failed.\n");
   } while(!feof(inFl));
fclose(inFl);
CBCciphertext = OPENSSL_malloc(fileLength + blocksize);
if (CBCciphertext == NULL)
   TerminateWithError("ERROR: OPENSSL_malloc() failed.\n");
CBCinputplaintextlen = count;
HybridEncrypt(c,iv,CBCciphertext,&CBCctextlen,
   CBCptextinput,CBCinputplaintextlen,&pubkey);
YY_BN_bn2binarray(c,NSQUARED_SIZE_IN_BYTES,cArray);
WriteBIGNUMtoFile(pubkey.n,outFl);
WriteArrayToFile(cArray,NSQUARED_SIZE_IN_BYTES,outFl);
WriteArrayToFile(iv,8,outFl);
WriteArrayToFile((unsigned char *) &CBCctextlen,4,outFl);
WriteArrayToFile(CBCciphertext,CBCctextlen,outFl);
printf("Ciphertext written to file \"%s\".\n\n",ctextFl);
OPENSSL_free(CBCptextinput);OPENSSL_free(CBCciphertext);
BN_free(c);fclose(outFl);
PaillierPubKeyFree(&pubkey);
}
```

The function HybridEncrypt() uses the Paillier public key to hybrid encrypt the plaintext data that `CBCplaintextinput` points to. This function generates a random 128-bit Blowfish key and an initialization vector on the fly. This is used to encrypt the plaintext file. The Blowfish key is encrypted using PaillierEncrypt(). HybridEncrypt() returns -1 on error. It returns 0 when no error is detected.

A full-sized random plaintext is used instead of padding the plaintext out with constant bits. Paillier is semantically secure against known plaintext attacks so this is not necessary to protect against known plaintext attacks.

```
int HybridEncrypt(BIGNUM *c,unsigned char *iv,
      unsigned char *CBCciphertext,int *CBCciphertextlen,
      const char *CBCplaintextinput,int CBCinputptextlen,
      const paillierpubkey *pubkey)
{
unsigned char *ptr,key[16],ptextArray[MODULUS_SIZE_IN_BYTES];
int returnvalue = 0,tmplen,length;
EVP_CIPHER_CTX ctx;

BIGNUM *ptext = BN_new();
BN_rand_range(ptext,(BIGNUM *) pubkey->n);
YY_BN_bn2binarray(ptext,MODULUS_SIZE_IN_BYTES,ptextArray);
ptr = ptextArray + (MODULUS_SIZE_IN_BYTES - 16);
memcpy(key,ptr,16);
RAND_bytes(iv,8);
PrintSymmetricEncryptionValues(key,iv);
PaillierEncrypt(c,ptext,pubkey);
EVP_CIPHER_CTX_init(&ctx); /* Encrypt using blowfish */
length = EVP_CIPHER_key_length(EVP_bf_cbc());
if (length != 16) {printf("ERROR: bad length.\n");exit(1);}
EVP_EncryptInit_ex(&ctx,EVP_bf_cbc(),NULL,key,iv);
if (!EVP_EncryptUpdate(&ctx,CBCciphertext,CBCciphertextlen,
      CBCplaintextinput,CBCinputptextlen))
   returnvalue = -1;
else
   {
   if (!EVP_EncryptFinal_ex(&ctx,
         CBCciphertext + *CBCciphertextlen,&tmplen))
      returnvalue = -1;
   else
      {
      *CBCciphertextlen += tmplen;
      EVP_CIPHER_CTX_cleanup(&ctx);
      }
   }
BN_clear_free(ptext);
return returnvalue;
}
```

We chose to adhere to the Paillier cryptosystem as originally defined [5] and allow $\gcd(\mathtt{m},\mathtt{n}) \neq 1$. However, we depart from the original algorithm slightly that says to choose $\mathtt{r}$ randomly subject to the constraint that $\mathtt{r} < \mathtt{n}$. Instead we choose $\mathtt{r}$ uniformly at random from $\mathbb{Z}_n^*$. Since Paillier encryption is the fundamental building block in computing perfect questionable encryptions, it is worth taking a close look at it's efficiency.

To speed up encryption we exploit the fact that $\mathtt{n}^2$ has already been precomputed and stored in the public key file. So, there is no need to square $\mathtt{n}$ during encryption.

The OpenSSL function `BN_mul()` implements Karatsuba-Ofman multiplication [2]. This forms the basis for multiplying two integers together efficiently in OpenSSL.

The three OpenSSL big number functions `BN_mod_mul_montgomery()`, `BN_from_montgomery()`, and `BN_to_montgomery()` implement Montgomery multiplication [4]. They are invoked automatically when `BN_mod_exp()` is called with suitable inputs, but they may also be useful when several operations need to be performed using the same modulus.

```
void PaillierEncrypt(BIGNUM *c,const BIGNUM *m,
      const paillierpubkey *pubkey)
{
BIGNUM *tmp,*tmp2,*r;

BN_CTX *ctx = BN_CTX_new();
tmp = BN_new();tmp2 = BN_new();r = BN_new();
BN_mod_exp(tmp,pubkey->g,m,pubkey->nsquared,ctx);
for (;;)
   {
   BN_rand_range(r,(BIGNUM *) pubkey->n);
   if (IsInZnstar(r,pubkey->n,ctx))
      break;
   }
BN_mod_exp(tmp2,r,pubkey->n,pubkey->nsquared,ctx);
BN_mod_mul(c,tmp,tmp2,pubkey->nsquared,ctx);
PrintEncryptionValues(m,r,c);
BN_clear_free(tmp);BN_clear_free(tmp2);BN_clear_free(r);
BN_CTX_free(ctx);
}
```

There are other optimizations that are possible for encryption that we chose not to implement. A fixed-base exponentiation algorithm can be used since the base $\mathtt{g}$ is fixed. It is possible to store the array $(\mathtt{g}, \mathtt{g}^2, \mathtt{g}^4, ...)$ as part of the public key (each value here is modulo $\mathtt{n}^2$). These values enable $\mathtt{g}^\mathtt{m} \bmod \mathtt{n}^2$ to be computed using multiplication only as opposed to square-and-multiply. For variants of fixed-base exponentiation, see [1].

Also, $\mathtt{r}^\mathtt{n} \bmod \mathtt{n}^2$ can be computed using a fixed-exponent exponentiation algorithm since $\mathtt{n}$ is fixed. This can be done using a vector addition chain that attempts to minimize the number of multiplications required for exponentiation. Knuth provides a comprehensive overview of vector addition chains [3].

8 The Decryption Source Code

QuesEncrHybridDecrFile() reads in the public and private key values as well as the ciphertext file. It compares the modulus n in the ciphertext file with the modulus in the public key. If they do not match then the function halts with an error. The function attempts to decrypt the ciphertext by calling HybridDecrypt(). When it succeeds it writes the resulting plaintext out to `plaintext2.txt`.

```
void QuesEncrHybridDecrFile(const char *ctextFl,
      const char *privkeyFlName,const char *pubkeyFlName,
      const char *ptextFl)
{
unsigned char *CBCciphertext,*CBCptextoutput,iv[8];
unsigned char cArray[NSQUARED_SIZE_IN_BYTES];
int isReal,CBCctextlen,CBCoutputptextlen;
BIGNUM *c,*n2;
FILE *inFl,*outFl;
paillierpubkey pubkey;
paillierprivkey privkey;

BN_CTX *ctx = BN_CTX_new();
c=BN_new();n2 = BN_new();
PaillierPubKeyNew(&pubkey);
PaillierPrivKeyNew(&privkey);
LoadPrivateKey(&privkey,privkeyFlName);
LoadPublicKey(&pubkey,pubkeyFlName); /* get g */
if ((isReal = FunctionF(privkey.p,pubkey.n,pubkey.g)) == 0)
   {
   printf("Attempting decryption with a fake key pair.\n");
   printf("We will continue, but decryption will fail.\n");
   }
if (isReal < 0)
   printf("key pair is neither \"real\" nor \"fake\".\n");
LoadFilesForDecryption(ctextFl,ptextFl,&inFl,&outFl);
ReadBIGNUMfromFile(inFl,MODULUS_SIZE_IN_BYTES,n2);
if (BN_are_not_equal(pubkey.n,n2))
   TerminateWithError("moduli do not match.\n");
ReadArrayFromFile(inFl,NSQUARED_SIZE_IN_BYTES,cArray);
BN_bin2bn(cArray,NSQUARED_SIZE_IN_BYTES,c);
ReadArrayFromFile(inFl,8,iv);
ReadArrayFromFile(inFl,4,(unsigned char *) &CBCctextlen);
if ((CBCciphertext = OPENSSL_malloc(CBCctextlen)) == NULL)
```

```
   TerminateWithError("ERROR: OPENSSL_malloc() failed.\n");
ReadArrayFromFile(inFl,CBCctextlen,CBCciphertext);
fclose(inFl);
if ((CBCptextoutput = OPENSSL_malloc(CBCctextlen)) == NULL)
   TerminateWithError("ERROR: OPENSSL_malloc() failed.\n");
HybridDecrypt(CBCptextoutput,&CBCoutputptextlen,c,
      CBCciphertext,&CBCctextlen,iv,&privkey);
WriteArrayToFile(CBCptextoutput,CBCoutputptextlen,outFl);
fclose(outFl);
printf("Plaintext written to file \"%s\".\n\n",ptextFl);
OPENSSL_free(CBCciphertext);OPENSSL_free(CBCptextoutput);
PaillierPubKeyFree(&pubkey);
PaillierPrivKeyFree(&privkey);
BN_free(c);BN_free(n2);
BN_CTX_free(ctx);
}
```

The function HybridDecrypt() attempts to decrypt the Paillier ciphertext `c` using PaillierDecrypt() to recover the Blowfish key. It then attempts to decrypt the cipher block chaining ciphertext that `CBCctext` points to using the initialization vector that was recovered from the ciphertext file along with the Blowfish key. This function returns -1 on error. It returns 0 when no error is detected.

```
int HybridDecrypt(unsigned char *CBCptextoutput,
      int *CBCoutputptextlen,const BIGNUM *c,
      unsigned char *CBCctext,int *CBCctextlen,
      unsigned char *iv,const paillierprivkey *privkey)
{
unsigned char key[16],ptextArray[MODULUS_SIZE_IN_BYTES];
int returnvalue = 0,declen,tmplen;
EVP_CIPHER_CTX ctx;

BIGNUM *ptext = BN_new();
PaillierDecrypt(ptext,c,privkey);
YY_BN_bn2binarray(ptext,MODULUS_SIZE_IN_BYTES,ptextArray);
unsigned char *ptr = ptextArray + (MODULUS_SIZE_IN_BYTES-16);
memcpy(key,ptr,16);
PrintSymmetricEncryptionValues(key,iv);
EVP_CIPHER_CTX_init(&ctx);
EVP_DecryptInit_ex(&ctx,EVP_bf_cbc(),NULL,key,iv);
if (!EVP_DecryptUpdate(&ctx,CBCptextoutput,&declen,CBCctext,
      *CBCctextlen))
   returnvalue = -1;
else
   {
   if (!EVP_DecryptFinal_ex(&ctx,CBCptextoutput + declen,
            &tmplen))
      returnvalue = -1;
   else
      *CBCoutputptextlen = declen + tmplen;
   }
BN_clear_free(ptext);
EVP_CIPHER_CTX_cleanup(&ctx);
return returnvalue;
}
```

The Paillier decryption algorithm that follows is based on the efficiency improvements presented in Section 7 of [5]. However, we digress from Section 7 by using the fast Chinese Remainder Theorem (CRT) implementation in RSA's PKCS #1 standard [7] (that uses `gInv`). Decryption utilizes 0 calls to the Extended Euclidean Algorithm instead of 2 calls. Quisquater and Couvreur pointed out the benefit using the CRT in RSA [6]. This idea applies to Paillier as well.

1. compute $m_p = \text{LFast}(c^{p-1} \bmod p^2, p,$ `pinvmod2tow`$)\ h_p \bmod p$
2. compute $m_q = \text{LFast}(c^{q-1} \bmod q^2, q,$ `qinvmod2tow`$)\ h_q \bmod q$
3. compute $h = (m_p - m_q)$`qInv` $\bmod p$
4. output $m = m_q + qh$ and halt

LFast is defined in Section 5 in the discussion of `FastFunctionL()`. The values m_p and m_q correspond to `msubp` and `msubq`, respectively. Similarly, h_p and h_q correspond to `hsubp` and `hsubq`, respectively.

```
void PaillierDecrypt(BIGNUM *m,const BIGNUM *c,
      const paillierprivkey *privkey)
{
BIGNUM *tmp,*pmin1,*qmin1,*msubp,*msubq;

BN_CTX *ctx = BN_CTX_new();
tmp = BN_new();pmin1 = BN_new();qmin1 = BN_new();
msubp = BN_new();msubq = BN_new();
BN_sub(pmin1,privkey->p,BN_value_one());
BN_sub(qmin1,privkey->q,BN_value_one());
BN_mod(tmp,c,privkey->psquared,ctx);
BN_mod_exp(tmp,tmp,pmin1,privkey->psquared,ctx);
FastFunctionL(tmp,tmp,privkey->p,privkey->pinvmod2tow,ctx);
if (BN_is_zero(tmp))
   TerminateWithError("ERROR: Key pair might be fake.\n");
BN_mod_mul(msubp,tmp,privkey->hsubp,privkey->p,ctx);
BN_mod(tmp,c,privkey->qsquared,ctx);
BN_mod_exp(tmp,tmp,qmin1,privkey->qsquared,ctx);
FastFunctionL(tmp,tmp,privkey->q,privkey->qinvmod2tow,ctx);
BN_mod_mul(msubq,tmp,privkey->hsubq,privkey->q,ctx);
FastChineseRemaindering(m,msubp,privkey->p,msubq,privkey->q,
   privkey->qInv,ctx);
PrintDecryptionValues(c,m);
BN_clear_free(tmp);BN_clear_free(pmin1);BN_clear_free(qmin1);
BN_clear_free(msubp);BN_clear_free(msubq);
BN_CTX_free(ctx);
}
```

References

[1] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In R. A. Rueppel, editor, *Advances in Cryptology—Eurocrypt '92*, pages 200–207, New York, 1992. Springer-Verlag.

[2] A. A. Karatsuba and Yu. P. Ofman. Multiplication of multidigit numbers by automata. *Physics Doklady*, 7:595–596, 1963. Translated from *Doklady Akad. Nauk*, vol. 145, no. 2, pages 293–294, 1962.

[3] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley Professional, 1969. Third edition, Nov. 4, 1997.

[4] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[5] Pascal Paillier. Public-key cryptosystems based on composite degree residue classes. In Jacques Stern, editor, *Advances in Cryptology—Eurocrypt '99*, pages 223–238. Springer-Verlag, 1999. Lecture Notes in Computer Science No. 1592.

[6] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, 1982.

[7] RSA Data Security, Inc. *PKCS #1: RSA Cryptography Standard, Version 2.1*, June 2002.

[8] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In Ross Anderson, editor, *Proceedings of the Fast Software Encryption Workshop*, pages 191–204. Springer-Verlag, December 1993. Lecture Notes in Computer Science No. 809.

[9] Adam L. Young and Moti M. Yung. On fundamental limitations of proving data theft. To Appear in *IEEE Transactions on Information Forensics and Security.*

[10] Adam L. Young and Moti M. Yung. *Malicious Cryptography: Exposing Cryptovirology.* Wiley, February 2004.

[11] Adam L. Young and Moti M. Yung. Questionable encryption and its applications. In *First International Conference on Cryptology in Malaysia—MyCrypt*, pages 210–221. Springer, 2005. Lecture Notes in Computer Science No. 3715.

[12] Adam L. Young and Moti M. Yung. Hiding information hiding. In *8th Information Hiding—IH.* Springer, 2006. *to appear.*