Chapter 6:
A Cryptocounter Based on the Paillier PKCS*

Adam L. Young and Moti M. Yung

Abstract

In this chapter we present our implementation of a *cryptocounter* that is based on the Paillier public key cryptosystem [4]. This material and the corresponding appendix (containing the corresponding source code) constitutes *Fundamental Research* on the notion of a cryptocounter. Informally, a cryptocounter [3, 7] is an asymmetric ciphertext of a plaintext counter that satisfies certain properties. It is produced and incremented using the public key of a key pair owner. It can only be deciphered using the corresponding private key. Confidentiality of a counter holds under the assumed intractability of deciding n^{th} degree composite residuosity. A cryptocounter satisfies the following properties: (1) the increment operation increments the underlying plaintext counter without first decrypting the cryptocounter, (2) the probabilistic re-encryption operation "re-encrypts" the underlying plaintext counter without changing it, and (3) it is intractable to correlate cryptocounters when they are updated in a black-box using the two aforementioned operations. These properties make a cryptocounter highly robust against reverse-engineering. The adversary can only learn the change in counter value by observing it being incremented. There are many possible applications of a cryptocounter. It can be used for digital rights management, to gather statistics in a secure fashion in the *honest-but-curious* threat model. It can also be used in a cryptotrojan that gathers statistics on a victim (e.g., how often the victim visits a particular website).

*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

Protecting software against reverse-engineering is a very active area of research today. There are many applications for such technologies, ranging from digital rights management (DRM) to protecting malware from being thoroughly understood. This chapter presents an implementation of a cryptocounter, which in layman's terms can be described as an "obfuscated" integer variable that can be incremented and decremented in software without being revealed and without first decrypting it before incrementing or decrementing.

How is this possible? Modern notions in public key cryptography make this possible. Katz et al described the notion of a cryptographic counter in [3] and presented an instantiation based on the presumed difficulty of distinguishing pseudosquares modulo n from quadratic residues modulo n.[1] Here n is the product of two primes p and q. This is known as the *decision composite residuosity* problem. Other methods were described in Chapter 5 of [7]. The chapter is titled "Cryptocounters," and it deals exclusively with the subject of cryptocounting. It presents a simple cryptocounter based on the Paillier public key cryptosystem. We utilize this construction in this chapter. However, we implement the counter using certain efficiency improvements that were not covered in [7].

The beneficial and clandestine applications of cryptocounters are numerous. The following is a beneficial application. Suppose that a company wants to monitor how often their employees download the latest antivirus updates.[2] One solution is to keep a log file of such activity on the computer of each employee. The aggregation of this information in plaintext could be a security vulnerability and/or a violation of privacy. So, it should be encrypted. But asymmetrically encrypting this data leads to a ciphertext log file that will grow monotonically in size over time.

A solution is as follows. Fifty-two cryptocounters are stored in the computer of an employee. There is a cryptocounter for each week of the year. Every time the employee downloads the antivirus updates, the cryptocounter for that week is incremented by one. When the year is over, the sum of the counters is divided by 52 to reveal the average number of downloads per week.

To help diminish the possibility of correlations, the re-encryption opera-

[1] A pseudosquare mod n is an integer a such that $\mathrm{L}(a/p) = \mathrm{L}(a/q) = -1$. Here $\mathrm{L}(a/r)$ denotes the Legendre symbol of a with respect to the prime r.

[2] Perhaps they don't want to "push" them onto the machines of the employees, but need to ensure that the updates are installed eventually.

tion is applied regularly to each cryptocounter. For example, every 5 minutes each cryptocounter can be re-encrypted automatically without changing the counter value (using the probabilistic re-encryption operation). This helps ensure that the appearance of all the cryptocounters changes if they are captured in a core dump, *not just* the cryptocounters that have been incremented. So, the solution stores the statistics in a confidential fashion (even with respect to the employee) using a fixed amount of space.

A malicious application is a cryptotrojan that spies on the user. For example, a cryptotrojan can contain 7 cryptocounters and store how often a user goes to `www.cryptovirology.com` each day of the week. Over time the sum for each day (Monday, Tuesday, etc.) will grow. By dividing the sum for Monday, say, by the total number of elapsed weeks, the average number of times that the user goes to `cryptovirology.com` on Mondays will be found. This is how cryptovirology can be used to develop cryptographically secure spyware (i.e., cryptotrojans).

Another application is the use of a single cryptocounter to store the propagation statistics of a cryptoworm. A malware designer creates a cryptoworm by placing his or her own public key within the worm. A cryptocounter is created using the public key and the cryptocounter is stored in the worm. The counter is initially zero.

Suppose that the cryptoworm deletes the old copy of itself before moving on to the next node in the network. So, there is only 1 copy of the worm in the network at any given time. Every time the worm enters a node on the network, whether the node has been visited before or not, the cryptoworm increments the cryptocounter by 1. A core-dump of the cryptoworm will reveal the cryptocounter. The malware designer decrypts the cryptocounter using his or her own private decryption key to reveal the raw number of nodes traversed.

Related Work: The notion of a cryptocounter is related to the more general notion of *cryptocomputing*. Readers interested in this subject may wish to read [5, 1, 6, 2]. There are other papers that relate to this subject as well.

2 Creating the Program

The program consists of the ANSI C source file `ccounter.c`. We constructed a simple makefile for this file. The program was compiled using `gcc` and it utilizes the OpenSSL cryptographic library. We used the Minimalist GNU for Windows development suite (MinGW). Compiling this program results in the MS DOS command-line program `ccounter.exe`.

3 Running the Program

For illustrative purposes, 128-bit primes `p` and `q` are used in the examples we give.[3] This is implemented by setting the #`define` **kover2** in `ccounter.c` 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.

```
"ccounter" Copyright (c) 2005-2006 by
Moti Yung and Adam L. Young. All rights reserved.
This program implements a cryptocounter.
For information on the use of cryptocounters in malware
see Chapter 5 entitled "Cryptocounters" in the book:
  "Malicious Cryptography: Exposing  Cryptovirology"
  by Adam Young & Moti Yung, Wiley, 2004.
See also:
  J. Katz, S. Myers, R. Ostrovsky, "Cryptographic
  Counters and Applications to Electronic Voting,"
  Proc. of Eurocrypt, 2001, Springer-Verlag,
  pp. 78-92, 2001.

RAND_status() returned 1.
The primes p and q will each be 128 bits in length.
Cryptocounter functions:
Type (a) to generate a Paillier key pair.
   This overwrites "pubkey.txt" and "privkey.txt".
Type (b) to create a new cryptocounter
   (and delete old cryptocounter).
Type (c) to probabilistically re-encrypt
   the cryptocounter value without changing the
   underlying plaintext counter (and without
   first decrypting the cryptocounter).
Type (d) to increment the counter by 1 (this
   also causes a probabilistic re-encryption).
Type (e) to decrypt the cryptocounter and
   display the plaintext counter value.

Enter command (a-e) :
```

The commands are relatively self-explanatory. We will cover commands (a) through (e) in this chapter.

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

Command (a) generates a Paillier private key and corresponding public key. The public key parameters are the three values $(\mathtt{n}, \mathtt{g}, \mathtt{n}^2)$. 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 $\mathtt{p}$, $\mathtt{q}$, $\mathtt{p}^2$, $\mathtt{q}^2$,... See Chapter 4 for more information on how these are used. At this point it suffices to say that whereas only $\mathtt{p}$ and $\mathtt{q}$ are required for decryption, 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-e) : a
n   = CDC04AB27C6194F0AB02C9D33392606B8FE2F8A8E39BFE35FA7B5D5E9A
BEF64B
g   = 134C50A82CD7977278221C2F9368BCA74BD3A213577351BDC72F3A262D
4EB3FB8D0E05B96AE8DB22EA89CED96F659BFC71BE2704CCE27540BB7E5C767A
89FDF3
n^2 = A55D8811FCBA8742791FEA71C4A8011F19EBD715ED6870D5DA9619E360
EB57A0D58CC5081C4A2C540437B2E342A5CC690EB03085E5D45AFDE8CBABD8C0
4839F9
p   = D9E0EAD675AF751420EF473AA51768E9
q   = F1C02B64AD3D8E8490553B29C9C4A513
p^2 = B96F13BB623E2A56BC89ACA03DF00A9C467DE2F856B30F5A5851F6F45E
DC2411
q^2 = E44B61F4AC8CBC75D9CD1AB736A0778B771AA587ED62CFB5E2B8045B05
897F69
p^{-1} mod 2^w = 5A762CE570340508BAB89F831DB3EF59
q^{-1} mod 2^w = 9899EB5EA602FEC464A5814CC4D0ED1B
h_p = 886D85DABF25D412B9867CE28A2E0ED
h_q = 7C8E59B3D3844ABCFD2A1F841F579DD8
q^{-1} mod p   = 25E5280FA0198C47AB6C9A33259E8AF7
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.

Command (b) generates a new cryptocounter. The value of the plaintext counter will be zero. The cryptocounter will be written to the file `counter.txt`. If this file already exists and is not read-only then it will be overwritten with a new cryptocounter. This text file can be opened and inspected in a text editor. It is instrumental to watch the bits of the cryptocounter change after the re-encryption and increment operations.

```
Enter command (a-e) : b
creating new counter (deleting old)
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
n   = CDC04AB27C6194F0AB02C9D33392606B8FE2F8A8E39BFE35FA7B5D5E9A
BEF64B
g   = 134C50A82CD7977278221C2F9368BCA74BD3A213577351BDC72F3A262D
4EB3FB8D0E05B96AE8DB22EA89CED96F659BFC71BE2704CCE27540BB7E5C767A
89FDF3
n^2 = A55D8811FCBA8742791FEA71C4A8011F19EBD715ED6870D5DA9619E360
EB57A0D58CC5081C4A2C540437B2E342A5CC690EB03085E5D45AFDE8CBABD8C0
4839F9
plaintext counter m = 0
cryptocounter c = 83284A31C9129A18ED7983BAC937F0557B98EF18AC2031
44BE4DD2F5D6F94FCE250A0B4907C74AFDB6851A0266DE9FF1277C84461764AD
03309D3C195CC4F2DD
Wrote cryptocounter to file "counter.txt".
--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

The probabilistic re-encryption operation completely randomizes the ciphertext of the counter. By executing command (c) the cryptocounter is probabilistically re-encrypted and saved to `counter.txt`.

```
Enter command (a-e) : c
probabilistically re-encrypting
```

```
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
n   = CDC04AB27C6194F0AB02C9D33392606B8FE2F8A8E39BFE35FA7B5D5E9A
BEF64B
g   = 134C50A82CD7977278221C2F9368BCA74BD3A213577351BDC72F3A262D
4EB3FB8D0E05B96AE8DB22EA89CED96F659BFC71BE2704CCE27540BB7E5C767A
89FDF3
n^2 = A55D8811FCBA8742791FEA71C4A8011F19EBD715ED6870D5DA9619E360
EB57A0D58CC5081C4A2C540437B2E342A5CC690EB03085E5D45AFDE8CBABD8C0
4839F9
cryptocounter c = 83284A31C9129A18ED7983BAC937F0557B98EF18AC2031
44BE4DD2F5D6F94FCE250A0B4907C74AFDB6851A0266DE9FF1277C84461764AD
03309D3C195CC4F2DD
re-encrypted c = 6B6E28BA32174612158D0C17F074402D9B875E4D3D7C979
D6D396BB2B512E632979E64F8B940D3A3E49D337802C1E009B659B5BCD0A9F80
7FBA44B91E739F1EE
Wrote cryptocounter to file "counter.txt".
--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

Command (d) is used to increment the counter by 1. This operation also completely re-randomizes the cryptocounter ciphertext.

The implementation fixes the cryptocounter increment value to 1. Paillier is an additive-homomorphic encryption scheme. So, this implementation can easily be generalized to do addition modulo n with any non-negative integer less than n (this permits an easy decrement operation, for example).

```
Enter command (a-e) : d
incrementing counter
About to read in public key values from file "pubkey.txt".
Public key values loaded successfully.
n   = CDC04AB27C6194F0AB02C9D33392606B8FE2F8A8E39BFE35FA7B5D5E9A
BEF64B
g   = 134C50A82CD7977278221C2F9368BCA74BD3A213577351BDC72F3A262D
4EB3FB8D0E05B96AE8DB22EA89CED96F659BFC71BE2704CCE27540BB7E5C767A
89FDF3
n^2 = A55D8811FCBA8742791FEA71C4A8011F19EBD715ED6870D5DA9619E360
EB57A0D58CC5081C4A2C540437B2E342A5CC690EB03085E5D45AFDE8CBABD8C0
4839F9
cryptocounter c = 6B6E28BA32174612158D0C17F074402D9B875E4D3D7C97
9D6D396BB2B512E632979E64F8B940D3A3E49D337802C1E009B659B5BCD0A9F8
07FBA44B91E739F1EE
```

```
incremented c = 8B77A381D215AD8307EFE6AB6B2BEE903784871C5BCB4BF7
FFF2B4679C766E3DE5A3A67771818CB7D745D1FD3FB5DEAC402F8CDB9F75243E
A5A1301C014C1D85
Wrote cryptocounter to file "counter.txt".
--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

Finally, command (e) is used to decrypt the cryptocounter value. The private key in `privkey.txt` is needed to do so. Since we incremented the counter exactly once beyond zero, the counter value is 1.

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

Private key values loaded successfully.
p   = D9E0EAD675AF751420EF473AA51768E9
q   = F1C02B64AD3D8E8490553B29C9C4A513
p^2 = B96F13BB623E2A56BC89ACA03DF00A9C467DE2F856B30F5A5851F6F45E
DC2411
q^2 = E44B61F4AC8CBC75D9CD1AB736A0778B771AA587ED62CFB5E2B8045B05
897F69
p^{-1} mod 2^w = 5A762CE570340508BAB89F831DB3EF59
q^{-1} mod 2^w = 9899EB5EA602FEC464A5814CC4D0ED1B
h_p = 886D85DABF25D412B9867CE28A2E0ED
h_q = 7C8E59B3D3844ABCFD2A1F841F579DD8
q^{-1} mod p   = 25E5280FA0198C47AB6C9A33259E8AF7
cryptocounter c = 8B77A381D215AD8307EFE6AB6B2BEE903784871C5BCB4B
F7FFF2B4679C766E3DE5A3A67771818CB7D745D1FD3FB5DEAC402F8CDB9F7524
3EA5A1301C014C1D85
plaintext counter m = 1
--------Memory Leaks displayed below--------
--------Memory Leaks displayed above--------
```

It is instructive to play with the counter by repeatedly incrementing it, decrypting it, re-encrypting it, and so on.

4 Review of Paillier

The value n in Paillier is the product of two large primes p and q. The cryptosystem utilizes a number g that has order $v \bmod n^2$ with v satisfying $v \equiv 0 \bmod n$. The value g is said to have *order* $v \bmod n^2$ if and only if v is the smallest positive integer satisfying $g^v \equiv 1 \bmod n^2$.

The public key is the pair (n, g) and the private key is $\lambda(n)$. Here λ is Carmichael's function. To encrypt $m < n$ a value r is chosen uniformly at random[4] from $\mathbb{Z}_n^*$ and the ciphertext c is computed to be $c = g^m r^n \bmod n^2$. The function $\mathrm{L}(x, n) = \frac{x-1}{n}$ is used for decryption. The following shows how to decrypt c to obtain the plaintext m.

$$m = \mathrm{L}(c^{\lambda(n)} \bmod n^2, n)\ \mathrm{L}(g^{\lambda(n)} \bmod n^2, n)^{-1} \bmod n$$

Fast methods for performing Paillier encryption and decryption are given in [4] and also Chapter 4 of this book.

5 The Cryptocounter Implementation

The plaintext m in Paillier is the plaintext counter value. So, computing a new cryptocounter simply amounts to encrypting $m = 0$. The ciphertext c of $m = 0$ is the cryptocounter. The plaintext counter is recovered by decrypting c and recovering the counter m.

To probabilistically re-encrypt c, a new value r is chosen uniformly at random from $\mathbb{Z}_n^*$. The new ciphertext is c' where $c' = cr^n \bmod n^2$. This operation is performed by `PaillierReEncrypt`.

```
void PaillierReEncrypt(BIGNUM *c, const paillierpubkey *pubkey)
{
BN_CTX *ctx = BN_CTX_new();
BIGNUM *tmp = BN_new();
BIGNUM *r = BN_new();
for (;;)
   {
   BN_rand_range(r,(BIGNUM *) pubkey->n);
   if (IsInZnstar(r,pubkey->n,ctx))
      break;
   }
```

[4] In [4], Section 4 has r chosen randomly from $\mathbb{Z}_n$ and Section 3 has r chosen randomly from $\mathbb{Z}_n^*$. The difference is practically insignificant.

```
BN_mod_exp(tmp,r,pubkey->n,pubkey->nsquared,ctx);
BN_mod_mul(c,c,tmp,pubkey->nsquared,ctx);
BN_clear_free(tmp);BN_clear_free(r);
BN_CTX_free(ctx);
}
```

To increment the counter by 1, a new value r is chosen uniformly at random from $\mathbb{Z}_n^*$. The new ciphertext is c' where $c' = cgr^n \bmod n^2$. The exponent of g is 1 and this represents the increment value. The new ciphertext c' contains the plaintext $m' = m + 1 \bmod n$. The increment operation is implemented in `PaillierIncrement`.

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

The rest of the cryptocounter code borrows heavily from the code for Chapter 4 of this book (perfect questionable encryptions).

6 Conclusion

We presented a simple cryptocounter based on Paillier. There are other ways to implement cryptocounters. However, this Paillier-based counter is quite desirable since the size of the plaintext counter is large relative to the size of the ciphertext that encrypts it (as compared to an implementation based on the decision composite residuosity problem). We encourage readers to send feedback and bug reports to `feedback@cryptovirology.com`.

References

[1] Joan Feigenbaum and Michael Merritt. Open questions, talk abstracts, and summary of discussions. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 2, pages 1–45. AMS, 1991.

[2] Pierre-Alain Fouque, Jacques Stern, and Geert-Jan Wackers. Cryptocomputing with rationals. In *Proceedings of the Sixth International Financial Cryptography Conference*. Springer-Verlag, March 11–14 2003.

[3] Jonathan Katz, Steven Myers, and Rafail Ostrovsky. Cryptographic counters and applications to electronic voting. In Birgit Pfitzmann, editor, *Advances in Cryptology—Eurocrypt '01*, pages 78–92. Springer-Verlag, 2001. Lecture Notes in Computer Science No. 2045.

[4] 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.

[5] Ronald L. Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 169–180. Academic Press, 1978.

[6] Tomas Sander, Adam L. Young, and Moti M. Yung. Non-interactive cryptocomputing for NC^1. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 554–567. IEEE, October 17–19, 1999.

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