Understanding and Implementing Encryption Backdoors

Introduction

The promise of privacy that comes with modern encryption techniques is, to a great extent, what enables broad use of the Internet today. Common online practices like shopping, banking, remote terminal access, etc would be largely impossible were it not for this promise. Foregoing it would mean, at a minimum, going back to the mid-1990's when online shopping was a novelty, online banking was non-existent, and software engineers were largely required to be on-site in order to do their jobs.

Yet how secure is this promise? Certainly, encryption technologies, like RSA and ElGamal, appear secure in the abstract. However, no one uses them in the abstract. Rather, for the most part, they purchase them from a vendor or download them from an open source site. This means that they must implicitly trust that the work of the hardware/software implementer has been both *sound* and *honest*. Being *sound* means that the encryption algorithm has not been weakened by implementation errors. Being *honest* means that the implementer has not, surreptitiously, embedded a backdoor (a.k.a. trapdoor) such that the implementer can invade the privacy of her customers. The practical validity of the promise of privacy offered by modern encryption rests upon these two pillars.

It is the second pillar that will be explored in this paper. More pointedly, the question of the extent encryption users, Alice and Bob, can trust an encryption implementer, EveCorp, will be examined. It will be examined by considering three encryption algorithms, RSA, ElGamal and Electronic Book Cipher, and then showing how the implementation of each algorithm leaves room for a backdoor.

The first algorithm to be examined will be RSA. Based upon the work of Ross Anderson, it will be shown that a backdoor can be planted into RSA by ensuring that the primes used to construct the public/private keys are on a specific arithmetic progression. By knowing this progression, a malicious implementer can determine the primes used to construct the public key modulus and then use this to determine the private key. Thus, she can read the private correspondence of the unsuspecting user.

The second algorithm to be examined will be ElGamal. Based upon the work of Young and Yung, it will be shown that a backdoor can surreptitiously be added by finding a valid private exponent that has a certain structure in its lower order bits. Again, by knowing this structure, EveCorp can secretly read the correspondence of unsuspecting users like Alice and Bob.

The last algorithm to be examined will be a new algorithm, which will be referred to henceforth as 'Electronic Book Cipher' or 'EBC'. This algorithm takes one or many electronic texts and uses them to construct a dictionary of word and letter replacements. Afterwards, it randomly selects possible replacements in order to create the ciphertext. Of course, EBC is significantly simpler than either of the other two algorithms being examined. In fact, after examining RSA and ElGamal, it will be fairly obvious that the more sophisticated an encryption algorithm the more opportunities to embed backdoors during implementation. Yet, in spite of this, it will be shown that simplicity itself does not always convey immunity to backdoors. A backdoor will be displayed for EBC such that some of the numbers used to encode words and letters are prefixed in a certain way. This prefixing,

again, allows EveCorp to violate the trust that encryption users have in their chosen encryption method.

By the end of this paper, by showing how easy it is to compromise the implementations of encryption algorithms, it will be clear that the trust the public has that their emails, online transactions, etc can remain private through encryption must be understood as largely a matter of faith. The public must, at some point, trust that the encryption implementer has done a sound and honest job implementing an algorithm. This is because, as will have been shown, backdoors can be planted into the implementations of most, if not all, of modern encryption algorithms.

Backdoor into RSA

The first backdoor to be presented is a symmetric backdoor into RSA. It is based upon the work of Ross Anderson. [And93] It allows efficient and reliable access to the contents of compromised RSA encrypted messages. Furthermore, compromised outputs are largely not distinguishable from uncompromised outputs, making it very difficult to discover. Finally, even with knowledge of its presence, external discovery and usage of the backdoor would be very difficult, given certain assumptions.[1]

At the high level, this backdoor requires that a backdoor key (prime) be generated and implanted within the key generation portion of the RSA algorithm. This backdoor prime, call it A, is then given a relationship to primes p and q using Dirichlet's Theorem. Of course, since the product of p and q become the public key modulus n, the backdoor prime A also has a relationship to n. Using this relationship, some much smaller numbers, p_{small} and q_{small}, used in the construction of p and q, respectively can be discovered. Using p_{small} and q_{small}, p and q can be easily discovered. Finally, using p and q, the private key d can be rediscovered. Thus, any messages encrypted using the RSA public modulus, n, are compromised. [And93]

Before describing the algorithm, a quick review of Dirichlet's Theorem (DT) is warranted. Dirichlet's Theorem is stated as follows:

> Let a, b, $k \in \mathbb{Z}$. If a and b are coprime, then the arithmetic progression $\{ ak + b \}$ contains infinitely many prime numbers. [Dud08]

The upshot of DT is that if a and b are fixed such that gcd(a, b) = 1, then as k increases (or decreases) infinitely many prime numbers will be encountered.

Let x be one of the primes encountered with a fixed a and b. It is also worth noting that $b \equiv x \pmod{a}$. This means that, given knowledge of a, the value of b can be "released" from x. Eventually, this is important for the RSA backdoor since it allows us to release a smaller product from the RSA public key modulus, n, which can then be factored and used to discover p and q, the primes that make up the product n.

Recall that this backdoor involves compromising the RSA public key generation process. The algorithm for the generation of compromised RSA public keys begins by assigning a unique, very large, prime number that will serve as the backdoor key. Again, this backdoor key is known as A. If the RSA primes p and q are expected to be t-bits in length, then A should be roughly (3/4) * t-bits in length.

[1] These assumptions are: (1) each compromised RSA device has a unique backdoor key; and (2) each backdoor key has a sufficient number of bits to preclude discovery.

With A defined, both p and q are discovered using the following compromised algorithm:

Algorithm: RSA Backdoor Prime Generation
Input:
    A - the prime backdoor key;
    t - the size of the requested prime;
    p_{small} - random integer of t/4-bits;
Output:
    p - the prime to be used for the public key modulus (product)
Let k = random integer of t/4-bits, seeded with p_{small};
Let p = null;
While p = null {
    if ($Ak + p_{small}$) is prime
        $p = Ak + p_{small}$;
    else
        $k = k + 1$;
}
return p;

Given that this algorithm is used to generate both p and q, the end result is a value of n, the public key modulus, that has the following structure:

$n = (Ak_p + p_{small})(Ak_q + q_{small}) = A^2k_pk_q + Ak_qp_{small} + Ak_pq_{small} + p_{small}q_{small}$ [Kal94]

Thus, just as p and q bear a special relationship to A, n does as well.[2]

With the value of A and knowledge of the structure of n, the values of p_{small} and q_{small} can be released from n using the modulo operation. This is done as follows:

$p_{small}q_{small} \equiv n \pmod{A}$

Of course, $p_{small}q_{small}$ is a product so it must be factored. However, remember that p_{small} and q_{small} each has only t/4-bits. Therefore, the time required to factor $p_{small}q_{small}$ is significantly less than the time required to factor n, i.e. pq. In fact, if p and q are supposed to be 300 bits in size, then p_{small} and q_{small} will each be no more than 75 bits in size, which means that their product can easily be factored.

Once p_{small} and q_{small} have been obtained, it is easy to discover the values p and q. This is done by feeding the each value, p_{small} and q_{small}, back into the RSA Backdoor Prime Generation Algorithm. Since the algorithm seeds its random number generator using the small prime passed as an argument (it is p_{small} in the algorithm), the initial value of k will be the same; this means that, just as it did during the key generation process, it will return the first prime that it encounters as the value of k is incremented. For p_{small}, this will yield p; for q_{small}, this will yield q. Thus, the values of p and q have been salvaged using only the backdoor key A and the public key modulus n. RSA now has a backdoor.

Backdoor into ElGamal

The second backdoor to be presented is a symmetric backdoor into ElGamal. It is based upon the work of Adam Young and Moti Yung.[3] [YY04] It allows efficient and reliable access

[2] Examples of honest and dishonest RSA keys appear within the Appendix, Figures 1 and 2.

[3] Young and Yung's backdoor for ElGamal is actually implemented as an asymmetric backdoor. However, I could not get the version that they describe to work. Therefore, I changed it slightly and implemented it symmetrically.

to the contents of compromised ElGamal encrypted messages. Compromised outputs are largely not distinguishable from uncompromised outputs, making it very difficult to discover. Finally, similar to the backdoor into RSA, even with knowledge of its presence, external discovery and usage of the backdoor would be very difficult, given certain assumptions.

At the high level, like the RSA backdoor, this backdoor requires that a backdoor key be generated and implanted within the transient key generation portion of the ElGamal algorithm.[4] This backdoor key, again call it *A*, will be used to encrypt the previous transient key into the current transient key; in short, the current transient key will always be the previous transient key raised to power of *A* (and scrambled through masking and hashing). Of course, this means that the previous transient key must also be stored within the device. It also means that the backdoor will not work for the very first encryption pair; this pair will always be honest. Once the second pair is sent, however, it and all subsequent messages can be read by using *A* to free the current transient key from the previous one.[5] [YY04]

Again, this backdoor involves compromising the transient key generation process. Let *p* be the large, prime number that defines the group within which the private and transient keys must lie. The honest algorithm for generating transient keys simply involves generating a random number *y*, where $1 < y < \varphi(p)$. The dishonest algorithm for generating transient keys is as follows:

Algorithm: ElGamal Dishonest Transient Key Generation

Input:

A - the backdoor key;
p - the large prime that defines the group [part of the public key]
g - the primitive for the group defined by *p* [part of the public key]
y_prev - the previous transient key

Output:

y - the transient key to be used to encrypt a message portion

```
If y_prev = null {
        y = random number where 1 < y < φ(p);   // return an honest key
} else {
        attacker_exponent = g^A (mod p);      // setup to encrypt the key
        y = attacker_exponent^(y_prev) (mod p);  // encrypt the previous transient key
        y = apply_a_mask_to( y );              // mask for additional scrambling
        y = apply_hashing_to( y );              // hash, again, for scrambling
}
secretly_store_transient_key( y );              // store the transient key for later
return y;
```

Assuming that the attacker is able to reliably intercept message pairs, he/she can very easily decrypt and read them. The process for dishonest decryption very much resembles reverse of the dishonest transient key generation algorithm. It is described in the following:

Algorithm: ElGamal Dishonest Decryption

Input:

A - the backdoor key;
c - a ciphertext block
p - the large prime that defines the group [part of the public key]
h - the discrete logarithm [part of the public key]

---

[4] The 'transient key' is the value that public key discrete logarithm (g^x) is raised to during encryption. Often, the public key discrete logarithm is given the designation 'h' while the transient key is given 'y'.

[5] Examples of honest and dishonest ElGamal transient keys appear within the Appendix, Figures 3 and 4.

y_prev - the previous transient key

Output:

m - plaintext

y_current = y_prev^A (mod *p*); // Decrypt the current transient key - still scrambled

y_current = apply_a_mask_to(*y_current*); // Reverse the mask

y_current = apply_hashing_to(*y_current*); // Reverse the hashing

d = $h^{y\_current}$ (mod *p*); // Get the decryption multiple

m = *c* / *d* (mod *p*); // Finally, decrypt the ciphertext

return *m*;

Thus, given the previous transient key, any ElGamal encrypted message can be read.[6] ElGamal now has a backdoor.

An Aside Concerning the Generation of Primitives for ElGamal

One of the issues that arose during the creation of the backdoor into ElGamal concerned how to efficiently generate a cryptographically sound public key. Of course, this issue has nothing whatsoever to do with the backdoor. Rather, it is an issue for any ElGamal implementation, honest or not.

Remember the public key for ElGamal has three components. The first component is a large prime number, *p*, that defines the group within which the rest of the values, private and transient, will reside. The second component is a primitive, *g*, (generator) for the group defined by *p* that has order φ(*p*). The final component is $h = g^x$ (mod *p*), where *x* is the chosen private key.

The difficulty in generating the public key for ElGamal arises when trying to find a suitable primitive, *g*, once the group prime *p* has been chosen. At first glance, this wouldn't appear to be a problem; after all, every prime number *p* has φ(*p* - 1) primitive roots. [Dud08] And, in fact, deploying a probabilistic approach is the correct approach. The difficulty arises when it comes time to test a putative *g* to see if it is, indeed, a primitive of *p* that has order φ(*p*).

In order for a putative primitive *g* to be a suitable generator for ElGamal, it must meet the following conditions: [1] $1 < g < \varphi(p)$, i.e. *p* - 1; [2] gcd(*g*, *p*) = 1, which ensures, by Euler's Theorem, that $g^{\varphi(p)} \equiv 1 \pmod{p}$; and [3] there is no factor *t* of φ(*p*) such that $g^{\varphi(p)/t} \equiv 1 \pmod{p}$, which guarantees that the order of *g* is φ(*p*).[7] It is this third step, [3], that causes a problem since it requires that φ(*p*) be factored, which even given the best algorithms may demand an exponential or pseudo-exponential cost.

So, the question is: how can a suitable primitive *g* be found efficiently? Barring a very noteworthy breakthrough, the factoring of φ(*p*) appears to be an insurmountable obstacle. However, there is one advantage that comes with along with generating the primitive for ElGamal. Group prime *p* is being chosen as well as primitive *g*. This means that there is some flexibility afforded in the route to choosing a suitable *p*.

Previously, it was assumed that group prime *p* was chosen by simply finding a prime of the needed size. Instead of starting with *p* and then being forced to factor φ(*p*),

[6] An example of this decryption is within the Appendix, Figure 5.

[7] This follows from Lagrange's Theorem: Let gcd(*a*, *m*) = 1 and *a* have order *j* (mod *m*). Then ak ≡ 1 (mod *m*) iff *j* divides *k*.

perhaps φ(*p*) can be constructed from factors. In other words, what if *p* were chosen by constructing φ(*p*) from randomly chosen (and properly sized) factors and then checking whether φ(*p*) + 1 is prime? This would mean that there would no longer be any need to factor φ(*p*) since it factors would already be known. Thus, the most costly step that comes with testing primitives of *p* can be eliminated. Public keys for ElGamal encryption can, therefore, be found efficiently.

Description of the Electronic Book Cipher (EBC)

At this point it is worth asking, what is it about RSA and ElGamal that makes them susceptible to backdoors? A somewhat obvious hypothesis is the relationship that the plaintexts have to their respective possible ciphertexts. Whenever there is a one-to-many (or many-to-many) relationship between the characters of the plaintext and their respective possible ciphertext representations, there is an opportunity for a backdoor.[8] In RSA, this relationship exists because plaintexts will be encrypted differently depending upon the public key modulus being used. In ElGamal, this relationship exists because plaintexts will be encrypted differently depending upon the transient keys.

In order to more fully evaluate this hypothesis, the creation of a vastly more simple encryption algorithm is warranted. Call this algorithm the 'Electronic Book Cipher' (EBC). In EBC, plaintexts will bear also bear a one-to-many relationship to ciphertexts. Furthermore, its simplicity will bring this characteristic to the forefront. The upshot is that if EBC can support a backdoor, then most, if not all, encryption algorithms with this characteristic are susceptible.

At its core, EBC is just a simple book cipher. The key for EBC consists of an ordered set of the titles of books and must be shared between sender-recipient. Unlike a normal book cipher, its encryption dictionary is built dynamically and can, probabilistically, change between runs. Its decryption dictionary, on the other hand, is quite static and can be used to decrypt ciphertexts.

In order to understand how EBC can support a backdoor, the encryption process must be understood starting with the construction of the encryption dictionary. The encryption dictionary is built each time a plaintext is encrypted; this is because some randomness is involved in its creation. It is constructed by processing the text of the books, indicated by the titles, in order. For each word in each book, its index is added to a list that is indexed by a hash table, unless more than a fixed number of indices (e.g. 500) have already been added to the list, in which case the index still has a random chance to replace one of the entries. Letters, for each word, are added as well and using a similar procedure, i.e. if the list for a letter has reached the maximum size then a new letter has a random chance of replacing one of the list entries.

After the encryption dictionary has been built, the plaintext is encrypted. For each word in the plaintext, the corresponding word in the encryption dictionary is found. If the word is found in the dictionary, then a member of the list of indices is selected randomly,

[8] I am rolling many-to-many relationship in with one-to-many. The main idea is that it must be possible to represent the same plaintext with many different ciphertexts such that a compromised ciphertext cannot easily be discerned from an honest ciphertext.

with an even distribution. If the word is not found in the dictionary, then the word is broken down into its letters and the letter dictionary is searched in a similar manner.[9]

Notice that, given the amount of randomness in the process, if a plaintext is encrypted multiple times using EBC, the ciphertext will almost never be the same.[10] This feature, of course, is in service of the hypothesis posited above, i.e. there is a one-to-many relationship between the plaintext and the possible ciphertexts. As such, EBC supports a backdoor where its ciphertext is compromised with enough information to reveal the encryption keys, i.e. the book titles.

Backdoor into the Electronic Book Cipher (EBC)

Placing the backdoor into EBC begins by recognizing that, generally speaking, there are lots of different possible indices (replacements) for the same word (and/or letter). For example, the word 'the' appears in *War And Peace* over 28,000 times; the word 'smile' appears 566 times; even the word 'unintelligible' appears 6 times. This fact provides lots of options when choosing replacements for words within the plaintext.

Consider the following change to the EBC algorithm: Instead of choosing replacements for letters based solely on probability, choose a replacement that can somehow represent a letter of one of the book titles that make up the encryption key. As an example, let the plaintext word being decrypted be 'pencil'. Let the number of possible replacements for 'pencil' be 432 and let some of these be '00234', '102873', '87977', and '10057'. Let the one of the book titles be 'War and Peace'. So, the idea is that instead of randomly choosing a replacement for 'pencil', choose, if possible, a replacement for it that somehow represents the next character of the book title (or titles). Assume that this process has just started; so by remembering that the ASCII for 'W' is 87 and then choosing '87977', the letter 'W' in the title can be represented. The same can be done for the rest of the characters in the title.[11]

The upshot of all of this is that, with a few refinements, this process can hide the titles of the books used to construct the encryption dictionary within the ciphertext.[12] With the titles of the books used during the encryption process in hand, ciphertexts are clearly compromised. Therefore, the Electronic Book Cipher, in spite of its simplicity, can also support a backdoor.

Conclusion

In the preceding, three backdoors were planted into the implementation of three very

[9] An example of an honest EBC ciphertext appears in the Appendix, Figure 6.

[10] Assuming that the books chosen are large enough to support lots of different possible word replacements.

[11] An example of a dishonest EBC ciphertext appears in the Appendix, Figure 7. An example of a dishonest EBC key exposure appears in the Appendix, Figure 8.

[12] Some of the refinements include: (1) Constructing the encryption dictionary so that indices whose first two characters can represent letters 'A' - 'Z' in ASCII; and (2) when a index replacement is not found that can also represent a specific letter, then it is best to choose one whose first two digits does not represent any other letter in ASCII as well; this is because two digits sequences that have no ASCII representation are ignored.

different encryption algorithms; two of the three encryption algorithms are generally considered to be highly secure. None of the backdoors required intellectual leaps in order to be understood. Furthermore, none of the backdoors required special training in order to implement; just a little ingenuity. This strongly suggests that planting backdoors into the implementation of encryption algorithms is something that can be accomplished by a great many people, let alone encryption implementers.

This suggestion leads one back to the assertion that trust the public has that encryption devices and software are not compromised is largely a matter of belief. As has been shown, the difficulty of implementing hidden backdoors is no impediment at all; in fact, they are quite easy to implement. Thus, this belief, at least for now, rests upon the public outrage that might occur if an implementer were found to be dishonest. Frankly, there is no other pillar for this belief to rest upon.

Appendix

```
e = 938548979932684192912682821581320544594789030164619059199783190156939470692310260059742117337930805914627366311257693964672312365587416140319953949
m = 620166436998158918666817767829683329294440255226047829397251407866415173137872432277761520497537652455904006605248252371964272455077088104244345783748247571939259596967955011946412007356299947014632603458721783290482583491
d = 761422615601077323109493149370545040271362229129467597008399064283951962324824757193925959696795501194641200735629994701463260345872178329048258349
```

Figure 1 - Honestly Generated RSA Public Keys

```
e = 386790337775613336797448143302837417775167115490883932124369286767156850268863525100854755481482176601525370409756497184290026175086493250786603096393461170480771185445079509324019324113327772648280569146923692361252675198181
d = 168668472696104237102792069353279819806474595781697042858359200280406862156356422937320502105544088310600311567903022116424792032847981967978652292432
```

Figure 2 - Dishonestly Generated RSA Public Keys

```
367701373653553269150332931167393987946048523884082649188040833069169893573093333883726626092991679988297373316941031697562162988725425515943259640995774236
4612594
898463771352447997235822811443115822552412088254270939449746763776175985586877097374278353666760281800657611341582888657869898595444868920958544445356322009840
```

Figure 3 - Honest ElGamal Ciphertext of Plaintext (“We intend to begin”)

```
720733345096373601441300333352462955637597610478032085916927040306447603793429673816717585126188188003463163876923449878118807666618498512656458145878817857016634
300751809787829536798952670294802778642285238044647489453125965700690981412923007518097878295367989526702948027786422852380446474894531259657006909814:1292
```

```
3976209750529737096501019228624433160285153135154232915924778057108990737182742671020
```

Figure 4 - Dishonest ElGamal Ciphertext of Plaintext (“We intend to begin”)

```
> eg.DishonestDecrypt(
'3007518097878295367989526702948027786422852380446474894531259657006909814:12923976209750529737096501019228624433160285153135154232915924778057108990737182742671020',
'720733345096373601441300333352462955637597610478032085916927040306447603 7' )
('to begin',
'3007518097878295367989526702948027786422852380446474894531259657006909814')
```

Figure 5 - Dishonest Decryption ElGamal Ciphertext of Plaintext (“We intend to begin”)

```
71308 441686 554583 124529 12726 208027 355849 10052 167195 305244:1 450108:1
246956:2 556800:2 539960:3 549694:0 275184:1 525203:0 197776:0 549119:3
488707:6 531084:2 -1:0 501231:2 03722:6 432655:0 262125:5 463022:0 403256:8
411505:4 501854:7 484837:3 -1:0 359013 328617 269730 197714 16459 250797
296272 102812 14023 168722 400683 304738 537762 520659 44241 42788 399666
06891 536013 22467 124043 68908 492759 09808 341786 551986 15533 381627 07594
90328 28254 06891 285721 511544 222223 287670 263120 23144 325933 242551
103270 421051 179064 565164 16767 391407 02619 200990 100226 402991 551986
55826 328598 519974:4 269881:4 227687:0 452574:1 514032:5 04996:0 544069:3
407137:2 151524:5 -1:0
```

Figure 6 - Honest Encryption of First Part of Zimmerman Telegram using EBC

```
00627 100600 545449 03857 448727 546671 87200 561296 167195 555511:2 558572:1
550353:1 551586:2 526271:0 558281:7 550353:1 564393:8 552214:0 558281:7
551586:2 480061:5 -1:0 526271:0 555511:2 386875:0 65460:3 531034:1 550353:1
564393:8 558572:1 551586:2 -1:0 291460 00627 00313 197714 399070 02926 561296
82284 545449 04945 546671 39040 283044 561296 44241 42788 399070 546671 41311
561296 65626 262405 492759 00627 06063 551986 484163 102315 561296 04816
448727 546671 08827 199265 06063 00006 12611 06063 00008 12611 46174 192669
23317 405794 374100 08630 448727 78016 00644 286304 551986 68049 545449
550353:1 551586:2 552214:0 554558:1 558572:1 01050:3 555511:2 551586:2
550353:1 -1:0
```

Figure 7 - Dishonest Encryption of First Part of Zimmerman Telegram using EBC

```
00627 100600 545449 03857 448727 546671 87200 561296 167195 555511:2 558572:1
550353:1 551586:2 526271:0 558281:7 550353:1 564393:8 552214:0 558281:7
551586:2 480061:5 -1:0 526271:0 555511:2 386875:0 65460:3 531034:1 550353:1
564393:8 558572:1 551586:2 -1:0 291460 00627 00313 197714 399070 02926 561296
82284 545449 04945 546671 39040 283044 561296 44241 42788 399070 546671 41311
561296 65626 262405 492759 00627 06063 551986 484163 102315 561296 04816
448727 546671 08827 199265 06063 00006 12611 06063 00008 12611 46174 192669
23317 405794 374100 08630 448727 78016 00644 286304 551986 68049 545449
550353:1 551586:2 552214:0 554558:1 558572:1 01050:3 555511:2 551586:2
550353:1 -1:0 546671 80107 250287 399070 558281:7 551586:2 69558:1 531034:1
526271:0 -1:0 00257 551986 405794 531034:1 550353:1 564393:8 00784:7 554558:1
558572:1 531034:1 -1:0 546671 457356 399070 59578 560194 65040 545449 307790
307790 00145 23817 546671 116506 561296 546671 67413 02009 04551 69867 151460
00165 524376 563789:0 464270:6 491188:0 439755:2 519792:1 551338:2 546140:3
94334:2 -1:0
>>> dishonest_book_cipher.get_book_titles_from_ciphertext( ctxt )
'WARANDPEACEMF'
```

Figure 8 - Dishonest Key Exposure from First Part of Zimmerman Telegram using EBC

References

[And93] Ross Anderson. Practical RSA Trapdoor. *Electronic Letters*. 29(11): 995, 1993.

[CS03] Claude Crépeau and Alain Slakmon. Simple Backdoors for RSA Key Generation. *CT-RSA'03 Proceedings of the 2003 RSA conference on The cryptographers' track*. pgs. 403-416, 2003.

[Dud08] Underwood Dudley. *Elementary Number Theory*. Dover Publications, Mineola, NY, 2008.

[FP09] Evangellos Fountas and Constantinos Patsakis. Creating RSA Trapdoors Using Lagrange Four Square Theorem. *2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*. pgs. 779-782, 2009.

[Kal94] Burton Kaliski. Anderson's RSA Trapdoor Can Be Broken. *Electronic Letters*. 29(15): 1387-88, 1993.

[Rie94] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. Birkhäuser, Boston, MA, 1994.

[Sin00] Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor Books, 2000.

[Tuc85] Barbara Tuchman. *The Zimmerman Telegram*. Ballentine Books, New York, NY, 1985.

[TW06] Wade Trappe and Lawrence Washington. *Introduction to Cryptography with Coding Theory*. Pearson Prentice Hall, Upper Saddle River, NJ, 2006.

[Yan02] Song Yan. *Number Theory for Computing*. Springer-Verlag, Berlin, Germany, 2002.

[You04] Adam Young. Mitigating Insider Threats to RSA Key Generation. *Cryptobytes, RSA Laboratories*. 7(1): 1-15, 2004.

[YY96] Adam Young and Moti Yung. The Dark Side of Black Box Cryptography, or: Should We Trust Capstone?. *CRYPTO '96 Proceedings of the 16th Annual*

International Cryptology Conference on Advances in Cryptology. pgs. 89-103, 1996.

[YY04] Adam Young and Moti Yung. *Malicious Cryptography: Exposing Cryptovirology*. Wiley, Indianapolis, IN, 2004.