Stronger Password-Based Encryption Using All-or-Nothing Transforms

Greg Zaverucha
Microsoft

August 5, 2015

1 Introduction

When encrypting data with a low-entropy key, the primary threat to consider is a brute-force key search. Let $C = C_0, \ldots, C_{s-1}$ be a ciphertext encrypting a plaintext $P = P_0, \ldots, P_{s-1}$, produced by a block cipher (a concrete example is AES in CBC mode). In a brute-force attack where P_0 is known, the attacker could focus on C_0, i.e., for each candidate key K, check whether $D_K(C_0) = P_0$. One idea to increase the cost of a brute force attack is due to Rivest [8]. Apply a randomized encoding to the plaintext, $P' = \mathsf{Encode}(P)$, such that in order to decode P' and obtain P, one needs *all of* P'. In this way $D_K(C_0) = P'_0$ does not give enough information about P to let the adversary decide if K is correct. In order to check that K is the correct key, the attacker must decrypt *all* s blocks of C to recover P', then compute $P = \mathsf{Decode}(P')$. The Encode operation is called an *all-or-nothing transform* (ANT), because it cannot be even partly reversed without all of the encoded output. The ANT is randomized, but not keyed (no secret values are required in Encode and Decode).

This adds a factor of s to each guess, which can be considerable. For example, when P is 1 GByte, and a 128-bit block cipher is used $s \approx 2^{25}$. So the cost of brute forcing a 40-bit key goes from 2^{40} to 2^{65} block cipher operations. There may also be a significant I/O cost, as the attack must work with 2^{25} ciphertext blocks instead of just one.

We are unaware of any papers, systems or software that uses Rivest's ANT idea for password-based encryption (PBE), based on the observation that export strength keys (Rivest's original motivation) are similar to passwords (in that they are both

of low-entropy). We'll explore this idea with two applications of password-based encryption as examples.

1. *Encryption of Microsoft Office documents.* The Microsoft Office suite of applications (Word, PowerPoint, Excel) have a feature that enables users to password-protect a document. Encryption and authentication keys are derived from a password, and then used to encrypt the document. The feature is documented in [4].

2. *The SQLCipher encrypted database [10].* This is an encrypted version of the SQLite database. SQLCipher encrypts database with a key derived from a user-chosen password. A page is 1024 bytes. By encrypting at the page-level instead of the file-level, random access to a large database is possible (without a full decryption).

Both applications use PBKDF2 [6] for key derivation[1], which is a salted, iterated function, designed to increase the cost of a brute-force password search. The encryption technique we describe is independent of the key derivation step, the input to encryption is a key (rather than a password). The choice of key derivation only becomes relevant in the security analysis of the overall PBE operation.

Similar features exist for ZIP and other archive formats, for PDF files and PFX files (but the encryption and key derivation details are different). The scrypt encryption utility [9] can password protect any file. The key is derived from the password using the scrypt key derivation function [7]. These other applications weren't explicitly considered by this work but may benefit from the technique described here.

Memory-hard KDFs. Some password-based key derivation functions, like scrypt, are designed to use a large amount of memory. This defense is similar to our approach of using all-or-nothing transforms, since they both increase the memory requirement of a brute force attack. The main difference is that our approach leverages the memory usage that the defender will use anyway, while memory use in a memory-hard KDF is purely overhead (with respect to the encryption operation). A key difference is that our approach does not randomize access to memory to prevent efficient parallelization of a brute-force attack, so our technique cannot directly replace a memory-hard KDF. An open problem is to design an all-or-nothing transform with a randomized access pattern, and investigate its use in this context. More generally, we went modular in our design, keeping the KDF and PBE operations separate, but it may be fruitful to look at tightly integrating them.

[1]Office uses a slight variation of PBKDF2, as documented in [4].

1.1 Summary

Our findings are as follows.

- We describe three ways of combining an all-or-nothing transform with encryption and authentication primitives to create an authenticated encryption mode suitable for password-based encryption.
  - All of them have one-pass encryption and require two-pass decryption.
  - Two are generic, i.e., they work for any block cipher and MAC, and the third uses a specific mode of operation for greater efficiency.
  - The computational overhead (for the defender) in our most efficient scheme (Scheme 3) is an additional block cipher call for each input block.
  - The ANT in all three constructions is an instance of OAEP.
- In terms of security, in general re-encoding the plaintext will not decrease security. The increase in difficulty of a brute force attack depends on multiple factors.
  - The size of the plaintext. In general a plaintext of n blocks increases the cost of a brute force attack by a factor of n. For example, in the SQLCipher use case, the plaintext is 64 AES blocks, resulting in an additional 64 AES operations per password guess in a brute-force attack. Whether this protection is sufficient to justify the added cost to the defender is debatable.
  - The relative cost of the key derivation step. Again, using the SQLCipher example, an additional 64 AES operations have trivial cost compared to 64k PBKDF2 iterations.
  - Whether the same password is used for multiple encryptions with different file sizes. The attacker can always choose to attack the smallest file.

To conclude, there are some settings when encoding with an ANT provides some additional resistance to brute-force attacks in the context of PBE, however the advantage is hard to quantify, and the range of outcomes is large. In the best case brute force attacks are made many times harder, and in the worse case the ANT technique provides no significant benefit (though security is not reduced, only efficiency).

2 All-or-Nothing Transforms

In this Section we review some known all-or-nothing transforms (ANTs). Readers familiar with OAEP may skip this section, since it is the ANT our constructions use.

Rivest's ANTs Rivest describes two ANTs in [8]. The first uses only block cipher operations and the second is simpler, but uses a hash function. Both are instances of OAEP (described below). Let E denote the blockcipher. The input is a plaintext $P_0, \ldots, P_{s-1}$, and the output is $y_0, \ldots, y_s, y_s$. The values P_i and y_i are blocks of the same size as the blocksize of E. Note the output is one block larger than the input.

- Choose a random key for E, denoted K.
- Compute $y_i = P_i \oplus E_K(i)$ for $i = 0, \ldots, s-1$.
- Compute $y_s = K \oplus e_0 \oplus \ldots \oplus e_{s-1}$, where $e_i = E_{K_0}(y_i \oplus i)$, and K_0 is a fixed, publicly known value.

The first s values are encryption of the plaintext in counter mode under the random key K. The final value is an "encryption" of K such that $y_0, \ldots, y_{s-1}$ are required to decrypt it.

The cost of this ANT is $2s$ encryptions. Encode makes a single pass over the data (the implementation must compute y_{s+1} incrementally at each input block, not as described above). Decode must make two passes.

The ANT is randomized by the choice of the key K. If K were fixed, and the ciphertext were known then an attacker could predict the value of y_0 and use this as a shortcut in a brute force attack. It has the advantage that the same block cipher may be used for the ANT as for the encryption, so no new primitives are added to the overall encryption operation. Only the forward mode of encryption is used (not decryption).

Rivest mentions y_s can be computed differently. Using a hash function H, compute

$$y_s = H(y_0 \| \ldots \| y_{s-1}) \oplus K$$

This ANT requires s encryptions, and hashing an input of length bs bits. Encode can be done in one pass, and Decode requires two passes. This is an instance of OAEP.

Optimal Asymmetric Encryption Padding (OAEP) The OAEP padding mode [2], used for RSA encryption, is an all-or-nothing-transform. A formal proof of this is given by Boyko [3], in the random oracle model. OAEP is a randomized ANT,

and uses two functions G and H. G is analogous to the counter mode encryption in Rivest's constructions used to compute $y_0, \ldots, y_{s-1}$, and H plays the role of the function used to compute y_s. Aside from the padding step of standard OAEP (which is only necessary when the output is required to have a fixed length), the Rivest constructions are the same, but with different choices of G and H.

OAEP is defined as follows. Let $G : \{0,1\}^b \rightarrow \{0,1\}^{sb}$ be a PRF that expands the seed r to a random value as large as the input message. We use the notation $G_r(i)$ to denote the i-th b-bit block of $G(r)$. The function $H : \{0,1\}^* \rightarrow \{0,1\}^b$ is a cryptographic hash function. The Encode operation is defined:

- Choose a random b-bit value r.
- Compute $y_i = G_r(i) \oplus P_i$ for $i = 0, \ldots, s-1$
- Compute $y_s = r \oplus H(y_0 \| \ldots \| y_{s-1})$.

The Decode operation is defined:

- Compute $y'_s = H(y_0 \| \ldots \| y_{s-1})$.
- Compute $r' = y'_s \oplus y_s$.
- Compute $P_i = y_i \oplus G_{r'}(i)$, for $i = 0, \ldots, s-1$.

We've defined r to be a b-bit value for convenience, but it does not have to match the blocksize. A possible choice for H is SHA-256 truncated to b-bits, and G could be AES in CTR mode.

Variable-Length Block Ciphers Rivest observes that variable-length block ciphers, such as BEAR and LION [1], can be inherently "all-or-nothing" because decrypting one block can mean decrypting the whole message (when the block size equals the message size). We chose not to investigate using variable-length block ciphers since the costs are comparable to applying OAEP before encryption.

Information Theoretic ANTs Stinson [12] looks at unconditionally secure ANTs, and gives very efficient constructions based on linear transforms. Unfortunately the definition is not strong enough for our application. In [12], an ANT is defined by three conditions; the first two ensure correctness, and the third is:

$$\mathsf{H}(P_i \| y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_s) = \mathsf{H}(P_i)$$

where H is the entropy function. However in our case, some of the P_i are known, so $\mathsf{H}(P_i) = 0$. For some of the constructions in [12] it is easy to see that given, e.g., y_1 and y_2, one can check whether y_1 encodes P_1.

The stronger definition given by Boyko [3] is required for our generic PBE constructions. However, we will see that with a careful choice of block cipher mode, a weaker transform is sufficient (albeit one with different properties from [12]).

3 Password-Based Encryption

Our goal is to combine an all-or-nothing transform with an authenticated encryption scheme to get a encryption scheme that requires $O(s)$ work to determine whether a candidate key is correct. After considering multiple ways to combine the ANT, encryption and MAC algorithms, we present three constructions for comparison. The first two are generic, loosely following the MAC-then-encrypt and encrypt-then-MAC designs. The third uses a specific encryption mode and ANT to reduce the cost.

The threat model assumes the attacker knows some portion of the plaintext, and could use this to check whether a password guess is correct. The known-ciphertext attack starts by by choosing a password (probably from a dictionary, or according to some rules). Then, a key is derived from the password (and optionally a salt), with key derivation algorithm specified by the PBE scheme. Finally, part of the ciphertext is decrypted, and the attacker outputs `correct` if the plaintext matches the known portion. For simplicity, assume the attacker knows one or more blocks of plaintext.

We also ignore the key derivation step. The inputs for our schemes are the MAC and encryption keys, K and K', along with a randomly chosen IV (when required by the encryption mode). We do not recommend deriving an IV from the password since we don't assume that the encryption algorithm will keep the IV secret (in general this is not a requirement of block cipher modes of operation).

3.1 Scheme 1: MAC-then-Encrypt

The first scheme uses a "MAC-then-encrypt" approach, the ciphertext is

$$C = E_K(ANT(P||M_{K'}(P)))$$

where

ANT is OAEP with functions G and H (as described in §2).

P is the plaintext, divided into s blocks each of b-bits, $P = P_0\| \ldots \|P_s$.

E_K is a blockcipher encryption with the key K. The blocksize is b bits.

$M_{K'}$ is a MAC with the key K'. In our description the length of the authentication tag is b bits.

H is the hash function used for OAEP. It has two functions, to work on streams of data: $H.\mathsf{ProcessData}(d)$ appends d to the data to be hashed, and $H.\mathsf{Finalize}()$ returns the b-bit digest.

G_r The PRG used for OAEP with seed r. $G_r(i)$ denotes the i-th b-bit block of output.

Algorithm 1 gives the pseudocode to implement Scheme 1 with one-pass encryption and two-pass decryption.

Remarks OAEP has the required properties, and our construction use it leaving open the choice of G and H. However, there may be other ANTs that are also suitable for Scheme 1, provided they have similar security properties as OAEP.

3.2 Scheme 2: Encrypt-then-MAC

In Scheme 2, ciphertexts are a pair (C, T) where

$$C = E_K(ANT(P)), \text{ and}$$
$$T = M_{K'}(C)$$

(the notation of §3.1). Algorithm 2 gives pseudocode to implement Scheme 2.

Remarks Note that any nonce or IV used by E must be authenticated, included with C in the computation of T. Decryption must be done carefully to use only two passes and not decrypt before validating the MAC[2]. A naïve implementation might require three-passes, one to check the MAC, one to decrypt and a third to decode. Alternatively, we use two passes, but combine MAC verification with decryption. The trick is to decrypt while recomputing the MAC, but to validate the MAC before decrypting the last block of C (which is required to decode the ANT). This works for the OAEP family of ANTs, but care must be taken with other ANT constructions.

[2]This property is sometimes called the "cryptographic doom principle" [5] since decryption before authentication has led to multiple security flaws.

Algorithm 1 Pseudocode for Scheme 1: MAC-then-Encrypt

Parameters: Algorithms E, M, G, and H, blocksize b (bits)
Keys: K and $K', \in \{0,1,\}^b$
Plaintext: $P = P_0 \| \ldots \| P_{s-1}$ (b-bit blocks)
Ciphertext: $C = C_0 \| \ldots \| C_{s+1}$ (b-bit blocks)

Encrypt(K, K', P)
Choose $r \in_R \{0,1\}^b$.
Initialize E_K, $M_{K'}$, G_r, H
for $i = 0, \ldots, s-1$ **do**
  $M_{K'}$.ProcessData(P_i)
  $temp = G_r(i) \oplus P_i$
  H.ProcessData$(temp)$
  $C_i = E_K(temp)$
end for
$temp = M_{K'}$.Finalize$() \oplus G_r(s)$
H.ProcessData$(temp)$
$C_s = E_K(temp)$
$temp = H$.Finalize$() \oplus r$
$C_{s+1} = E_K(temp)$
Output $C = C_0 \| \ldots \| C_{s+1}$

Decrypt(K, K', C)
Initialize $D_K, M_{K'}, H$
for $i = 0, \ldots, s-1$ **do**
  $P_i = D_K(C_i)$
  H.ProcessData(P_i)
end for
$tag = D(C_s)$
H.ProcessData(tag)
$r = H$.Finalize$() \oplus D_K(C_{s+1})$
Initialize G_r
for $i = 0, \ldots, s-1$ **do**
  $P_i = P_i \oplus G_r(i)$
  $M_{K'}$.ProcessData(P_i)
end for
$tag = tag \oplus G_r(s)$
if tag `!=` $M_{K'}$.Finalize() **then**
  return `error`
end if
Output $P = P_0 \| \ldots \| P_{s-1}$

3.3 Scheme 3: OFB and OAEP$^-$

This is a variant of the encrypt-then-MAC scheme that uses a specific encryption mode and an OAEP variant (denoted OAEP$^-$). OAEP$^-$ simplifies OAEP by removing H, or equivalently, by setting $H(x) = 0$ for all inputs x. OAEP$^-$ reduces the defender's computational overhead, by avoiding the hash function H. OAEP$^-$ is *not* an all-or-nothing transform. The ciphertext for Scheme 3 is (C, T) where C is computed as

$$\begin{aligned} C &= E_K(\text{OAEP}^-(P)) \\ &= E_K((G(r) \oplus P) \| r) \end{aligned}$$

Algorithm 2 Pseudocode for Scheme 2: Encrypt-then-MAC

Parameters: Algorithms E, M, G, and H, blocksize b (bits)
Keys: K and $K', \in \{0,1,\}^b$,
Plaintext: $P = P_0 \| \ldots \| P_{s-1}$ (b-bit blocks)
Ciphertext: $C = C_0 \| \ldots \| C_{s+1}$ (b-bit blocks)

Encrypt(K, K', P)
Choose $r \in_R \{0,1\}^b$.
Initialize E_K, $M_{K'}$, G_r, H
for $i = 0, \ldots, s-1$ **do**
    $temp = G_r(i) \oplus P_i$
    H.ProcessData$(temp)$
    $C_i = E_K(temp)$
    $M_{K'}$.ProcessData(C_i)
end for
$temp = H$.Finalize$() \oplus r$
$C_s = E_K(temp)$
$M_{K'}$.ProcessData(C_s)
$C_{s+1} = M_{K'}$.Finalize$()$
Output $C = C_0 \| \ldots \| C_{s+1}$

Decrypt(K, K', C)
Initialize $D_K, M_{K'}, H$
for $i = 0, \ldots, s-1$ **do**
    $M_{K'}$.ProcessData(C_i)
    $P_i = D_K(C_i)$
    H.ProcessData(P_i)
end for
$M_{K'}$.ProcessData(C_s)
if C_{s+1} `!=` $M_{K'}$.Finalize$()$ **then**
    return `error`
end if
$r = H$.Finalize$() \oplus D_K(C_s)$
Initialize G_r
for $i = 0, \ldots, s-1$ **do**
    $P_i = P_i \oplus G_r(i)$
end for
Output $P = P_0 \| \ldots \| P_{s-1}$

and

$$T = MAC_{K'}(IV||C) \ .$$

The cipher E is chosen to be AES in OFB mode, since OFB mode has the property that it is not possible to decrypt a block without decrypting all previous blocks in the ciphertext. Therefore, the attacker can't decrypt the last block to recover r, without making a pass over the whole ciphertext. In our implementation, M is GMAC, and G is AES-CTR. The remarks from Scheme 2 apply to Scheme 3 as well. Algorithm 3 gives the pseudocode to implement Scheme 3.

3.4 Other Possible Variants

There are some additional combinations that we did not investigate, which may have interesting properties. Replacing H in OAEP with HMAC and omitting a separate

Algorithm 3 Pseudocode for Scheme 3: OFB and OAEP$^-$

Parameters: Algorithms E, M, G, blocksize b (bits)
Keys: $K, K', \in \{0,1,\}^b$,
Plaintext: $P = P_0 \| \ldots \| P_{s-1}$ (b-bit blocks)
Ciphertext: $C = C_0 \| \ldots \| C_{s+1}$ (b-bit blocks)

Encrypt(K, K', P, IV)
Choose $r \in_R \{0,1\}^b$.
Initialize E_K, $M_{K'}$, G_r
$M_{K'}$.ProcessData(IV)
for $i = 0, \ldots, s-1$ **do**
  $temp = G_r(i) \oplus P_i$
  $C_i = E_K(temp)$
  $M_{K'}$.ProcessData(C_i)
end for
$C_s = E_K(r)$
$M_{K'}$.ProcessData(C_s)
$C_{s+1} = M_{K'}$.Finalize()
Output $C = C_0 \| \ldots \| C_{s+1}$

Decrypt(K, K', C, IV)
Initialize D_K, $M_{K'}$
$M_{K'}$.ProcessData(IV)
for $i = 0, \ldots, s-1$ **do**
  $M_{K'}$.ProcessData(C_i)
  $P_i = D_K(C_i)$
end for
$M_{K'}$.ProcessData(C_s)
if C_{s+1} != $M_{K'}$.Finalize() **then**
  return `error`
end if
$r = D_K(C_s)$
Initialize G_r
for $i = 0, \ldots, s-1$ **do**
  $P_i = P_i \oplus G_r(i)$
end for
Output $P = P_0 \| \ldots \| P_{s-1}$

MAC may be a way to reduce the space overhead of encryption by one block. The ANT could also be applied outside the encryption (i.e, encode the ciphertext rather than the plaintext). Do other modes of operation have the same property as OFB? Namely, that the blocks must be decrypted in order. The requirement of a random IV may sometimes be removed, since the plaintext is randomized by the ANT.

4 Evaluation

In this section we discuss some of the advantages and disadvantages of using an ANT with password based encryption. The evaluation is informal.

Two-Pass Decryption All three ANT PBE schemes require two passes for decryption. This is a drawback when compared to common AES schemes like GCM and encrypt-then-MAC combinations of AES and HMAC, which can be implemented with a single pass. Many implementations choose to use two passes, in order to avoid decryption before checking the MAC (aka. the crypto doom principle [5]). If this is the case, then the overhead of a second decryption pass is already present, so only the computational overhead of of the new schemes is relevant.

Overhead vs. Attack Cost Adding an ANT to the the encryption scheme should increase the cost of a brute-force attack, but it also has a cost. We can easily compare the computational costs. Table 4 counts the number of *cryptographic operations per input block*, i.e., the number of inputs blocks that must be encrypted, hashed or MAC'd. This is a rough measure, but is still informative. The ratio (overhead/attack costs), describes how much of the defender work translates to attacker work. A ratio α means that for α defender operations the attacker must do one operation. Of the new schemes, Scheme 1 has the lowest ratio ($\alpha = 2$), while Scheme 2 has the highest ($\alpha = 4$) and Scheme 3 is in between ($\alpha = 3$) because OAEP$^-$ is cheaper than OAEP by a factor of s. For comparison, in GCM, since the attacker must only decrypt one block per guess, the ratio is $\approx s$. An attack with cost $s + 1$ seems inherent in the encrypt-then-MAC design, since the attacker can always verify the MAC to check a candidate password. By contrast, in the MAC-then-encrypt design, the MAC is protected by the encryption and ANT operations.

Our estimate of the attack cost is naïve, because it does not account for the memory and I/O costs of processing s-block ciphertexts, or any costs that might be amortized by a parallel attack or specialized hardware. (Recall that in a brute force attack multiple candidate password may be tried simultaneously, sharing computations or data, if this benefits the attacker.)

ANT and PBE We must also consider whether the benefits of using an ANT are sufficient in the context of PBE to justify the increased costs of encryption and decryption, and the increased implementation complexity. Since the attack cost increases by a factor of s, the plaintext size is important. In the extreme case of a 1-block plaintext (i.e., $s = 1$), there is no benefit to using an ANT.

	Scheme 1 (MtE)	Scheme 2 (EtM)	Scheme 3	AES-GCM
Enc/Dec cost	$4(s+1)$	$4(s+1)$	$3s+2$	$\approx 2s$
Attack cost	$2(s+1)+2$	$s+1$	$s+1$	2
Ratio	≈ 2	4	≈ 3	$\approx s$
Decrypt before MAC check	Yes	No	No	Optional

Table 1: Comparison of Schemes 1–3 (presented in this paper), and AES-GCM (for comparison). For the first three rows, the number of cryptographic operations per block is given. The final row indicates whether an implementation can check that the MAC is valid before decrypting any ciphertext.

The increased attacker work caused by the ANT must be put in context, since the key derivation step is designed to be expensive. We can make some rough estimates to determine when it becomes beneficial to use an ANT as the plaintext size increases. Table 4 estimates the benefit provided by using an ANT, by comparing the number of cryptographic operations (hash function and block cipher calls), required per guess in a brute-force attack when an ANT is used, compared to when it isn't. More precisely, if A is the number of operations per guess when an ANT is used, and B is the number when an ANT is not used, then the *slowdown factor* is A/B. A slowdown factor of two means that each guess in a bruteforce attack is twice as slow when the ANT construction is used. In Table 4 we show the slowdown factor when the number of PBKDF2 iterations varies. With smaller iteration counts, the ANT technique is more effective for smaller plaintexts. For example, at 50K iterations, the slowdown factor is 1.6 for a 1MB plaintext, while at 5K iterations it is 1.6 for a 100KB plaintext.

Again, we stress that this is a rough comparison, because it makes the following simplifications. i) It assumes attacker must make a single additional pass over the data, true for Schemes 2 and 3 but not Scheme 1 (it would require two additional passes). We also count a PBKDF2 iteration as two hash function calls (ignoring the specific hash function used), and assume these have the same cost as a block cipher call. We've also again assumed the attack uses the naïve algorithm.

From these examples, we can conclude that the technique can provide meaningful defense against brute force attacks, especially when the plaintext is large. Going back to our examples of Section 1, ANT PBE could strengthen encryption of Office documents.

For the SQLCipher example, the plaintext size is small and fixed at 1024 bytes.

Plaintext size	Decryption	PBKDF2	Without ANT	With ANT	Ratio
100KB	6250	200K	200K + 1	206250	1.03
1 MB	62500	200K	200K + 1	262500	1.31
10 MB	625000	200K	200K + 1	825000	4.12
100 MB	6250000	200K	200K + 1	6450000	32.25

Table 2: Comparison of brute-force attack costs with various plaintext sizes. The *Decryption* column gives the number of block cipher operations to decrypt the plaintext, the *PBKDF2* column shows the number of hash function calls when PBKDF2 is configured to use 100K iterations, the *Without ANT* column is the attack cost in cryptographic operations when no ANT is used, and *With ANT* is the attack cost when an ANT is used. The *Ratio* column is the slowdown factor (with/without ANT).

PBKDF2 Iterations	Plaintext size	Ratio
5K	100KB	1.62
	1 MB	7.24
	100 MB	625.93
10K	200KB	1.62
	1 MB	4.12
	100 MB	313.48
50K	1 MB	1.62
	10 MB	7.25
	100 MB	63.5

Table 3: Sample plaintext sizes where the ratio from Table 4 is 1.5 or greater, for three sample PBKDF2 iteration counts.

The default PBKDF2 iteration count is 64K[3], therefore the slowdown factor when using an ANT is only 1.00049. The additional 64 AES decryptions are negligible compared to the cost of the 128K hash computations.

Password re-use A deployment challenge of the ANT approach is that it is sensitive to password re-use, and this can't easily be communicated to users. Once a small file is encrypted with a password, all files encrypted with the same password can be attacked with the same cost as the small ciphertext, regardless of their size. If a minimum plaintext size cannot be ensured (say by padding), the ANT technique is at best an opportunistic defense (meaning it only provides additional security if all files happen to be large).

Dynamic KDF workfactor The overall (attacker) workfactor of a password-based encryption operation is the the sum of the costs of the KDF and decryption steps. With the ANT technique, it's possible to reduce the workfactor of the KDF, while keeping the overall workfactor constant. For example, a large plaintext would be encrypted with a relatively cheap KDF step, as the encryption/decryption step keeps the overall workfactor high. Conversely, with small plaintexts the decryption workfactor is low, so the KDF workfactor must be large.

5 Acknowledgments

Thanks to Tolga Acar, Josh Benaloh, Marsh Ray and Dan Shumow for helpful discussions about this idea.

References

[1] R. Anderson and E. Biham. Two practical and provably secure block ciphers: BEAR and LION. *Proceedings of FSE'96, LNCS* **1039** (1996), 113-120.

[2] M. Bellare, P. Rogaway. Optimal Asymmetric Encryption. *Proceedings of EUROCRYPT'94, LNCS* **950** (1994), 92–111.

[3] V. Boyko. On the Security Properties of OAEP as an All-or-Nothing Transform. *Proceedings of CRYPTO'99, LNCS* **1666** (1999), 503–518.

[3] As of October 2014.

[4] Microsoft Corporation. *MS-OFFCRYPTO: Office Document Cryptography Structure*. Revision 3.0, April 2014. Available online `http://msdn.microsoft.com/en-us/library/cc313071%28v=office.12%29.aspx`

[5] M. Marlinspike. The Cryptographic Doom Principle. December 2011, accessed October 2014. Available online `www.thoughtcrime.org/blog/the-cryptographic-doom-principle`

[6] B. Kaliski. PKCS #5: Password-Based Cryptography Specification. IETF RFC 2898, version 2.0, September 2000. `www.ietf.org/rfc/rfc2898.txt`

[7] C. Percival. Stronger key derivation via sequential memory-hard functions. Presented at BSDCan'09, May 2009 (2009). Available online `www.daemonology.net/papers/scrypt.pdf`.

[8] R. Rivest. All-or-Nothing Encryption and the Package Transform. *Proceedings of FSE'97* (2009).

[9] Tarsnap. The scrypt encryption utility. `www.tarsnap.com/scrypt.html`

[10] Zetetic. SQLCipher. `http://sqlcipher.net`

[11] SQLite. `http://sqlite.org`

[12] D. Stinson. Something about all or nothing (transforms). *Designs, Codes and Cryptography* **22** (2001), 133-138.