Whitenoise Laboratories Inc.

Software Specifications

For

Tinnitus Utilizing Whitenoise Substitution Stream Cipher (Revised)

Written by
Stephen Boren email: sboren@bsbutil.com
Andre Brisson email: brisson@bsbutil.com

Created: 6/24/2002 By: Stephen Boren
Last Modified: 2/24/2004 By: Stephen Boren

TABLE OF CONTENTS

Abstract

This paper introduces an iteration of a new generation stream cipher called Whitenoise and more specifically how it is implemented in an application called Tinnitus. Initial research supports that in its base state Whitenoise is among the fastest stream ciphers known while maintaining a very high security level when deployed from software. Whitenoise holds out great potential for significant speed increases when deployed from hardware or a chip, and the speed of Whitenoise will be readily scalable upwards because of the ability to optimize its performance. Optimization includes pre-processing and pre-caching. The Whitenoise bit independent architecture also holds out great promise for dramatic speed improvements because of its ability to be deployed in multiple channels with many parallel threads encrypting, transmitting, and decrypting simultaneously. Testing also indicates that the level of randomness generated is among the highest levels of entropy generated. A Security Evaluation of Whitenoise is posted at http://eprint.iacr.org/2003/218/.

By utilizing a two-byte draw on the substitution, a one-way function is created. This iteration of Whitenoise utilizes an s[65536].

This is a large substitution box with 65536 slots for the full 16 bit (2 bytes) addressing, but only outputs 1 byte or 8 bits for one slot. This makes for 256 outputs that would be the same value and 256 different values but would be arranged randomly (each value 0 to 255 would appear 256 times) to completely delinearize the output stream. To improve the data to noise ratio, the output is XORed with a third Whitenoise generated byte. This improved cipher was revised based on the attack given at http://eprint.iacr.org/2003/250/.

1. Keywords:

secret-key cryptography, cryptanalysis, stream cipher, Whitenoise

2. Summary of design goals

It was desirable to design a symmetric stream cipher algorithm that would be very flexible, very fast, very random, and very secure. It was desirable to be able to rapidly create key streams of specific sizes and to be able to make and rapidly utilize keys of enormous size. It was necessary that the key streams always be larger than the data encrypted, that the offsets and counters had variable starting points and that identical sequences were never used more than once. Rapid key generation is desirable to overcome the difficulties of identifying primes and prime composites attendant with other encryption techniques. The ability to jump to any part of a stream is useful. Finally, it was desirable to have an algorithm that could be used in varied contexts. For instance, the algorithm can be used either in a Public Key Infrastructure or in an independent context. Overall, the goal was to create a cipher that would mimic multi-dimensionalism in a two dimensional context.

3. Algorithm Description

Whitenoise Substitution Stream Cipher is a multi-key-Super key hierarchical cryptographic process. This cryptographic system utilizes a method of encryption that can reasonably be described conceptually as an algorithmic representation of a multi-dimensional encrypting cipher matrix. The Whitenoise algorithm takes several sub keys and then creates a very long non-repeating key stream.

The resulting key stream is then put through a 2 byte substitution cipher that returns only 1 byte which is then XORed with a 3rd byte, the output of which is used to randomize the plaintext.

The Whitenoise component. The mechanism for computing the "SuperKey" from the "sub keys" works as follows. Let $s_j^{\{i\}}$ denote the *j*-th byte of the *i*-th "sub key". Let $\ell^{\{i\}}$ denote the length of the *i*-th "sub key." For example, we might have $\ell^{\{1\}}$ = 13, $\ell^{\{2\}}$ = 17, and so on. Create from the "sub key" *i* the unending sequence of bytes

$$s_0^{\{i\}}, s_1^{\{i\}}, s_2^{\{i\}}, \cdots, s_{(\ell^{\{i\}}-1)}^{\{i\}}, s_0^{\{i\}}, s_1^{\{i\}} \cdots$$

Let $s_j^{\{i\}}$ denote the *j*-th byte of the above sequence, if *j* is any number 0 to ∞; or, in other words, we implicitly reduce the subscript of $s_j^{\{i\}}$ modulo $\ell^{\{i\}}$. Then, the *j*-th byte of the "SuperKey," call it Z_j, is defined by

$$z_j = s_j^{\{1\}} \oplus s_j^{\{2\}} \oplus \cdots \oplus s_j^{\{n\}}$$

Here, "$\oplus$" denotes the XOR operation. In other words, to be more explicit,

$$z_j = s^{\{1\}}_{j \bmod \ell^{\{1\}}} \oplus s^{\{2\}}_{j \bmod \ell^{\{2\}}} \oplus \cdots \oplus s^{\{n\}}_{j \bmod \ell^{\{n\}}}$$

Where j mod $\ell^{\{i\}}$ returns an integer in the range 0,1,2,…, $(\ell^{\{i\}} - 1)$

The "SuperKey" has a j value that ranges from 0 to $((\ell^{\{1\}} \times \ell^{\{2\}} \times \ell^{\{3\}} \cdots \ell^{\{n\}}) - 1)$.

Let $P_0,P_1,P_2,P_3,..$ be the bytes of the plaintext, and $C_0,C_1,…$ the bytes of the ciphertext, in order. Also, $z_0,z_1,…$ denotes the bytes of the "SuperKey" (computed as already described in Section 1).
The Substitution portion is used to delinearize the above stream. This is done by having a substitution using two bytes of the Superkey stream that are used to index the full 65536 array. This array is a random scrambling of values 0 to 255, 256 of each value.

We define the ciphertext by $C_i := P_i$ xor $S[z_{(i-10)}*256 + z_{(i-3)}]$ xor z_i. The ciphertext C is formed by concatenating the bytes $C_0,C_1,..,$and then C is returned as the result of the encryption process.

Decryption works in reverse in the obvious manner.

4. Algorithm Implementation

The interface to Whitenoise Substitution is as follows. The input to the stream cipher is two seed values, used to create the key. In this implementation this is done during fob creation and never done by the user.

ENCRYPTION INTERFACE:
  INPUTS:
    a key K = created by (seed1, seed2)
    a big number counter T(set by key creation and updated as it is used.
    a L-bit plaintext P, for some L
  OUTPUT:
    a L-bit ciphertext C, which is the encryption of P

DECRYPTION INTERFACE:
  INPUTS:
    a key K = created by (seed1, seed2)
    a big number counter T(same as above)
    a L-bit ciphertext C, for some L
  OUTPUT:
    a L-bit plaintext P, which is the decryption of C

The counter T is set during key creation for the first message, then incremented by the size of the plaintext for the next message, and so on, for the third message, …etc.

In this implementation, the user is the only one using the encrypt/decrypt, so no keys need to be transmitted.

We can assume that the counter T is known to any eavesdropper/thief who can acquire all ciphertexts. The attacker cannot exert any influence over T; T will always simply

count from starting point on up. The counter value T might not appear in the actual programming API for any real implementation of Whitenoise Substitution-- for instance, the Whitenoise Substitution implementation might instead be stateful and implicitly maintain T as part of its private state -- but the effect is the same, and this way we can specify Whitenoise Substitution as though it were a stateless, deterministic mathematical function.

The key K is pair of values, termed seed1 and seed2. Seed1 is a N-digit (our big number library uses char digits to do limitless math, and it is only necessary to preset the offsets) value chosen secretly, uniformly at random, independently of everything else, and never disclosed. Seed2 is a 32-bit value chosen secretly, uniformly at random, independently of everything else, and never disclosed. The value N is a security parameter, and will typically be in the range of 500-700 digits or 1600-2400 bits. Any choice in this range should lead to acceptable security. The value N may be safely made public, without endangering the security of the system, and it might be the same for all users (for instance, it might be hard-coded in the software). All security resides in the secrecy & unpredictability of the key K.

Whitenoise Substitution encryption is decomposed into two components:
(1) key setup, and (2) output generation. The interface to key setup is as follows:

KEY CREATION INTERFACE:

```
INPUTS:
  a key K = (seed1, seed2)
OUTPUTS:
  an integer n, in the range 10,11,..,30
  a list (l^1,l^2,..,l^n) of n lengths
  a list (s^1,s^2,..,s^n) of n sub keys
  a big number counter T
  an array S[65536] of bytes
```

The integer n is the number of sub keys. Each length l^i of the first 10 is a prime number in the range 2,3,5,.., 15991 (there are 1862 different primes in this range), and l^i represents the length in bytes of the i^{th} sub key, and the remainder of the subkeys are of the range 2,…,16000 with no prime requirement but stipulated that it is not divisible by a previous size. Each sub key s^i is l^i bytes long. S[] is the substitution; it is a permutation on bytes. We will sometimes write s^i_j to denote the j^{th} byte of s^i, for j in the range 0,1,..,l^i-1. All the outputs of the key setup phase must be kept secret, and they may be different for each message enciphered.

The interface to output generation is as follows:

OUTPUT GENERATION INTERFACE:

```
INPUTS:
  an integer n, in the range 10,11,..,30
  a list (l^1,l^2,..,l^n) of n lengths
  a list (s^1,s^2,..,s^n) of n sub keys
  a big number counter T
  an array S[65536] of bytes
```

```
    a L-bit plaintext P, for some L
  OUTPUT:
    a L-bit ciphertext C, which is the encryption of P
```

Whitenoise Substitution is obtained by plugging the implementation of the key setup phase into the implementation of the output generation phase. To fully specify Whitenoise Substitution, it suffices to separately specify each of these two phases.

KEY CREATION ALGORITHM:

1. Treat seed1 as the decimal representation of an integer in the range of 500-700 digits.
2. Let X := seed1
3. Let Y := $\sqrt{X}$ is the irrational number generated by square rooting X
4. Let Z_1, Z_2, Z_3, Z_4, .. be the digits after the decimal point in the decimal representation of Y. Each Z_i is in the range 0,..,9.
5. Call srand(seed2). // only the first time
6. Call rand() to get the irrational starting point, start.
7. Let start := rand() mod 100. start is in the range 0,1,..,99.
8. Throw away Z_1 and Z_2 all the way to Z_{start}.
9. Let tmp := $10*Z_{(start+1)} + Z_{(start+2)}$. Throw away those used values.
10. Let n := 11 + (tmp mod 20) and n is in the range 11,12,..,30.
11. For i := 1,2,..,10, do:
12. Let j = 4*(i-1)
13. Let tmp be the next byte from the Z stream.
14. Let tmp := $1000*Z_{j+1} + 100*Z_{j+2} + 10Z_{j+3} + Z_{j+4}$
15. Let t := 1862 - (tmp mod 1862) and t is in the range 1,2,.., 1862.
16. Let u be the t^{th} prime among the sequence 2,3,5,.., 1862.
17. If u is equal to any of I^1, I^2, .., $I^{\{i-1\}}$, set t to (t+1)mod 1862 goto 16
18. Set I^i = u.
19. Next I : goto 11 until all 10 subkey sizes are set.
20. Then get remainder of lengths
21. For i := 11,..,n, do:
22. Let j = 5*(i-1)
23. Let tmp be the next byte from the Z stream.
24. Let tmp := $10000*Z_{j+1} + 1000*Z_{j+2} + 100Z_{j+3} + 10*Z_{j+4} + Z_{j+5}$
25. Let t := 16000 - (tmp mod 16000). t is in the range 1,2,..,16000.
26. Let u be the t.
27. If u is divisible by any of I^1, I^2, .., $I^{\{i-1\}}$, (or u is not co-prime) set t to (t+1)mod 16000 goto 26
28. Set I^i = u.
29. Next I : goto 21 until all subkey sizes are set.
30. For i := 1,2,..,n, do:
31. For j := 0,1,2,..,I^i, do:
32. Let k := 4*j
33. Let tmp be the next byte from the Z stream.
34. Let tmp := $(1000*Z_k + 100*Z_{k+1} + 10*Z_{k+2} + Z_{k+3}) \bmod 256$
35. Let s_j^i := tmp
36. Next j : Next subkey byte
37. Next I : Next subkey

38. For i := 0,1,2,..,65535, do:
39. Set S[i] := Int(i / 256) (integer value only, truncated)
40. Next i: initialize S-box to 256 of each value
41. For i := 0,1,2,..,65535, do:
42. Let j := 4*i
43. Let tmp := $(10000*Z_j + 1000*Z_{j+1} + 100*Z_{j+2} + 10*Z_{j+3} + Z_{i+4})$ mod 65536
44. Set tmp2 := S[i]
45. Set S[i] := S[tmp].
46. Set S[tmp] := tmp2.
47. Next i
48. Let offset := $Z_iZ_{i+1}…Z_{i+9}$
49. Return n, $(l^1,l^2,..,l^n)$, $(s^1,s^2,..,s^n)$, S[65536] and offset.
50. Save in keyfile and add seed1 and start value to DB
51. Increment seed1 and goto 2 //repeat until enough keys are created

It is possible to use srand() and rand() as they are only used to create the keys at the manufacturing stage and saving the resulting keys on the FOB. In a distributed system seed2 would just be used directly as only 1 key is made from the two seeds.

5. Key Creation Rules

Tinnitus is a personal security system deployed from a key FOB which you plug into your computer and which contains your key. You can encrypt all the files that you wish to secure. Therefore, there is no key distribution system required. It is not for use in any kind of broadcast situation. Without your FOB you cannot access the encrypted files.

Each FOB contains its own unique key (the business version will contain two such keys for use with a master key for limited communication abilities). The manner in which those keys are created is in the following definitions.

First the system must be seeded with two random seed values. The first seed value will be in the range of 500 to 700 decimal digits. The second seed value is a 32-bit value simply used to seed the rand function. To create the keys the square root of the first value is used to create a pseudo random sequence to be used to create the key. The second seed value is used to initialize this using the srand function and rand is then used to set the starting offset (0 to 100). The next key is created using seed1 + 1 and the next rand call, and this is continued…(of course each seed1 is tested to make sure it is not a perfect square).

For Tinnitus, the maximum subkey size is 16000 bytes. This means there are 1862 different subkey lengths that follow the rule that the first ten subkey lengths must be prime and the subsequent subkeys greater than ten need only to be not divisible by any of the previous subkey lengths. The first number to be calculated is the number of subkeys to be used. This is calculated by taking the first two digits generated by our random stream. MOD this by 20 and add 11 to give the key structure of between 11 and 30 subkeys. Then use the following system repeatedly to generate each subkeys length. A 4-digit section is generated and then is MODed by 1862 then you take 1862 minus this value to choose the prime value. If this value is already used it takes the next available larger value. This is used to get the prime lengths, the remaining lengths are

then filled similarly but only must be co-prime. Then the subkeys are filled in turn byte by byte. The substitution cipher is then created by grabbing a byte in sequence to randomize it. Once the key is created it is saved into a file using the file format defined in the next section and the next key is started until the entire sequence of keys requested have been created.

For Tinnitus, there is no key distribution system. This is because the key is only being used for personal security on the users' computer. Therefore the key is created during the manufacturing process of each FOB and is unique for each device. For retrieval purposes the seed values used to create each key will be stored for authenticated retrieval in the event of a lost FOB. Each key is then stored on each FOB.

As each file is encrypted a new file named {OLDFILENAME}.wn, which has a small header added to handle versions and different keys (see file header section). The OLDFILENAME includes the extension to allow for easy decryption and maintaining the same file format for functionality. As each file is encrypted, it is immediately decrypted and compared to the original and then both the test copy and the original file are deleted using a clean sweep deletion process (the entire file rewritten as 0's and then 1's and then deleted).

6. Key File Format

The Tinnitus key file format is defined as below.

```
typedef struct wnkeyfiletype {
   char    fileid[2]; // must be WN to identify file format
   char    version;  // file type version number to allow changes
   ulong64  offset; // the offset is a large number stored as a
                    // string of decimal digits delineated by “”’s
   long    numsk; // the number of subkeys
   long    sklen[numsk]; // the individual subkey lengths
   char    sk1[sklen[1]]; // the 1st subkey
   char    sk1[sklen[2]]; // the 2nd subkey
   char    sk1[sklen[3]]; // the 3rd subkey
           …          …          …          …
   char    sknumsk[sklen[numsk]]; // the numskth subkey
   char    substit[65536]; // the substitution cipher key
} WNKEYFILE;
```

The Tinnitus Key file format is fairly standard.

7. Notes:

We would like to thank David Wagner and Hongjun Wu for all their invaluable input and advice.