Simple Pseudorandom Number Generator with Strengthened Double Encryption (Cilia)*

Henry Ng

Henry.Ng.a@gmail.com

Abstract. A new cryptographic pseudorandom number generator Cilia is presented. It hashes real random data using an iterative hash function to update its secret state, and it generates pseudorandom numbers using a block cipher. Cilia is a simple algorithm that uses an improved variant of double encryption with additional security to generate pseudorandom numbers, and its performance is similar to double encryption. Furthermore, cryptanalytic attacks are presented.

Keywords: Pseudorandom Number Generator, Block Cipher, Hash Function, Pseudorandom Function, Cryptanalysis.

1 Introduction

Cryptographic applications heavily rely on pseudorandom number generators (PRNG) to generate secrets such as session keys, passwords, and key pairs. They also require random numbers that are public such as salts. Random numbers are generated with the assumption that they are unpredictable to an attacker. As well, randomness is collected from real random data to seed a PRNG, and once seeded, PRNGs produce a long sequence of random numbers.

1.1 Objectives and Definitions of Cilia

A PRNG is an algorithm that generates numbers that appear random. These numbers should be indistinguishable from real random numbers to an attacker who has no knowledge of the PRNG secret state. PRNGs are normally constructed from primitives such as block ciphers, hash functions, and stream ciphers. In addition, the PRNG must update its secret state by collecting real random data or *samples* from entropy sources. To maintain and to achieve this secret state, the PRNG must constantly update itself by processing many samples. Samples are unpredictable to an attacker. Samples are input to the PRNG, and pseudorandom numbers are output from the PRNG.

Cilia is constructed using existing cryptographic primitives and it has several components. Cilia uses two cryptographic primitives: the encryption function E

* Cryptology ePrint Archive Report 2005/086; March 19, 2005; http://eprint.iacr.org/2005/086

of an n-bit block cipher with a k-bit key, and an m-bit iterative hash function h. $m = 2k$ is assumed. A hash function and a block cipher are used because they are well studied and widely available. The *generation mechanism* generates the pseudorandom numbers or *outputs* using the secret state and E. Samples are stored in an *entropy pool*. Cilia uses h to hash the samples in the entropy pool with the secret state or to *reseed* to update the secret state. These components and primitives form Cilia.

Cilia has been designed with a few goals. Attacks on PRNGs such as the ANSI X9.17 PRNG, the DSA PRNG, the RSAREF PRNG, and CryptoLib were described in [8]. The PRNGs that were analyzed in [8] have security weaknesses that the Yarrow PRNG [7] attempts to prevent by addressing all known attacks. Cilia attempts to resist against all known attacks also. Some comparisons are made between Cilia and Yarrow [7], which is a generally well-designed and well-known PRNG, throughout this paper. However, Yarrow uses single block cipher encryption with a k-bit key to generate outputs and it inputs samples to two entropy pools. The design of the Yarrow entropy pools is generally complex, and the security of the Yarrow pseudorandom number generation could be improved. Cilia, on the other hand, attempts to generate pseudorandom numbers that are less distinguishable from true random numbers and to utilize a simpler entropy pool.

1.2 The Rest of This Paper

In this paper, section 2 describes Cilia. Cryptanalytic attacks on Cilia are presented in section 3. Performance tests are presented in section 4. Test vectors are available in appendix A.

2 Description of Cilia

Initialize, AddSamples, and GetOutputs functions respectively initialize, add samples to the pool, and retrieve outputs. GetOutputs utilizes GenerateBlocks and Reseed functions to generate outputs and to reseed, respectively. These simple functions form Cilia.

Public: Initialize

Output: state S.

//Set the counters C_1, C_2, the keys K_1, K_2, and the reseed flag R to zero.
//Set the pool P to the empty string.
$(K_1, K_2, C_1, C_2, R, P) \leftarrow (0, 0, 0, 0, 0, \epsilon)$
//Package the state S.
$S \leftarrow (K_1, K_2, C_1, C_2, R, P)$
return S

Public: AddSamples

Input: state S; samples s ($s \neq \epsilon$).

//Add samples to the entropy pool P by appending.
$P \leftarrow P \parallel s$

Private: GenerateBlocks

Input: state S ($R = 1$); number of blocks to generate j.
Output: string of generated blocks g.

//Set the string of generated blocks to empty string.
$g \leftarrow \epsilon$
//Generate blocks.
for $i = 1$ to j **do**
  $g \leftarrow g \parallel E_{K_1}(C_1) \oplus E_{K_2}(C_2 \oplus E_{K_1}(C_1))$
  $C_1 \leftarrow C_1 + 1 \bmod 2^n$
  if $C_1 = 0$ **then**
    $C_2 \leftarrow C_2 + 1 \bmod 2^n$
  end if
end for
return g

Private: Reseed

Input: state S.

//Hash the samples with the current keys to create two k-bit halves for the keys.
$(K_1, K_2) \leftarrow \textit{get-pair}(h(h(P) \parallel K_1 \parallel K_2))$
//Set the reseed flag to one to indicate that a reseed occurred.
//Set the pool to empty string.
$(R, P) \leftarrow (1, \epsilon)$

Public: GetOutputs

Input: state S; number of blocks to generate j ($2^{n/4} \geq j > 0$).
Output: string of generated blocks g.

//Reseed if the pool has more than $2k$ bits of samples.
if $\textit{get-bit-length}(P) > 2k$ **then**

RESEED(S)
end if
//*Generate blocks. The reseed flag must be set to one in order to generate blocks.*
$g \leftarrow$ GENERATEBLOCKS(S, j)
//*Change keys by using two k-bit halves of the first 2k bits of the generated blocks.*
$(K_1, K_2) \leftarrow$ *get-pair*(*get-first-2k-bits*(GENERATEBLOCKS(S, $\lceil 2k/n \rceil$)))
return g

The GETOUTPUTS function handles output requests. Two counters are used in GENERATEBLOCKS because they provide an output cycle of 2^{2n}. K_1 specifies one permutation on 2^n ciphertext blocks using $E_{K_1}(C_1)$; however, C_2 and K_1 specify 2^n permutations on 2^n ciphertext blocks using $C_2 \oplus E_{K_1}(C_1)$. Figure 1 shows the generation mechanism. When the entropy pool contains more than $2k$ bits of samples, a reseed occurs. The RESEED function combines the entropy pool with the current keys to generate new keys because this requires the attacker to know the old keys in order to know the new keys even if the samples are known [7]. New keys are generated after handling the output requests. Changing the keys in this fashion prevents compromise of past outputs that are generated under past keys if the current keys are compromised; this is forward security [4]. These are the features of GETOUTPUTS.

2.1 Generation Mechanism

$E_{K_1}(C_1) \oplus E_{K_2}(C_2 \oplus E_{K_1}(C_1))$ is a pseudorandom function (PRF) in the GENERATEBLOCKS function. Under a fixed key K, a n-bit block cipher is a pseudorandom permutation (PRP) over $\{0,1\}^n$ [10]. Using two independent PRPs p_1, p_2 over $\{0,1\}^n$, a PRF is defined by XORing two PRP results $p_1(x) \oplus p_2(x)$ [10].

Theorem 1. *The random permutation p over $\{0,1\}^n$ is a $(q, q^2/2^{n+1})$-secure PRF.*

Theorem 2. *The function $f(x) = p_1(x) \oplus p_2(x)$ is a $(q, q^3/2^{2n-1})$-secure PRF, where p_1, p_2 are independent random permutations over $\{0,1\}^n$ and $q \leq 2^{n-1}$.*

An attacker who makes at most q queries from a (q, a)-secure PRF can get at most advantage or chance a for distinguishing the (q, a)-secure PRF from an ideal PRF. An ideal PRF is a $(\infty, 0)$-secure PRF. It is assumed that block ciphers are ideal PRPs. Some block cipher based PRNGs use single encryption to generate outputs. The attacker can easily distinguish it because $q^2/2^{n+1} \lessapprox 1$ if $q \gtrapprox 2^{n/2}$. A collision is expected in about $2^{n/2}$ output blocks; however, PRPs produce no collisions. Although the $(q, q^3/2^{2n-1})$-secure PRF is not ideal, it is better than the $(q, q^2/2^{n+1})$-secure PRF because $0 < q^3/2^{2n-1} < q^2/2^{n+1}$ if $q > 0$. This is a slight security improvement. For Cilia, $2^{n/4} \geq q > 0$ queries can be made such that $q^3/2^{2n-1} \gtrapprox 0$, which is close to the ideal PRF with no advantage for the attacker. In this paper, assume that getting any n-bit output

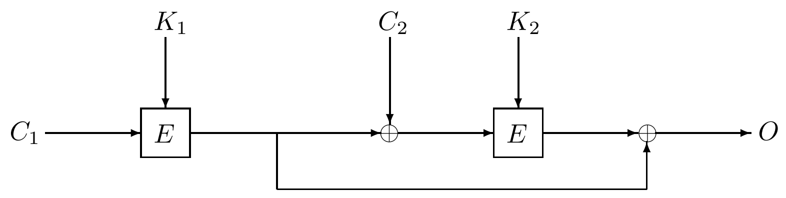

Fig. 1. The Cilia generation mechanism.

block from Cilia has 2^{-n} chance with negligible deviations. Security proofs of the theorems are available in [10].

Yarrow uses single encryption in counter mode as its generation mechanism [7]. It has a $(q, q^2/2^{n+1})$-secure PRF. Also, it has $k/2$ bits of security against key-collision attacks [2], and has k bits of security against exhaustive key search. It is conjectured that the Cilia generation mechanism is better.

There are other constructions that form a good PRF and are better than the $(q, q^2/2^{n+1})$-secure PRF. Post processing the output blocks with a hash function's compression function described in [5] is a possible technique to form a PRF. Except, $E_{K_1}(C_1) \oplus E_{K_2}(C_2 \oplus E_{K_1}(C_1))$ is generally faster than a hash function's compression function. Furthermore, frequent data-dependent re-keying of the block cipher is another technique of a PRF, $E_{E_K(x)}(x)$ [3]. But, frequent re-keying of the block cipher would repeatedly run the key schedule for every output block and this is usually computationally expensive in comparison to encryption where the key remains the same for multiple outputs. Besides, in contrast to efficient encryption, a design technique for block ciphers is to make re-keying computationally expensive [11]. Furthermore, $E_{K_1}(x) \oplus E_{K_2}(x)$ is a natural method for a PRF [3]. This simple technique $E_{K_1}(x) \oplus E_{K_2}(x)$ is not used because a constant stream of zeros are generated if $K_1 = K_2$, and therefore, the technique prohibits 2^k pairs of equal keys out of the 2^{2k} key pairs from keying the block cipher. $E_{K_1}(x) \oplus E_{K_2}(x)$ does not have a key space of 2^{2k}. Additionally, truncating a few bits of the n-bit ciphertext block of the underlying block cipher can be used to build a PRF [6]. This technique is nearly as fast as the underlying block cipher encryption unlike most other PRF constructions, which are generally slow. Single encryption has $k/2$ bits of theoretic strength [2] and k bits of security against exhaustive key search. However, the objective is to provide more security than single encryption with a k-bit key. Also, most block ciphers cannot extend their key lengths easily, except block ciphers such as AES [13]. Also, more truncated output blocks need to be generated to complete output requests in multiples of n bits. The Cilia generation mechanism appears to be sufficient for generating random sequences considering both speed and security.

2.2 Entropy Pool

Yarrow uses two entropy pools [7]. The two pools are the *fast and* slow pools. A portion of the samples form one string of samples for the fast pool, and another portion of the samples form another string of samples for the slow pool. Each pool is a separate running hash of a portion of the samples. The fast pool ensures that key compromises have a possibility of a short duration by providing frequent reseeds. Also, the slow pool ensures that Yarrow eventually does a secure reseed even though the pools may have very optimistic entropy estimates. Yarrow reseeds from either the fast pool or the slow pool.

For simplicity, Cilia uses only one pool. This should reduce the possibility of making mistakes during implementation, and the rate of inputting samples should be greater. In addition, simpler designs should encourage more cryptanalysis.

3 Cryptanalytic Attacks on Cilia

3.1 Iterative Guessing Attack

For an iterative guessing attack, the inputs between time t and $t+\epsilon$ are guessable [8]. The objective is to extend a successful compromise of S at time t to future outputs at time $t+\epsilon$.

Attack 1. Assume only x bits of entropy are inputted between time t and $t+\epsilon$, where x is small $x < k$ (this makes the attack appear more practical). Assume that the other entropy sources produce known inputs that are observable. Therefore, all other inputs are known between time t and $t+\epsilon$. The attacker successfully compromises S at time t and extends the compromise to time $t+\epsilon$ by doing a 2^x search. The outputs generated between time t and $t+\epsilon$ are compared to the search in order to conclude (K_1, K_2) at time $t+\epsilon$.

3.2 Backtracking Attack

The backtracking attack uses the compromise of S at time t to learn outputs at time $t-\epsilon$ [8]. The objective is to extend a compromise to previous outputs.

Attack 1. S is compromised at time t, and the attacker learns that the most recent reseed was at time $t-\epsilon$. At most $2^{n/4}$ outputs blocks are compromised between $t-\epsilon$ and t since the counters C_1, C_2, can be decremented with negligible effort. Also, at most $2^{n/4}$ outputs from time t to $t+\epsilon$ can be easily learned by incrementing C_1, C_2. However, if the next reseed occurs at time $t+\epsilon$, the attacker then loses knowledge of (K_1, K_2) at time $t+\epsilon$.

Attack 2. It is possible to backtrack past outputs by observing future outputs and comparing them to some past outputs. This attack is feasible when the counters produce a short 2^{2n} cycle. It is possible to compromise past (K_1, K_2) given future (K_1, K_2) when C_1, C_2 contain unknown values.

The attacker finds (K_1, K_2) without a direct attack on the generation mechanism. Assume that an ordered sequence of outputs is watched starting when $C_1 =$

x_1 and $C_2 = x_2$ at time $t-\epsilon$. The attacker waits until time t a total of 2^{2n} outputs when C_1 and C_2 has cycled back to $C_1 = x_1$ and $C_2 = x_2$. The output sequence at t is also observed. Let (K_1', K_2') denote (K_1, K_2) at time $t-\epsilon$, and let (K_1'', K_2'') denote (K_1, K_2) at time t. If the sequence $O^{(1)}, \ldots, O^{(\lfloor 2k/n \rfloor + 1)}$, at time $t - \epsilon$ equals the sequence $O^{(1)}, \ldots, O^{(\lfloor 2k/n \rfloor + 1)}$, at time t, $(K_1', K_2') = (K_1'', K_2'')$ is expected with $\Pr[(K_1', K_2') \neq (K_1'', K_2'')] = 2^{2k-n(\lfloor 2k/n \rfloor + 1)}$ ($\lfloor 2k/n \rfloor + 1$ is the unicity distance that is required to make it improbable that those keys do not equal). When (K_1, K_2) is known at time t, (K_1, K_2) is known at time $t - \epsilon$ for this case. Therefore, past outputs are backtracked.

3.3 Input-Based Attack

The attacker uses knowledge or control of the samples to attack a PRNG when doing an input-based attack [8]. A hash function h processes the samples in the pool, and it may be correct to assume that h would securely process the samples preventing input-based attacks. However, along with timing analysis, the input-based attack is effective.

Data-dependent execution paths open a timing channel that leaks information to an attacker [9]. The attacker monitors the precise timings of the compression operations when samples are appended to P. The compression operations of an iterative hash function will leak how many l-bit blocks of samples are compressed by how much time the operations require. Using the timing attack, the attacker determines the input rate of an unknown entropy source. With information about the input rate, a permanent compromise attack is attempted.

Attack 1. Assumptions are made to simplify the attack. Assume that the iterative hash function parses inputs into l-bit message blocks, and the compression function of the iterative hash function operates on the l-bit message blocks individually. An x-bit input produces $\lfloor x/l \rfloor$ l-bit message blocks with the remaining $x \bmod l$ input bits handled by a padding rule and with those remaining inputs compressed during hash finalization. SHA-256 is one example of this [12]. Assume the samples are parsed and compressed on the fly as they are appended to P (generally, practical implementations have a running hash of all the samples). The number of compression operations is watched through a timing channel, and the attacker can count the number of calls to AddSamples. There are two entropy sources e_1, e_2. Assume that the strings of samples from e_2 all have one common fixed length x_1, and one string of samples from e_2 is appended to P after every one common fixed time interval x_2. One string of samples with x_1 bits and x_1 bits of entropy is appended to P after every x_2 seconds from e_2. $x_1 > 0$ and $x_2 > 0$. Also, assume that the attacker has full control of e_1. These assumptions are made to simplify the presentation.

The attacker determines the rate at which the entropy is accumulated from e_2. Later, with the knowledge of the rate, a permanent compromise of (K_1, K_2) is attempted. The first objective is to solve for x_1, x_2. This works as follows:

1. Force a reseed by inputting more than $2k$ sample bits and requesting an output. Now, P is empty.

2. Wait until c_1 compression operations that are triggered by one string of samples from e_2 that is appended to P is detected, where $c_1 > 0$. Since one string of samples from e_2 that is appended to P requires one call to AddSamples, the attacker knows when the wait for c_1 compression operations finishes. Record the time t' seconds required for the wait. Keep count of the number of one string samples c_2 from e_2 that are appended to P during the t' seconds.
3. For $i = 1$ to $l/8$, do:
   (a) Input more than $2k$ sample bits and request an output. Now, P is empty. Append i arbitrary bytes to P. If one compression operation was detected, then $c_3 \leftarrow 1$, else $c_3 \leftarrow 0$. Assume this requires no time to execute.
   (b) Wait for c_2 strings of samples from e_2 to append to P, and sum the number of compression operations c_4 that occurred during this wait with c_3, $c_3 \leftarrow c_4 + c_3$.
   (c) If $c_3 = c_1 + 1$, break the loop.
4. The attacker learns that $b = c_3 \cdot l - 8i$ bits of entropy accumulates every t' seconds, where i is the i-th iteration when breaking the loop in step 3c. Finally, the attacker learns that one string of samples with $x_1 = b/c_2$ bits of entropy is appended to P after every $x_2 = t'/c_2$ seconds from e_2.

3.4 Permanent Compromise Attack

The permanent compromise attack uses the compromise of S at time t to learn all past or future S [8]. The objective is to extend a compromise to all past or future outputs.

Attack 1. The following attack is a continuation of the input-based attack (described above), and it is a simple example of doing a permanent compromise attack given knowledge of the input rate of e_2. Assume that the attacker compromised S immediately after a reseed occurred. To make this attack appear practical, assume that the attacker knows that one string of samples with x_1 bits and x_1 bits of entropy is appended to P after every x_2 seconds from e_2, where $x_1 = 8$ bits, $x_2 = 60$ seconds. Executing the permanent compromise attack works as follows:

1. Input $2k$ bits to the pool using e_1. These inputs are known.
2. Wait x_2 seconds for e_2 to append one string of samples to P. Then, force a reseed by requesting an output O. Now, P is empty.
3. The current (K_1, K_2) has x_1 bits of entropy. Do a 2^{x_1} search to recover the current (K_1, K_2). Verify the search by comparing with the output block O.
4. Redo steps 1 to 3 for every reseed. Since the same search is carried for all future keys with small effort, the attacker permanently breaks Cilia—it should never recover.

3.5 Meet-in-the-Middle Attack

Using the meet-in-the-middle attack, the attacker recovers S at $t+\epsilon$ with knowledge of S at time t and $t+2\epsilon$ [8]. However, it is unlikely that this attacks is useful because (K_1, K_2) and P are processed with h, which is one-way. If h has x unknown input bits, then about 2^x work is required to find the unknown bits.

In addition, meet-in-the-middle attacks apply to multiple encryption. Although $E_{K_1}(C_1) \oplus E_{K_2}(C_2 \oplus E_{K_1}(C_1))$ is similar to double encryption (multiple encryption), it is unlikely that a meet-in-the-middle attack is useful since the plaintext $E_{K_1}(C_1)$ for E_{K_2} is XORed with the ciphertext from E_{K_2}.

3.6 Key Search Attack

To attack the generation mechanism, the attacker searchs for (K_1, K_2). Once (K_1, K_2) is recovered, the attacker learns all outputs generated under (K_1, K_2) until Cilia accumulates enough entropy to reseed and to recover from the compromise.

For simplicity, assumptions are made when attacking the generation mechanism. The generation mechanism is similar to Davies Meyer hashing mode $x \oplus E_K(x)$ [11], and block ciphers such as DES are insecure in Davies Meyer hashing mode [14]. Assume that the underlying block cipher is not weak when it is used in Davies Meyer hashing mode and it does not have other weak properties. Assume that C_1, C_2, and O blocks are freely available for all cases of attack. Furthermore, assume keys (K_1, K_2) are unchanging targets, except for a collision attack. Use these assumptions for the attacks on the generation mechanism.

Table T contains known counters and known outputs. For $i = 1$ to $\lfloor 2k/n \rfloor + 1$, T contains $(C_1^{(i)}, C_2^{(i)}, O^{(i)})$, where $C_1^{(i)}$ determines a unique tuple. These will be used for verifying a successful key search.

Attack 1. Do 2^{2k} steps. Use table T. This works as follows:

1. For $i = 1$ to 2^{2k}, do:
   (a) Choose keys K', K''. $(K_1, K_2) = (K', K'')$ is expected with $\Pr[(K_1, K_2) \neq (K', K'')] = 2^{2k-n(\lfloor 2k/n \rfloor+1)}$ if $\forall j$ $(E_{K'}(C_1^{(j)}) \oplus E_{K''}(C_2^{(j)} \oplus E_{K'}(C_1^{(j)}))$ $= O^{(j)})$, where $j \in \{1, \ldots, \lfloor 2k/n \rfloor + 1\}$. Assume this is one step of work.

Attack 2. The attacker guesses the $E_{K_1}(C_1^{(1)})$ ciphertext block. To find K_1, do $2^n \cdot 2^k = 2^{n+k}$ steps. $2^n \cdot 2^{k-n} = 2^k$ false key hits of K_1 are expected. For every possible K_1 key found, search for K_2. To find K_2, do an additional $2^k \cdot 2^k = 2^{2k}$ steps. Do $2^{2k} + 2^{n+k} \approx \max\{2^{2k}, 2^{n+k}\}$ steps in total. Use table T. This works as follows:

1. For $a = 1$ to 2^n, do:
   (a) Choose a ciphertext block B. Assume this requires negligible work and storage.
   (b) For $b = 1$ to 2^k, do:

i. Assume this is one step of work. Choose a key K'. If $B = E_{K'}(C_1^{(1)})$, then for $c = 1$ to 2^k, do:
   A. Choose a key K''. $(K_1, K_2) = (K', K'')$ is expected with $\Pr[(K_1, K_2) \neq (K', K'')] = 2^{2k-n(\lfloor 2k/n \rfloor+1)}$ if $\forall i(E_{K'}(C_1^{(i)}) \oplus E_{K''}(C_2^{(i)} \oplus E_{K'}(C_1^{(i)})) = O^{(i)})$, where $i \in \{1, \ldots, \lfloor 2k/n \rfloor + 1\}$. Assume this is one step of work.

Attack 3. The attacker guesses the $E_{K_1}(C_1^{(1)})$ ciphertext block. To find K_1, do $2^n \cdot 2^k = 2^{n+k}$ steps. To find K_2, do $2^n \cdot 2^k = 2^{n+k}$ steps. For every guessed $E_{K_1}(C_1^{(1)})$, 2^{k-n} false key hits of K_1 and 2^{k-n} false key hits of K_2 are expected. Possible keys are stored in the list L_1 for possible K_1 keys and the list L_2 for possible K_2 keys. Therefore, $2 \cdot 2^{k-n} = 2^{k-n+1}$ units of storage are required. With L_1, L_2, do $2^{k-n} \cdot 2^{k-n} = 2^{2k-2n}$ steps to test if the possible keys equal (K_1, K_2) for every guessed $E_{K_1}(C_1^{(1)})$. Do $2^n \cdot 2^{2k-2n} = 2^{2k-n}$ steps in total when keys in L_1, L_2 are used. Do $2^{2k-n} + 2^{n+k+1} \approx \max\{2^{2k-n}, 2^{n+k+1}\}$ steps in total to complete the attack. Use table T. This works as follows:

1. For $a = 1$ to 2^n, do:
   (a) Choose a ciphertext block B, and empty L_1, L_2. Assume this requires negligible work and storage.
   (b) For $b = 1$ to 2^k, do:
      i. Choose a key K'. If $E_{K'}(C_1^{(1)}) = B$, store K' into L_1. Assume this requires one step of work, and one unit of storage.
      ii. Choose a key K''. If $E_{K''}(C_2^{(1)} \oplus B) = B \oplus O^{(1)}$, store K'' into L_2. Assume this is one step of work, and requires one unit of storage.
   (c) Let $K'_1, \ldots, K'_{2^{k-n}}$, denote all keys in L_1. Let $K''_1, \ldots, K''_{2^{k-n}}$, denote all keys in L_2. $(K_1, K_2) = (K', K'')$ is expected with $\Pr[(K_1, K_2) \neq (K', K'')] = 2^{2k-n(\lfloor 2k/n \rfloor+1)}$ if $\forall b \forall c \forall d(E_{K'_b}(C_1^{(d)} \oplus E_{K''_c}(C_2^{(d)} \oplus E_{K'_b}(C_1^{(d)})) = O^{(d)})$, where $b, c \in \{1, \ldots, 2^{k-n}\}$ and $d \in \{1, \ldots, \lfloor 2k/n \rfloor + 1\}$. Assume this is $2^{k-n} \cdot 2^{k-n} = 2^{2k-2n}$ steps of work.

Attack 4. Using the key-collision attack [2], the attacker finds (K_1, K_2) with 2^{k+1} steps and 2^k storage. Assume C_1, C_2, are unchanging, outputs are all generated under random keys, and $n > 2k$. This attack is practical when the attacker frequently calls Initialize to freeze $C_1 = 0, C_2 = 0$. h has $m/2 = k$ bits of security against collision attacks. 2^k outputs are generated with random keys, and they are stored in a table. For each output request that the attacker makes, the output's existence in the table is checked. According to the birthday paradox, it is expected that one of the first 2^k requested outputs discloses a (K_1, K_2). This works as follows:

1. For $i = 1$ to 2^k, do:
   (a) Choose keys K', K'', randomly. Let $O' = E_{K'}(C_1) \oplus E_{K''}(C_2 \oplus E_{K'}(C_1))$. Store $(O', (K', K''))$ in the table T' indexed by O'. Assume this requires one step of work, and one unit of storage.

(b) Reset C_1, C_2. Request an output O. If O is in T' as an index with (K', K''), $(K_1, K_2) = (K', K'')$ is expected with $\Pr[(K_1, K_2) \neq (K', K'')] = 2^{2k-n}$. Assume this requires one step of work.

Table 1. The complexities of attacking the Cilia generation mechanism.

Steps	Storage	Attack model
2^{2k}	—	Exhaustive search
$2^{2k} + 2^{n+k}$	—	Exhaustive search; $E_{K_1}(C_1^{(1)})$ guess
$2^{2k-n} + 2^{n+k+1}$	2^{k-n+1}	Exhaustive search; $E_{K_1}(C_1^{(1)})$ guess; key lists
2^{k+1}	2^k	Key-collision; C_1, C_2 fixed; $n > 2k$

Additional Security Observations. According to Table 1, the most efficient attack against Cilia with two unknown k-bit keys should have a complexity of about 2^k. $n \geq k - 1$ should be a design requirement. If resetting C_1, C_2, is disallowed and $n \geq k - 1$, the most efficient attack on the generation mechanism should have a complexity of 2^{2k} instead of 2^k.

Also, the $2k$ bits of security of the generation mechanism is not like double encryption $E_{K_1}(E_{K_2}(x))$, and $E_{K_1}(E_{K_2}(x))$ is weaker in comparison because there is a known-plaintext meet-in-the-middle attack on $E_{K_1}(E_{K_2}(x))$ with a complexity of 2^k. Generally, attacking double encryption should have a complexity of 2^k (*with* or *without* a collision attack) and attacking the Cilia generation mechanism should have a complexity of 2^{2k} (*without* a collision attack). Although the complexities of attacking the Cilia generation mechanism can vary, the Cilia generation mechanism is generally *securer*.

Table 2 presents a summary of the attacks on double encryption and single encryption. See appendix B for step-by-step attack details.

Table 2. The complexities of attacking double encryption and single encryption.

Encryption	Steps	Storage	Attack model
Double	2^{2k}	—	Exhaustive search; $n > 2k$
Double	2^{k+1}	2^k	Meet-in-the-middle; $n > 2k$
Double	2^{k+1}	2^k	Key-collision; $n > 2k$
Single	2^k	—	Exhaustive search; $n > k$
Single	$2^{k/2+1}$	$2^{k/2}$	Key-collision; $n > k$

4 Performance

A C implementation of the generation mechanism runs at about the same performance as double encryption. It is about two times slower than single encryption. The tests used a C implementation of AES with $k = 128$ and $n = 128$ on a 1.82 GHz Pentium 4 with the Windows 98 operating system. K_1, K_2, were constant. C_1 was incremented for every output block, and C_2 was incremented if $C_1 = 0$. In Table 3, the Cilia generation mechanism is compared with double encryption and single encryption.

When inputting samples to the pool, the performance is expected to be identical to performance of the iterative hash function. Therefore, no tests were done.

Table 3. Performance comparisons.

Output Generation	MB/s
$E_{K_1}(C_1) \oplus E_{K_2}(C_2 \oplus E_{K_1}(C_1))$	26.2
$E_{K_1}(E_{K_2}(C_1))$	27.4
$E_{K_1}(C_1))$	53.5

5 Conclusion

The generation mechanism is a $(q, q^3/2^{2n-1})$-secure PRF assuming the block cipher is an ideal PRP. The n-bit outputs are less distinguishable in comparision to outputs from a $(q, q^2/2^{n+1})$-secure PRF.

Cilia is weak when there is insufficient entropy in the samples. System unique entropy sources such as configuration files, variable sources such as date and time, and external random sources such as keystroke timings can be generalized respectively to have low, medium, or high entropy [1]. To ensure that sufficient entropy is provided, high entropy samples from high entropy sources should be used only.

In addition, many entropy sources should be sampled to thwart the attacker. It is more difficult for the attacker to permanently break Cilia if samples are collected from many entropy sources. Thus, any information about the samples such as the rate at which samples are input is more difficult to determine.

The maximum number of outputs that can be requested by the user is $2^{n/4}$. If backtracking of past outputs is of high concern, this number should be lowered.

The 2^{2n} cycle is a minor issue if n is large. For example, if $n = 128$, the cycle would be 2^{256}, which is generally impossible for the attacker to wait. This attack is impractical for most contemporary n-bit block ciphers.

$n \geq k - 1$ is a requirement. If this is met, it is conjectured that the Cilia generation mechanism can provide $2k$ bits of security. It may be possible to achieve this level of security if C_1, C_2, do not reset.

Like other new designs, Cilia should not be used until it has received addtional cryptanalysis.

References

1. M. Atreya, "Pseudo Random Number Generators (PRNGs)," RSA Laboritories, `http://www.rsasecurity.com/products/bsafe/overview/Article4-PRNG.pdf`.
2. E. Biham, "How to Forge DES-Encrypted Messages in 2^{28} Steps," Technical Report CS884, Technion, August 1996.
3. M. Bellare, T. Krovetz, P. Rogaway, "Luby-Rackoff Backwards: Increasing Security by Making Block Ciphers Non-Invertible," *Advances in Cryptology—EUROCRYPT '98 Proceedings*, Springer-Verlag, 1998, pp. 266–280.
4. A. Desai, A. Hevia, and Y.L. Yin, "A Practice-Oriented Treatment of Pseudorandom Number Generators," *Advances in Cryptology—EUROCRYPT 2002*, Springer-Verlag, 2002, pp. 368–383.
5. P. Gutmann, "Software Generation of Practically Strong Random Numbers," USENIX Security Symposium, 1998.
6. C. Hall, D. Wagner, J. Kelsey, and B. Schneier, "Building PRFs from PRPs," *Advances in Cryptology—CRYPTO '98 Proceedings*, Springer-Verlag, 1998, pp. 370–389.
7. J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator," *Selected Areas in Cryptography—SAC '99 Proceedings*, Springer-Verlag, 1999, pp. 13–33.
8. J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic Attacks on Pseudorandom Number Generators," *Fast Software Encryption—FSE '98 Proceedings*, Springer-Verlag, 1998, pp. 168–188.
9. P.C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Advances in Cryptology—CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 104–113.
10. S. Lucks, "The Sum of PRPs Is a Secure PRF," *Advances in Cryptology—EUROCRYPT 2000 Proceedings*, Springer-Verlag, 2000, pp. 470–484.
11. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, "Handbook of Applied Cryptography," CRC Press, 1996.
12. National Institute of Standards and Technology (NIST), "FIPS Publication 180-2: Secure Hash Standard," August 2002.
13. National Institute of Standards and Technology (NIST), "FIPS Publication 197: Advanced Encryption Standard (AES)," November 2001.
14. B. Preneel, R. Govaerts, and J. Vandewalle, "Hash functions based on block ciphers: a synthetic approach," *Advances in Cryptology—Crypto '93 Proceedings*, Springer-Verlag, 1994, pp. 368–378.

A Test Vectors

The following tests used SHA-256 with $m = 256$ and AES with $k = 128$ and $n = 128$.

```
============
===Test 1===
============
```

At $t = 0$
$//K_1 =$ 00000000000000000000000000000000
$//K_2 =$ 00000000000000000000000000000000
$//C_1 =$ 00000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000

At $t = 1$
Input $s =$
000000000000000000000000000000000000000000000000000000000000000000

At $t = 2$
$//h(P1)\|K_1\|K_2 =$
;7f9c9e31ac8256ca2f258583df262dbc7d6f68f2a03043d5c99a4ae5a7396ce9
;0000000000000000000000000000000000000000000000000000000000000000
$//h(h(P_1)\|K_1\|K_2) =$
;6d816c62cacfa92a73fd4a3c91538257c472632e44af249872a150a77266dee3
$//K_1 =$ 6d816c62cacfa92a73fd4a3c91538257
$//K_2 =$ c472632e44af249872a150a77266dee3
$//C_1 =$ 00000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//O =$ 11696bd7fe7e3cb2bcb9e113b4dda6fe
$//C_1 =$ 01000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//O =$ a3c6d55719324bf878a1b676ff214218
$//C_1 =$ 02000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//O =$ 48252e5997198cc04cbf7adcb69389d5
$//C_1 =$ 03000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//K_1 =$ a3c6d55719324bf878a1b676ff214218
$//K_2 =$ 48252e5997198cc04cbf7adcb69389d5
Get $O =$ 11696bd7fe7e3cb2bcb9e113b4dda6fe

At $t = 3$
$//K_1 =$ a3c6d55719324bf878a1b676ff214218
$//K_2 =$ 48252e5997198cc04cbf7adcb69389d5
$//C_1 =$ 03000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//O =$ a2a65c2a3ae5c8cdc7987e6780901398
$//C_1 =$ 04000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//O =$ 86bf5b0f4253153d88b25c2e2b3c3f58

$//C_1 =$ 05000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//O =$ f32baf843af191d98acebff1020ae871
$//C_1 =$ 06000000000000000000000000000000
$//C_2 =$ 00000000000000000000000000000000
$//K_1 =$ 86bf5b0f4253153d88b25c2e2b3c3f58
$//K_2 =$ f32baf843af191d98acebff1020ae871
Get $O =$ a2a65c2a3ae5c8cdc7987e6780901398

============
===Test 2===
============
At $t = 0$
$//K_1 =$ 00000000000000000000000000000000
$//K_2 =$ 00000000000000000000000000000000
$//C_1 =$ feffffffffffffffffffffffffffffff
$//C_2 =$ fffffffffffffffffffffffffffffffe

At $t = 1$
Input $s =$
0000000000000000000000000000000000000000000000000000000000000000

At $t = 2$
$//h(P1)||K1||K2 =$
;7f9c9e31ac8256ca2f258583df262dbc7d6f68f2a03043d5c99a4ae5a7396ce9
;0000000000000000000000000000000000000000000000000000000000000000
$//h(h(P1)||K1||K2) =$
;6d816c62cacfa92a73fd4a3c91538257c472632e44af249872a150a77266dee3
$//K_1 =$ 6d816c62cacfa92a73fd4a3c91538257
$//K_2 =$ c472632e44af249872a150a77266dee3
$//C_1 =$ feffffffffffffffffffffffffffffff
$//C_2 =$ fffffffffffffffffffffffffffffffe
$//O =$ eeb20aa047023a584070cbbba6a6f908
$//C_1 =$ ffffffffffffffffffffffffffffffff
$//C_2 =$ fffffffffffffffffffffffffffffffe
$//O =$ 2a3a742cf180a57c9d6cb44ac6704feb
$//C_1 =$ 00000000000000000000000000000000
$//C_2 =$ 000000000000000000000000000000ff
$//O =$ 91372a7b1c179429e9cf232265669669
$//C_1 =$ 01000000000000000000000000000000
$//C_2 =$ 000000000000000000000000000000ff
$//K_1 =$ 2a3a742cf180a57c9d6cb44ac6704feb
$//K_2 =$ 91372a7b1c179429e9cf232265669669
Get $O =$ eeb20aa047023a584070cbbba6a6f908

At $t = 3$
$//K_1 =$ `2a3a742cf180a57c9d6cb44ac6704feb`
$//K_2 =$ `91372a7b1c179429e9cf232265669669`
$//C_1 =$ `01000000000000000000000000000000`
$//C_2 =$ `000000000000000000000000000000ff`
$//O =$ `8d0ca78ff7d60a49ce721de6b9bff74a`
$//C_1 =$ `02000000000000000000000000000000`
$//C_2 =$ `000000000000000000000000000000ff`
$//O =$ `ee45e9448943aa3734fa8adb8001873f`
$//C_1 =$ `03000000000000000000000000000000`
$//C_2 =$ `000000000000000000000000000000ff`
$//O =$ `ce644c2d1346b8b821d4977329cf04f6`
$//C_1 =$ `04000000000000000000000000000000`
$//C_2 =$ `000000000000000000000000000000ff`
$//K_1 =$ `ee45e9448943aa3734fa8adb8001873f`
$//K_2 =$ `ce644c2d1346b8b821d4977329cf04f6`
Get $O =$ `8d0ca78ff7d60a49ce721de6b9bff74a`

B Attacks on Double Encryption and Single Encryption

Full Exhaustive Key Search on Double Encryption. Let $y = E_{K_1}(E_{K_2}(x))$. The attacker finds (K_1, K_2) with 2^{2k} steps. For simplicity, assume $n > 2k$ so it is unlikely there is a false key hit. The plaintext-ciphertext pair (x, y) is known. This works as follows:

1. For $i = 1$ to 2^{2k}, do:
   (a) Choose keys K', K''. If $y = E_{K'}(E_{K''}(x))$, $(K_1, K_2) = (K', K'')$ is expected with $\Pr[(K_1, K_2) \neq (K', K'')] = 2^{2k-n}$. Assume this requires one step of work.

Known-Plaintext Meet-in-the-Middle Attack on Double Encryption. Let $y = E_{K_1}(E_{K_2}(x))$. A known-plaintext meet-in-the-middle attack on $E_{K_1}(E_{K_2}(x))$ requires 2^{k+1} steps and 2^k storage. For simplicity, assume $n > 2k$ so it is unlikely there is a false key hit. The plaintext-ciphertext pair (x, y) is known. This works as follows:

1. For $i = 1$ to 2^k, do:
   (a) Assume this is one step of work, and requires one unit of storage. Choose K''. Store $(E_{K''}(x), K'')$ in the table T' indexed by $E_{K''}(x)$.
2. For $i = 1$ to 2^k, do:
   (a) Assume this is one step of work. Choose K'. If $E^{-1}_{K'}(y)$ is in T' as an index with K'', $(K_1, K_2) = (K', K'')$ is expected with $\Pr[(K_1, K_2) \neq (K', K'')] = 2^{2k-n}$.

Key-Collision Attack on Double Encryption. Let $y = E_{K_1}(E_{K_2}(x))$. Using the key-collision attack, the attacker finds (K_1, K_2) with 2^{k+1} steps and 2^k storage. For simplicity, assume $n > 2k$ so it is unlikely there is a false key hit. Assume x is constant. Assume that keys K_1, K_2, are randomly generated for every request y. This works as follows:

1. For $i = 1$ to 2^k, do:
   (a) Choose keys K', K'', randomly. Let $y' = E_{K_1}(E_{K_2}(x))$. Store $(y', (K', K''))$ in the table T' indexed by y'. Assume this requires one step of work, and one unit of storage.
   (b) Request a y. If y is in T' as an index with (K', K''), $(K_1, K_2) = (K', K'')$ is expected with $\Pr[(K_1, K_2) \neq (K', K'')] = 2^{2k-n}$. Assume this requires one step of work.

Exhaustive Key Search on Single Encryption. Let $y = E_{K_1}(x)$. The attacker finds K_1 with 2^k steps. For simplicity, assume $n > k$ so it is unlikely there is a false key hit. The plaintext-ciphertext pair (x, y) is known. This works as follows:

1. For $i = 1$ to 2^k, do:
   (a) Choose a key K'. If $y = E_{K'}(x)$, $K_1 = K'$ is expected with $\Pr[K_1 \neq K'] = 2^{k-n}$. Assume this requires one step of work.

Key-Collision Attack on Single Encryption. Let $y = E_{K_1}(x)$. Using the key-collision attack, the attacker finds K_1 with 2^k steps. For simplicity, assume $n > k$ so it is unlikely there is a false key hit. Assume x is constant. Assume that K_1 is randomly generated for every request y. This works as follows:

1. For $i = 1$ to $2^{k/2}$, do:
   (a) Choose a key K' randomly. Let $y' = E_{K_1}(x)$. Store (y', K') in the table T' indexed by y'. Assume this requires one step of work, and one unit of storage.
   (b) Request a y. If y is in T' as an index with K', $K_1 = K'$ is expected with $\Pr[K_1 \neq K'] = 2^{k-n}$. Assume this requires one step of work.