High-Speed Pseudorandom Number Generation with Small Memory

William Aiello[1], S. Rajagopalan[2], and Ramarathnam Venkatesan[3]

[1] ATT Labs – Research, Florham Park, NJ, U.S.A. `aiello@research.att.com`†
[2] Telcordia Technologies, 445 South Street, Morristown, NJ 07960, U.S.A. `sraj@research.telcordia.com`
[3] Microsoft Research, One Microsoft Way, Redmond, WA, U.S.A. `venkie@microsoft.com`†

Abstract. We present constructions for a family of pseudorandom generators that are very fast in practice, yet possess provable strong cryptographic and statistical unpredictability properties. While such constructions were previously known, our constructions here have much smaller memory requirements, e.g., small enough for smart cards, etc. Our memory improvements are achieved by using variants of pseudorandom functions. The security requirements of this primitive are a weakening of the security requirements of a pseudorandom function. We instantiate this primitive by a keyed secure hash function. A sample construction based on DES and MD5 was found to run at about 20 megabits per second on a Pentium II.

1 Introduction

We present a simple and practical construction for generating secure pseudo random numbers in software, improving on the prior work [1] by reducing the memory requirement significantly. The algorithm presented in [1] is efficient and possesses provable strong cryptographic and statistical unpredictability properties. The approach was to start with a slow but cryptographically secure pseudorandom generator based on a strong block cipher and then stretch the outputs using the intractability of a certain coding problem and the rapidly mixing property of random walks on expander graphs. In a typical configuration, this generator implemented in C ran at speeds up to 70 Mbits/second on a 200MHz PentiumPro running WindowsNT 4.0. This generator was also incorporated into an IPSec testbed and integrated with a high quality duplex video teleconferencing system on Unix and NT platforms. This typical configuration required 16 to 32 Kbytes of memory which, unfortunately, is quite significant for devices with a limited amount of memory such as smart cards or cell-phones.

Our improvement here is achieved by replacing the stretching mechanism in [1] by using what we call *random-input pseudorandom (family of) functions* (RI-PRF) or *hidden-input pseudorandom (family of) functions* (HI-PRF). These

† Work done while at Telcordia.

functions accept *random* inputs and map them into *longer* pseudorandom outputs. The former allows the adversary to see the random inputs while the latter does not. Hence their security requirements are weaker than usual pseudorandom functions (PRF) that must map *adversarially and adaptively* chosen inputs into pseudorandom outputs. The existence of a RI-PRF is equivalent to the existence of a PRF (and both are equivalent to the existence of a one-way function by [11] and [13]). However, while every PRF is also a RI-PRF, not every RI-PRF is a PRF. For example, consider the following simple modification to a PRF. On input 0, the modified function always outputs zero. Such a modified function can obviously be distinguished from a random function with one query chosen by the adversary. However, the modified function remains a RI-PRF.

We suggest that RI-PRF's can be instantiated in practice by well known secure hash functions such as MD5, and SHA-1. The current attacks against these hash functions ([6], [7], [18], [20]) do not contradict their being secure RI-PRF's.

We note that secure hash functions have been used to construct different primitives. For example, [4] construct PRF's on arbitrary length inputs from PRF's on fixed length inputs which they instantiated by well-known compression functions. [3] construct a message authentication code (an unpredictable PRF family) from well-known compression functions. [17] have independently explored several variations of the PRF model given by varying the requirement between predictability and indistinguishability, and varying the query model from adaptively chosen to random. While our RI-PRF model is the same as one of their variations, our HI-PRF model is new.

Our implementation in C using MD5 and DES ran at 28 to 40 Mbits/sec on a 233 MHz PentiumII running Windows NT 4.0. The memory requirement ranges from 128 Bytes to 3 KBytes. The algorithm can be adapted for encrypting packets that are sent across a network and may be received out of order.

2 The Generator $\mathbf{G_{vra}}$

In [1], a generator named VRA ("Video Rate Algorithm") was presented. We will denote this generator by $\mathbf{G_{vra}}$ and briefly recall its construction for purposes of comparison. The generator presented in this paper is denoted G and described in the next section. In addition, we will present some improvements to $\mathbf{G_{vra}}$ itself in this paper in section 5.4.

$\mathbf{G_{vra}}$ has parameters (n, k, L). It starts with a short preprocessing step when it runs a slow but secure generator, denoted by G_0, the output of which is subsequently stretched to a much longer output during the run of the algorithm in a way many desirable properties are preserved. The secure generator G_0 may be based on a one way permutation (OWP) or block cipher. The computational cost of the preprocessing step is amortized over a long output. The outputs of $z_1, z_2, \ldots, z_m$ are each L bits long. The overall generator needs $(n+1)L$ bits of memory and it takes roughly k simple machine operations like shift, exclusive-or and table look-up to generate each machine word-sized block. The relevant security parameter is $\binom{n}{k}$ where n and k can be chosen so that $\binom{n}{k}$ is large enough to be considered infeasible (e.g. 2^{60}) and $\frac{1}{\binom{n}{k}}$ negligible. Note here that n is not the length of the seed input to the strong generator. Also, n and k can be varied within certain bounds to provide a smooth time-space tradeoff.

2.1 Construction of $\mathbf{G_{vra}}$

The approach in [1] is to begin with a few bits from a high quality source with cryptographic properties. The algorithm "stretches" these bits to a much longer sequence while maintaining many desirable properties.

The generator $\mathbf{G_{vra}}$ has a simple description. First, let us assume we are given a possibly slow but perfect or cryptographically secure generator G_0 (it was shown in [1] how to construct one from a strong cipher such as DES or Triple-DES). Let $x \leftarrow G_0(\cdot)$ denote that x is assigned a random string of length $|x|$ by making a call to G_0 and obtaining the next block of random bits of appropriate length. Below, we use a d-regular expander graph with 2^L nodes each labeled with a L-bit string. Since the graph is d-regular, it can be edge-colored with colors $\{1, \ldots, d\}$. For any node y, $1 \leq r \leq d$, let $neighbor(y, r)$ denote the neighbor of y reached by traversing the edge colored r. $\mathbf{G_{vra}}$ has two processes $\mathbf{P}_0$ and $\mathbf{P}_1$ each producing a sequence of L-bit numbers. Here $\oplus$ stands for bitwise exclusive-or of strings of equal length. $A[i]$ denotes the i-th row of matrix A. p is a probability parameter $\in [0, \frac{1}{2}]$.

```
Initialization for both processes
    P_0 : $A \leftarrow G_0(\cdot), A \in \{0,1\}^{n \times L}$.
    P_1 : $y_0 \leftarrow G_0(\cdot), y_0 \in \{0,1\}^L$.
On-line steps for $i := 1, 2, \ldots.$ do
    P_0 : $c_1, \ldots, c_k \leftarrow G_0(\cdot), c_j \in \{0, 1, \ldots, n-1\}, c_j$ distinct
          $x_i := \oplus_{j=1}^{k} A[c_j]$, $x_i \in \{0,1\}^L$
    P_1 : $b \leftarrow G_0, b \in \{0,1\}^{\lceil \log \frac{1}{p} \rceil}$.
          if $(b = 00\ldots0)$ $y_i := y_{i-1}$ [i.e., stay at the same node with probability $p$]
          else $r \leftarrow G_0, r \in \{1, \ldots, d\}, y_i := neighbor(y_{i-1}, r)$ [i.e., move to a random
          neighbor]
Output $z_i := x_i \oplus y_i$.
```

Note that for process P_1, the random walk on an expander, the computation of each neighbor of a node must be both efficiently computable and require small memory. The Gabber-Galil [9] expander, defined below, is the most efficient in computation and memory of all known expanders. The neighbors of (x, y) are $(x, x + 2y + \{0, 1, 2\})$ and $(2x + y + \{0, 1, 2\}, y)$ where x and y are $\ell = L/2$ bit strings and arithmetic is done $\bmod 2^{\ell}$.

The generator thus stretches $k\lceil \log n \rceil + \lceil \log d \rceil + \lceil \log \frac{1}{p} \rceil$ bits (all logs are binary) to L bits. For example, for the setting $(n = 256, k = 6, d = 6, L = 512, p = \frac{1}{2})$ the stretch factor is approximately 10. The properties of the generator are detailed in [1].

2.2 The Large Memory Requirement of $\mathbf{G_{vra}}$

Note that $\mathbf{G_{vra}}$ requires a matrix A of $n \times L$ random bits. A typical setting for achieving high speed and reasonable security is $n = 256$ and $L = 1024$ [1], i.e., an A of size 32 Kbytes which is significant and may result in two difficulties. First, the memory requirement may be unacceptable to small devices such as cell phones, handheld computing devices, and even low-end set-top boxes. On platforms such as smart cards, a realistic size for available memory is about 1Kbyte. Second, the cache architecture on modern CISC CPUs such as i86xx is such that data structures that do not fit in a cache page suffer from high page swap penalties. $\mathbf{G_{vra}}$ suffers from this problem as the random row accesses to the matrix A cannot be effectively pipelined. The survey in [21] recommends that tables should not be larger than 4Kbytes. Furthermore, for applications such as e-commerce transactions which require both high speed and semantic security (i.e. every bit must be unpredictable to the adversary even if all the other bits are known), the proof in [8] shows that the memory requirement of $\mathbf{G_{vra}}$ is much larger. There seems to be no easy way to configure $\mathbf{G_{vra}}$ with small memory and achieve high security.

In the following section, we show how to reduce the memory requirement to about 1Kbyte while still achieving high security and reasonable speed. We do so by introducing a new primitive called a *random-input pseudorandom function.*

3 Construction and Properties of the Generator G

As in the case of $\mathbf{G_{vra}}$, our approach is to begin with a few pseudorandom bits from a high quality source with strong cryptographic properties and "stretch" these bits to a much longer sequence while maintaining security properties. For this we will use an additional assumption. To begin with, we consider using a pseudorandom function (PRF) [11]. However, as we will see, we do not need the full security of a PRF. We will be able to use a length-increasing function with a weaker security requirement which we call a *random-input pseudorandom* function. Whereas in the case of a PRF, its outputs must appear to be random to a computationally bounded adversary even when the adversary makes adaptive queries to the function, for a length-increasing random-input pseudorandom function (RI-PRF) we only require that the outputs appear to be random to a computationally bounded adversary when the inputs are chosen at random. The expectation is that weaker security requirements for a random-input pseudorandom function may be traded off for greater efficiency.

Next we will describe our construction of a pseudorandom generator G assuming the availability of a RI-PRF. In Section 4.2 we formally define RI-PRF's and suggest implementations.

3.1 The Construction of G

Our generator is a simple modification of $\mathbf{G_{vra}}$. G takes parameters ℓ, m_1, m_2, and n. As before, we assume we have a cryptographically strong PRG G_0. We also assume that we are given a random-input pseudorandom function family with parameters l, m_1 and m_2, $f : \{0,1\}^l \times \{0,1\}^{m_1} \to \{0,1\}^{m_2}, m_2 > m_1$. We write $f_K(x) \equiv f(K,x)$. The parameter n is typically less than 10. Let $L = nm_2$. The P_1 process for G is exactly the same as that for $\mathbf{G_{vra}}$.

A simple description of $\mathbf{P}_0$ is as follows. In the pre-processing step, randomly choose n keys of length l for the n random-input pseudorandom functions. On each on-line step, choose a random input of length m_1 and feed it to each of the RI-PRF's. x_i is the concatenation of all the outputs of the n functions.

We describe the new P_0 process formally below.

```
Initialization P_0 : $K_1, K_2, \ldots, K_n \leftarrow G_0(\cdot), |K_i| = \ell$.
On-line steps for $i := 1, 2, \ldots$. do
  P_0 : $B \leftarrow G_0(\cdot), B \in \{0,1\}^{m_1}$.
        $x_i := f_{K_1}(B) \circ \cdots \circ f_{K_n}(B)$.
```

As in $\mathbf{G_{vra}}$, the i-th output of G is $z_i := x_i \oplus y_i$ where y_i is the ith output of P_1.

Clearly, the memory requirement of P_0 is $\leq n \times \ell + m_1$ bits (not counting the internal memory requirements of f which is used as a blackbox). In a prototypical implementation using MD5, $m_2 = 128$ $m_1 = 64$, $\ell = 512 - 64$. Typically $n = 4$ and hence the memory needed is about 0.5Kbyte. In the on-line steps, the generator thus stretches $m_1 + \lceil \log \frac{1}{p} \rceil + \lceil \log d \rceil$ bits to L bits where $L = n \times m_2$. For example, for the setting $(n = 4, d = 6, l = 448, m_1 = 64, m_2 = 128, p = \frac{1}{2})$ the stretch factor is approximately 8.

Outline: In section 4, we give definitions and conventions used in this paper we show that the process $\mathbf{P}_0$ has strong security properties. In section 5 we describe the properties of the full generator. In section 5.3, we describe some variations in the construction of G and $\mathbf{G_{vra}}$ and their usefulness in specific scenarios.

4 Definitions and Notations

4.1 Preliminaries

For any string $x \in \{0,1\}^\star$, let $|x|$ be its length and let $\circ$ denote the concatenation operator. We first define strong one-way permutations. Let $f : \{0,1\}^\star \to \{0,1\}^\star$ be a length-preserving function that is easy (e.g. polynomial time) to compute. f is a permutation if it is one-to-one as well. Let $T : \mathbb{N} \to \mathbb{N}$ and $\epsilon : \mathbb{N} \to [0,1]$. The security of f is $(T(n), \epsilon(n))$ if the following statement is true for infinitely many n. For a given n, for all probabilistic adversaries A that run for time $\leq T(n)$, $\text{Prob}[A(f(x)) \in f^{-1}(f(x))] \leq \epsilon(n)$ where the probability is taken over the input $x \in \{0,1\}^n$ and the coin-flips of A. f is said to be a "strong" one-way function if it has security $(n^{O(1)}, n^{-\omega(1)})$, where $\omega(1)$ is any function on integers that goes to infinity asymptotically. For ease of exposition, whenever n is understood, we will use T and ϵ in place of $T(n)$ and $\epsilon(n)$, respectively.

A pseudorandom generator g accepts a short random seed x of length, say n, and produces a sequence of bits $g(x) = b_1, b_2, \ldots, b_m$ for some $m > n$. g is defined to have bit security $(T_b(n), \epsilon_b(n))$ if, for any $i, 1 \leq i \leq m$, any probabilistic algorithm that is given $b_1, \ldots, b_{i-1}$, after running for time $\leq T_b$ cannot predict b_{i+1} with probability $> \frac{1}{2} + \epsilon_b$[1]. Analogously, a pseudorandom *sequence* $b_1, \ldots, b_l$ is defined to have security (T_s, ϵ_s) if any probabilistic algorithm, after running for time T_s cannot distinguish $b_1 \ldots b_l$ from l truly random bits with probability $> \frac{1}{2} + \epsilon_s$. Yao [22] showed that the bit security of a generator and the security of any l bits of its output are tightly related: $\frac{T_s}{\epsilon_s} \geq \frac{T_b}{l\epsilon_b}$. g is *cryptographically strong* (or simply strong) if its security is such that whenever $T_b(n)$ is feasible $\epsilon_b(n)$ is negligible. Hence, if g is strong, no efficient algorithm can distinguish g from a random source with better than negligible probability.

Goldreich and Levin [12] gave a construction for a PRG using any one-way permutation such that the bit security of the generator is nearly the same as the security of the underlying one-way function. In [1], a similar construction for a PRG is given for any cipher (which is secure in an appropriately defined model) such that the bit security of the generator is nearly that of the cipher. In our construction of G given in section 3 we assume that G_0 is such a strong generator.

4.2 Pseudorandom Functions

Pseudorandom functions were defined in [11]. They presented a construction that makes as many calls as the input length to a PRG that doubles the input length. Recently, a more economical construction based on the Decisional Diffie-Hellman assumption was given in [16]. However, all constructions whose security is reducible to the security of well-known number theoretic problems are too slow for many applications. In situations which require an efficient PRF, a keyed cryptographic hash function or a block cipher (with Merkle's construction which destroys the invertibility) are often used. Rather than the security of the PRF being reduced to that of a well known problem, it is simply be asserted on the strength of empirical evidence. In such situations it seems prudent to try as much as possible to reduce the security requirements of the cryptographic primitive whose security is being asserted. This is the approach we take here.

Let us first recall the definition of a PRF. Traditionally, PRF's are used as length-preserving functions but we will allow the general version here. A function family with parameters (k, m_1, m_2) family $\mathcal{F}$ is a collection of efficiently computable functions f_K mapping m_1-bits to m_2 bits indexed by $K \in \{0,1\}^k$. Given the string K one can easily compute the function f_K. We will assume that the parameters of the family can be increased suitably and without any bound.

Definition 1 (Security of Prf) *A pseudorandom function family $\mathcal{F}$ has security (T, U, δ) if, for every valid m_1, no probabilistic adversary $\mathcal{A}$ given access to*

[1] Bit security of g has also been defined as $s_b = \frac{T_b}{\epsilon_b}$ (see,e.g., [14]) but we prefer the two-parameter model as it is more transparent.

an oracle of $f_K \in_{random} \mathcal{F}_{m_1}$ running in time T and making at most U queries to the oracle, can distinguish the oracle from a random function oracle with probability (over the adversary's coin flips and choice of K) $> \delta$.

We observe that a slight generalization of the classical definition of a PRF family is to allow the polynomially bounded adversary access to many instances f_{K_i} (the number is determined adaptively by the adversary) where the adversary does not know any of the K_i's but can query any f_{K_i} at the cost of a query each. Next, we define a weaker notion of PRF that we suggest is useful and perhaps more tenable. This notion was independently proposed as *random-challenge pseudorandom* function in [17]. A function family is *random-input pseudorandom* (RI-PRF) if the adversary has to distinguish a member of this family from a truly random one on random queries. That is, the adversary does not adaptively choose the queries as in the definition of a pseudorandom function but rather random query points are chosen for her. More formally, the adversary sees a sequence $(x_1, f_K(x_1)), \ldots, (x_u, f_K(x_u))$ where each x_i is chosen uniformly at random from $\{0,1\}^{m_1}$ and the key $K \in \{0,1\}^l$ is chosen uniformly and is unknown to the adversary.

Definition 2 (Security of RI-PRF family) *A function family $\mathcal{F}$ is random-input pseudorandom with security (T, U, δ) if for every valid m_1, no probabilistic adversary $\mathcal{A}$ given access to a random-input oracle of $f \in_{random} \mathcal{F}_{m_1}$ running in time T and seeing at most U can distinguish U input-output pairs for f_K from U input-output pairs for a truly random function where the input queries are random with probability (over the adversary's coin flips, and choice of K and x_i's) $> \delta$.*

One can trivially get a strong PRG g from a RI-PRF and a pseudorandom source (or a long enough seed). g uses the pseudorandom source to select a random member of the RI-PRF family and an intial random input. Iteratively, for every successive block of pseudorandom bits to be output, g flips as many bits as the input length of the RI-PRF and outputs these bits as well as the value of the RI-PRF on that input. One can stop here and use this construction for a strong generator if one can find concrete functions that are efficient in practice and can be modeled as RI-PRF's with good security. However, the RI-PRF assumption may be too strong for some concrete functions and we would like to weaken the assumption of security further. To this end, we first note that the security of a RI-PRF is at least as much or higher if the input was hidden from the adversary. We call such a function a *hidden-input pseudorandom* (HI-PRF). More formally, we define an HI-PRF family as follows.

Definition 3 (Security of HI-PRF Family) *A function family $\mathcal{F}$ is hidden-input pseudorandom with security (T, U, δ) if for every valid m_1, no probabilistic adversary $\mathcal{A}$ given access to a random-input oracle of $f \in_{random} \mathcal{F}_{m_1}$ running in time T and seeing at most U can distinguish U outputs of f_K from U outputs of a truly random function where the input queries are random and unknown with probability (over the adversary's coin flips, and choice of K and x_i's) $> \delta$.*

A further variation on HI-PRF is when the random inputs are chosen without replacement. Note that drawing with or without replacement for RI-PRF is an inessential distinction since the adversary can see when an input is repeated.

4.3 PRF's vs RI-PRF's vs HI-PRF's vs PRG's

Any RI-PRF or HI-PRF is a one-way function as is clear from the definition. Thus, the notions of PRF, RI-PRF, HI-PRF, and PRG are equivalent in terms of their existence to that of one-way functions. Nevertheless, we show that there exists a function which is an HI-PRF but not a RI-PRF, and another that is a RI-PRF but not a PRF.

First, let us assume we are given a pseudorandom function f from n bits to n bits. Note that the simple function $F_K(x) := (x, f_K(x))$ is an HI-PRF from n bits to $2n$ bits. However, it is not a RI-PRF since the input, seen by the adversary, and output are obviously correlated. A RI-PRF which is not a PRF can be constructed as follows using a standard two-round Feistel network: $F_{K_1,K_2}(x, y) = (z, f_{K_2}(z) \oplus x)$ where $z = f_{K_1}(x) \oplus y$. This function is not a PRF since, for inputs (x, y_1) and (x, y_2), $z_1 \oplus z_2 = y_1 \oplus y_2$, and thus the adversary who can choose these inputs can easily distinguish this function from a random one. However, for random choices for x and y, using ideas of [19] we can show that the outputs are indistinguishable from random if f is a PRF.

Following the comment after Definition 1, we note that the model of security for RI-PRF and HI-PRF can be generalized (as in the case of PRF's) to allow multiple independent instantiations that an adversary can query. Secondly, we note that any function that is a RI-PRF is an HI-PRF. The distinguishing probability for an adversary who does not see the inputs in the HI-PRF model may be significantly lower than when she can see the inputs in the RI-PRF model for the same function. Finally, note that an HI-PRF which is not length increasing is trivial (the identity function is one).

Note that a secure HI-PRF family is equivalent to a PRG whose outputs on correlated seeds are indistinguishable from random when the seeds are of the form $K \circ r_i$ where K is random but fixed and the r_i's are random and independent. However, we can distinguish HI-PRF's from PRG's as follows. As before, let us say that we have a HI-PRF family whose members are indexed with $k = |K|$ bits and that we have n independent instantiations of this family. Then, in i iterations, for an input of $k \cdot n + i \cdot m_1$ random bits, we get $i \cdot n \cdot m_2$ output bits. In order to compare similar things, let us consider a PRG which takes n seeds of m_1 bits each and outputs $i \cdot n \cdot m_2$ bits in i iterations. The difference between the two is that the HI-PRF construction uses an extra $n \cdot k$ bits which may make a difference in their security in the following sense. There may exist functions whose security is lower when used as a PRG (i.e. without the extra $n \cdot k$ random bits input) as compared to HI-PRF's.

To make this concrete, consider the example of MD5 with $n = 1$, $i = 2$, and $m_1 = 64$ (implying $k = 448 = 512 - 64$. The difference between the two modes is as follows: in the case of the PRG, in each iteration we use MD5 with m_1 random bits and a fixed pad known to the adversary of length k . Contrast this with the HI-PRF mode of using MD5 where the k pad bits are chosen at random and are fixed and secret from the adversary. At each iteration, we run MD5 on m_1 random bits and this pad. It is quite conceivable that the bits output by MD5 in the second mode are much more secure than in the first.

[15] define a *pseudorandom synthesizer* as a function whose outputs cannot be distinguished from random even when the adversary knows the outputs on a matrix of inputs generated as a cross-product of two lists of random values. Our construction may be seen as a one-dimensional case of a synthesizer where one of the two inputs is held constant while the other varies over a random list.

4.4 Security of G

Note that from the construction of G and the definition of security of HI-PRF, if G_0 is a secure PRG and if $F_{K_1,\ldots,K_n}(x) := f_{K_1}(x) \circ \cdots \circ f_{K_n}(x)$ is a secure HI-PRF, then G is a secure PRG. In the next lemma we will show that if f is a secure RI-PRF then F is a secure RI-PRF. It follows immediately that F is also a secure HI-PRF.

Lemma 1 *If f_K, with K chosen at random, is a secure* RI-PRF *with security (T, U, ϵ) then $f_{K_1} \circ \ldots \circ f_{K_n}$ with $K_1, \ldots, K_n$ chosen at random is a* RI-PRF *with security (T', U, ϵ') where $T'/\epsilon' \leq nT/\epsilon$.*

Proof Sketch Define D_n to be the input/output sequence given by the composite construction. That is, one input/output pair looks like $r, f_{K_1}(r), \ldots, f_{K_n}(r)$ where r is random. Define D_0 to the the input/output sequence given by n truly random functions, i.e., one input/output pair looks like $r, R_1, \ldots, R_n$ where all the values are random. More generally, let D_i be the input/output sequence defined by the following input/output pairs: $r, f_{K_1}(r), \ldots, f_{K_i}(r), R_{i+1}, \ldots, R_n$, where $R_{i+1}, \ldots, R_n$, and r are random. Using a standard argument initiated in [22], one can show that if there is an adversary distinguishing D_0 from D_n with probability ϵ' then there is some i such that the same adversary can distinguish between D_{i-1} and D_i with probability ϵ'/n. This adversary can then be used to build an adversary distinguishing (with the same probability) sequences whose input/output pairs are given by $r, f_K(r)$, for r random, from sequences whose input/output pairs are given by r, R where both r and R are random. □

4.5 Random-Input Pseudorandom Functions from Hash Functions

As noted earlier, when "provable" PRF's are often too slow in software one resorts to constructions based on cryptographic hash-functions like MD5 and SHA-1 under the assumption that they behave like PRF's. As noted above milder security assumptions are preferable. We use the compression function of secure hash function here. These have considerably longer input buffers than the output. We limit the attackers ability in choosing the inputs, by fixing a substantial portion of the input buffer with a random string so that in effect the compression function on the remainder of the input acts as a length-increasing function.

Definition 4 (RI-PRF's from secure compression functions) *A family $\mathcal{F}_{m_1}$ of* RI-PRF*'s $h : \{0,1\}^{l+m_1} \to \{0,1\}^{m_2}$ where $m_1 < m_2$, is defined as follows. Let $Mix(\cdot)$ be a length preserving function that is easy to compute. and let K be randomly chosen from $\{0,1\}^l$. Then, $f_K : \{0,1\}^{m_1} \to \{0,1\}^{m_2}$ is defined as $f_K(x) := h(Mix(K \circ x))$.*

One may choose *Mix* reflecting the beliefs about the security of the hash function. Examples of $Mix(K, x)$ are $(K \circ x)$, $(x \circ K)$, $(\mathit{lefthalf}(x) \circ K \circ \mathit{righthalf}(x))$ etc. A successful attack on such RI-PRF's must work for most random choices of K by the user (*not* the attacker's random choices). Our current knowledge of attacks on MD5 and SHA-1 suggest that it is reasonable to assume that their compression functions yield suitable RI-PRF's. Even if this assumption proves to be false, the construction of generator still holds as long as there exists a single efficient RI-PRF.

4.6 The Feedback Construction

An alternative method for $\mathbf{P}_0$ to get the m_1-bit inputs is by feeding back some of the output rather than by getting them from G_0. The output length in each iteration then becomes $nm_2 - m_1$. In this section, we analyze the various ways in which output bits can be fed back into the input so that the construction can be iterated. For clarity, let us set $n = 1$, i.e. there are no multiple instantiations of the underlying function. The following discussion also holds for polynomially many independent instantiations of the underlying function. To be more concrete, one can imagine that the function family is instantiated using a hash function like MD5 where the index is simply the random bits used in the pad. However, the arguments hold in general. First, let us consider the simplest variant g_1 of the generator. Let $(y)_{\leftarrow j}$ and $(y)_{\rightarrow j}$ be, respectively, the first and last j bits of a string y. Then, g_1 can described as follows: at the i-th step ($i > 0$), output $y_i := f_K((y_{i-1})_{\leftarrow m_1})$ with y_0 being a random seed of m_1 bits. If f_K is a member of a RI-PRF family, g_1 is a strong generator. Because an HI-PRF derived from a RI-PRF by hiding the input is at least as and perhaps much more secure, we can claim that the following generator g_2 has the same or more securitythan g_1. At the i-th iteration, $y_i := f_K((y_{i-1}))_{\leftarrow m_1}$; if i is the last iteration, output y_i, else output $(y_i)_{\rightarrow(m_2-m_1)}$.

Finally, we can show that the following generator g_3 has higher security than g_2 under the assumption that we have access to a pseudorandom source g_0 whose security is higher than the security of f_K. g_3 can be described simply as: at the i-th step, g_0 outputs a random x_i of length m_1 and g_3 outputs $f_K(x_i)$. Let ϵ be the probability of distinguishing these m_1 feedback bits from random. It can be shown by induction that after t iterations, the probability of distinguishing these m_1 feedback bits is now only bounded by $t\epsilon$. Thus, the security of t outputs of the alternative $\mathbf{P}_0$ is degraded by this amount. Let us compare this to using G_0 instead of truly random bits for t outputs of the standard $\mathbf{P}_0$, suppose that the security of m_1 bits of G_0 is ϵ'. It can be shown that the security of t outputs of $\mathbf{P}_0$ is degraded by $t\epsilon'$. Thus, whenever G_0 is more secure than the feedback bits of $\mathbf{P}_0$, using G_0 will result in a more secure (albeit, potentially slower) generator. The G_0 we will use in our implementations is based on a strong cipher such as DES and possibly much more secure than any m_1 bits of the $\mathbf{P}_0$.

5 Properties of G

5.1 Dealing with Birthday Collisions

As shown in the previous section the process $\mathbf{P}_0$ in-and-of-itself produces a cryptographically strong pseudorandom sequence. However, in practice, when actually setting the parameters so that $\mathbf{P}_0$ is very fast, the value of m_1 is relatively small so as to reduce the load on the strong pseudorandom generator. However, in such a case, m_1 may not be large enough to avoid birthday collisions. By the birthday paradox, in $\approx \sqrt{2^{m_1}}$ steps, some x_i is quite likely to repeat, which would help to distinguish outputs of $\mathbf{P}_0$ from a truly random sequence of comparable length. A similar problem occurred in [1]. The solution we propose here is the same as for $\mathbf{G_{vra}}$. In parallel with $\mathbf{P}_0$ we preform a random walk $\mathbf{P}_1$ and then take the bitwise xor of the two outputs as the output of the generator $\mathbf{G}$. This is precisely the generator described in Section 2.

The intuition behind this construction is as follows. For a random walk on an expander of 2^L nodes, the probability that the walk is at the same node that it was t steps ago is very small, say $2^{-c.t}$ for some $c > 0$. Hence for large enough t this probability is small and offers a good "long range" decorrelation properties, while in the short range, an output of $\mathbf{P}_1$ can easily repeat. However, the probability that $\mathbf{P}_0$ returns in t steps is negligible for small t ("in the short range") and as mentioned above in the long range the choice m_1 may make the output repeat. The idea is that by xoring the two processes, the probability of return will be relatively small for all values of t.

Before we quantify these remarks, we observe that adding $\mathbf{P}_1$ to $\mathbf{P}_0$ did not weaken the cryptographic security of $\mathbf{P}_0$. Indeed, if we are given a distinguisher D for $\mathbf{G}$ we can attack $\mathbf{P}_0$ as follows: Generate $\mathbf{P}_1$ of suitable lengths independently and xor it with the output of $\mathbf{P}_0$. Now D will distinguish this from random strings; however this would be impossible if $\mathbf{P}_0$ were purely random.

5.2 Return Time for G

A generator is said to enter a cycle of period M after a transient period of M_0 steps, if $z_i = z_{i+M}$ for all $i > M_0$, for some M_0. It is desirable for a PRG to have large cycle length M, since it gives an upper bound on the maximum number of useful outputs that can be obtained from G with a single seed. A generator with security $(T(n), s(n))$ cannot, *on average*, have cycle length substantially smaller than $T(n)(1 - \epsilon(n))$ since one can distinguish the generator's output from random sequences with certainty in time proportional to the cycle length. However, this does not preclude the generator from having small cycle lengths on some small fraction of inputs (seeds). In this section, we show that the probability that G repeats an input is very small. Note that repeating an earlier output is necessary but not sufficient for a generator to be in a cycle. Using the theory of random walks on expanders, [1] showed that the probability of $\mathbf{G_{vra}}$ repeating its starting point is negligible. In fact, it is smaller than either the probability that P_0 repeats or the probability that P_1 repeats. A similar result holds for the probability that G repeats its starting point (or any other point for that matter) and is given in the lemmas below. To state the lemma we let $V(m)$ and $W(m)$ be the probability that P_0 and P_1 repeat after m steps.

Lemma 2 *The probability that G repeats its first output at the m-th step, for* $m \geq 2$*,*

$$V(m)W(m) \leq R(m) := Prob[Z_m = Z_1] \leq min(V(m), 2^{-L} + \bar{\lambda}_2^m(1 - 2^{-L}))$$

where the probability is over the choice of inputs to $\mathbf{P}_0$ *and the random walk at each step. Moreover,* $\bar{\lambda}_2$ *is the second largest eigenvalue of the expander graph in* P_1 *in absolute value.*

The following lemma gives a lower bound on $W(m)$ in terms the degree of the expander.

Lemma 3 (Alon-Boppana) *The probability* $W(m)$ *of return of* $\mathbf{P}_1$ *in* m *steps is bounded below as:*

$$\begin{aligned} Prob[Y_m = Y_1] &\geq md^{-m}p^{m-1}(1-p)\rho(m-1) \quad \textit{for odd } m > 3 \\ &\geq d^{-m}p^m\rho(m) \quad \textit{for even } m > 2 \end{aligned}$$

where p *is the probability of staying at the same node,* $\rho(2r) = \frac{1}{r}\binom{2r-2}{r-1}d(d-1)^{r-1}$*,* $r \geq 2$*. (It is easy to compute* $W(m)$ *exactly for* $m = 1, 2, 3$*.)*

Random walks on expander graphs have other strong statistical properties. For example, they generate a sequence of numbers that are statistically almost as good as random and independent numbers for use in Monte-Carlo estimations as was shown by Gillman [10]. That is, the mean of any bounded function over the node values can be estimated with an error which decreases exponentially with the number of samples. This was generalized in [1] to the case when the random walk node label was xored with the output of the $\mathbf{P}_0$ given in that paper and it extends to our case with G.

5.3 Alternate Ways to Avoid Birthday Repeats

Note that the security is not independent of m_2. Indeed, after $2^{m_1/2}$ input-output pairs, some output is likely to repeat whereas for a true random source this does not happen until $2^{m_2/2}$ outputs. This is taken care of by the $\mathbf{P}_1$ process. However, the $\mathbf{P}_1$ process can be difficult to handle for small devices since it involves arithmetic modulo large numbers. We suggest some new ways of avoiding the birthday attack without using P_1.

The essential idea is that the birthday collisions $\mathbf{P}_0$ works in time $2^{m_1/2}$. This is because we have fixed l bits of the input. However the RI-PRF may have significantly more security than $2^{m_1/2}$. We can exploit this additional security by modifying the construction as follows. Some number of bits, say m_3, of the m_2 output bits are fed back and only $m_2 - m_3$ bits are output. These bits are temporarily stored until enough bits are accumulated to form a new $n \times l$ matrix A of key values. It takes $\lceil nl/m_3 \rceil$ rounds to compute this new A. After this new A has been computed, the key values for the random-input pseudorandom functions are given by this A. Since $\lceil nl/m_3 \rceil$ is much less than $2^{m_1/2}$, the birthday attack is thwarted. In fact, it can be shown that the security of the scheme can be reduced to syntactically weaker security requirements for the RI-PRF since the adversary is only given $\lceil nl/m_3 \rceil$ queries per random key with which to distinguish the outputs of the function from random. We should also note that the bits for generating the new A need not be feedback bits alone. It may be desirable for some or all of these bits to come from G_0. The decision about where to get the new bits of A depends on the details of the security limitations of G_0 and the random-input pseudorandom function.

One need not wait until all of the bits of the new A are ready before incorporating some of these bits into the new keys. In fact, the feedback bits could be used as new bits for the key values of the next round. That is, at each on-line step, we output the value of a new function $f_K : \{0,1\}^{m_1} \to \{0,1\}^{m_2}$ where K changes slightly at each step. For the MD5 construction, m_3 could be 32, and the buffer is "rotated" right with the extra 32 bits appended to the buffer. Heuristically, this appears to be a better scheme than that above since the keys change at every step, but we are not able to prove this formally.

5.4 Variations on $\mathbf{G_{vra}}$

A similar idea to the above for reducing birthday collisions can be used for $\mathbf{G_{vra}}$. The $n \times L$ matrix A can be interpreted as a linear array. The feedback bits can be appended to the linear array to form a new linear array from 1 to $n \times L + m_3$. A new matrix can be fashioned from the linear array from locations m_3 to $n \times L + m_3$. Alternatively, each row of A can be "rotated" right using some of the feedback bits.

We also suggest some changes to $\mathbf{G_{vra}}$ that make it more suitable to a larger set of applications. Recall that the matrix A is completely filled in the pre-processing step with random bits. As noted earlier, this may be a large cost for some applications. The case of inadequate memory has been addressed with G. Here we provide a solution for applications that occasionally need only a short string of pseudorandom bits or need to amortize the overhead of the preprocessing step. For example, if the number of bits needed is much less than $n/k \times L$, then some rows of A will not be used. For this scenario, we offer the following alternative. We do not fill A in the pre-processing step. Instead, each row has a "filled" bit which indicates whether that row has been initialized or not. In each on-line step, we first generate the random choice of rows. For every row thus chosen, we first check the "filled" bit for that row and if it is not set, we go ahead and generate the random bits of the row. Thus, only those rows that will actually be used will be filled in A.

6 Implementation

In this section, we present the results of our sample implementation on a Pentium II 233 Mhz workstation Windows NT 4.0 with Visual C++ 5.0. The algorithm was implemented entirely in C with no heroic attempts being made for optimization. In order to implement the generator, we have to choose instances of a strong pseudorandom generator and a RI-PRF. For the strong generator, we chose two examples: first, we used the outputs of single DES in OFB mode. We also implemented the Goldreich-Levin generator based on Triple DES in OFB mode. For comparison, we also implemented the strong generator using "alleged" RC4. For the RI-PRF implementation, we chose MD5 and SHA-1. These choices constrain the parameters of our implementation of G as follows. In the case of MD5, the input buffer is of length 512 bits and the hashed output is 128 bits long. For n parallel instances of MD5, the memory requirement is around $512n$ bits. For small devices one can take $n \leq 8$ with a memory requirement ranging from 64 to 512 bytes. We chose $m_1 = 64$, which is the block length of DES.

We used the public domain implementations of DES and MD5/SHA-1. The raw speeds of MD5 and SHA-1 were about 45Mbits/sec and 32 Mbits/sec, respectively. The raw speed of DES in OFB mode was found to be 25 Mbits/sec. The raw speed of the Goldreich-Levin generator based on Triple DES was 10 Mbits/sec. For comparison, RC4 ran at about 64 Mbits/second on this platform. The following paragraph summarizes the speeds for various combinations of our generator parameters using DES or RC4 as the initial generator G_0.

Here we summarize the speeds in Mbits/sec of our implementations for various values of n. In all cases, the strong generator was DES in OFB mode, and the P_1 process was used. We observed that the P_1 process slows down the generator by about 10% for the parameter values tested. The Goldreich-Levin slowed down the generator by approximately 20%. We also noted that for higher values of n the speeds continued to increase but the memory requirement for $\mathbf{P}_0$ was more than 1Kbyte. The speeds for $n = 1, 2, 4, 8$ for MD5 were, respectively, $23, 30, 35, 38$ Mbits/s. The corresponding figures for SHA-1 were $21, 25, 27, 28$ Mbits/s respectively. For comparison we report the speed of G when using the RC4 as G_0. When $n = 8$, G ran at 41 Mbits/sec and 32 Mbits/sec using MD5 and SHA-1, respectively. Although RC4 is much faster than DES, it does not improve the speed of G significantly. The conclusion is that the factor limiting the speed of G is the hash function: MD5 and SHA-1.

References

1. W. Aiello, S. Rajagopalan, R. Venkatesan. "Design of Practical and Provably Good Random Number Generators," *J. Algorithms* **29**, (1998) 358-389. Appeared previously in *5th Annual ACM-SIAM Symp. Disc. Alg.* (1995) 1-8.
2. N. Alon, "Eigenvalues and expanders," *Combinatorica* **6** (1986) 83-96.
3. M. Bellare, R. Canetti, H. Krawczyk, "Keying Hash Functions for Message Authentication," *Proceedings of Crypto '96*, 1-15.
4. M. Bellare, R. Canetti, H. Krawczyk, "Pseudorandom functions revisited: The Cascade Construction and its Concrete Security," *Proceedings of the 37th Symposium on Foundations of Computer Science*, 1996.
5. M. Blum, S. Micali, "How to Generate Cryptographically Strong Sequences of Pseudorandom Bits," *Proceedings of FOCS* (1982); *SIAM J. Computing*, **13** (1984) 850-864.
6. B. den Boer, A. Bosselaers, "Collisions for the compression function of MD5," *Proceedings of Eurocrypt '93*, 293-304, 1994.
7. H. Dobbertin, "The Status of MD5 After a Recent Attack," *CryptoBytes* v2 No.2 (Summer 1996).
8. J.-B. Fischer, J. Stern, "An efficient pseudorandom generator provably as secure as syndrome decoding," in *Proceedings of EUROCRYPT96* (1996) 245-255.
9. O. Gabber, Z. Galil, "Explicit constructions of linear-sized superconcentrators," *J. Comput. Sys. Sci.* **22** (1981).
10. D. Gillman, "A Chernoff bound for random walks in expander graphs," *Proceedings of FOCS* (1993).
11. O. Goldreich, S. Goldwasser, and S. Micali, "On the cryptographic applications of random functions," in *Proceedings of CRYPTO 84* (1984) 276–288.
12. O. Goldreich, L.A. Levin, "Hard Core Bit For Any One Way Function," *Proceedings of STOC* (1990); *J. Symbolic Logic* **58** (1993) 1102-1103.
13. J. Hastad, R. Impagliazzo, L.A. Levin, M. Luby, "Pseudorandom Generation From One-Way Functions, *Proceedings of STOC* (1989) 12-24; "Pseudorandom Generators under Uniform assumptions," *Proceedings of STOC* (1990) 395-404.
14. M. Luby, "Pseudorandomness and its Cryptographic applications" *Princeton Univ Press*, 1996.
15. M. Naor, O. Reingold, "Synthesizers and their application to the parallel construction of Pseudorandom Functions," *Proceedings of FOCS*, (1995) 170-181.
16. M. Naor, O. Reingold, "Number-Theoretic Constructions of Efficient Pseudorandom Functions," *Proceedings of FOCS*, (1997) 458-467.
17. M. Naor, O. Reingold, "From Unpredictability to Indistinguishability: A simple construction of Pseudorandom Functions from MACs," *Proceedings of Crypto'98*, 267-282.
18. P. van Oorschot, M. Wiener, "Parallel collision search with application to hash functions and discrete logarithms," in *Proceedings of 2nd ACM Conference on Computer and Communication Security*, 1994.
19. J. Patarin, "New Results on Pseudorandom Permutation Generators Based on the DES Scheme," *Proceedings of Crypto'91*, (1991) 301–312.
20. M. J. B. Robshaw, "On Recent Results for MD2, MD4 and MD5," *RSA Laboratories Bulletin* No. 4 (November 12, 1996).
21. B. Schneier, D. Whiting, "Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor," *Fast Software Encryption Workshop* (1997) 242-259.
22. A.C. Yao, "Theory and Applications of Trapdoor Functions," *Proceedings of FOCS* (1982) 80-91.