Cryptanalytic Attacks on Pseudorandom Number Generators

John Kelsey[1], Bruce Schneier[1], David Wagner[2], and Chris Hall[1]

[1] Counterpane Systems
e-mail: {kelsey,schneier,hall}@counterpane.com
[2] University of California Berkeley
e-mail: daw@cs.berkeley.edu

Abstract. In this paper we discuss PRNGs: the mechanisms used by real-world secure systems to generate cryptographic keys, initialization vectors, "random" nonces, and other values assumed to be random. We argue that PRNGs are their own unique type of cryptographic primitive, and should be analyzed as such. We propose a model for PRNGs, discuss possible attacks against this model, and demonstrate the applicability of the model (and our attacks) to four real-world PRNGs. We close with a discussion of lessons learned about PRNG design and use, and a few open questions.

1 Introduction and Motivation

It is hard to imagine a well-designed cryptographic application that doesn't use random numbers. Session keys, initialization vectors, salts to be hashed with passwords, unique parameters in digital signature operations, and nonces in protocols are all assumed to be random[1] by system designers. Unfortunately, many cryptographic applications don't have a reliable source of real random bits, such as thermal noise in electrical circuits or precise timing of Geiger counter clicks [FMK85,Gud85,Agn88,Ric92]. Instead, they use a cryptographic mechanism, called a Pseudo-Random Number Generator (PRNG) to generate these values. The PRNG collects randomness from various low-entropy input streams, and tries to generate outputs that are in practice indistinguishable from truly random streams [SV86,LMS93,DIF94,ECS94,Plu94,Gut98].

In this paper, we consider PRNGs from an attacker's perspective. We discuss the requirements for PRNGs, give a basic model of how such PRNGs must work, and try to list the possible attacks against PRNGs. Specifically, we consider ways that an attacker may cause a given PRNG to fail to appear random, or ways he can use knowledge of some PRNG outputs (such as initialization vectors) to guess other PRNG outputs (such as session keys).

[1] Note that "random" is a word that is easily misused. In this paper, unless we say otherwise, the reader may assume that a "random value" is one sample of a random variable which is uniformly distributed over the entire set of n-bit vectors, for some n.

1.1 Applications of Results

This research has important practical and theoretical implications:

1. A PRNG is its own kind of cryptographic primitive, which has not so far been examined in the literature. In particular, there doesn't seem to be any widespread understanding of the possible attacks on PRNGs, or of the limitations on the uses of different PRNG designs. A better understanding of these primitives will make it easier to design and use PRNGs securely.
2. A PRNG is a single point of failure for many real-world cryptosystems. An attack on the PRNG can make irrelevant the careful selection of good algorithms and protocols.
3. Many systems use badly-designed PRNGs, or use them in ways that make various attacks easier than they need be. We are aware of very little in the literature to help system designers choose and use these PRNGs wisely.
4. We present results on real-world PRNGs, which may have implications for the security of fielded cryptographic systems.

1.2 The Rest of This Paper

In Section 2, we define our model of a PRNG, and discuss the set of possible attacks on PRNGs that fit this model. In Section 3 discuss applications of those attacks on several real-world PRNGs. Then, in Section 4, we end with a discussion of the lessons learned, and a consideration of some related open problems.

2 Definitions

In the context of this paper, a PRNG is a cryptographic algorithm used to generate numbers that must appear random. Examples of this include the ANSI X9.17 key generation mechanism [ANSI85] and the RSAREF 2.0 PRNG [RSA94]. A PRNG has a secret state, S. Upon request, it must generate outputs that are indistinguishable from random numbers to an attacker who doesn't know and cannot guess S. In this, it is very similar to a stream cipher. Additionally, however, a PRNG must be able to alter its secret state by processing input values that may be unpredictable to an attacker. A PRNG often starts in an state that is guessable to an attacker (usually unintentionally), and must process many inputs to reach a secure state. Sometimes, the input samples are processed each time an output is generated: e.g., ANSI X9.17. Other times, the input samples are processed as they become available: e.g. RSAREF 2.0 PRNG.

Note that the inputs are intended to carry some unknown (to an attacker) information into the PRNG. These are the values typically collected from physical processes (like hard drive latencies [DIF94]), user interactions with the machine [Zim95], or other external, hard-to-predict processes. Typically, system implementers and designers will try to ensure that there is sufficient entropy in these inputs to make them unguessable by any practical attacker.

Note that the outputs are intended to stand in for random numbers in essentially any cryptographic situation. Symmetric keys, initialization vectors, random parameters in DSA signatures, and random nonces are common applications for these outputs.

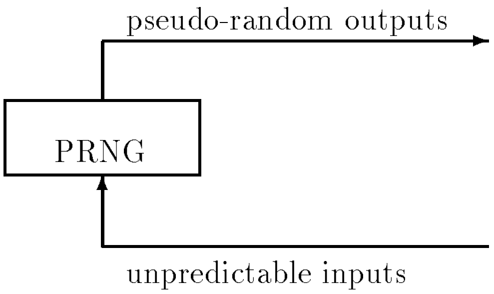
See Figure 1 for a high-level view of a PRNG. Also, Figure 2 refines the terminology a bit, and Figure 3 shows a PRNG with periodic reseeding.

PRNGs are typically constructed from other cryptographic primitives, such as block ciphers, hash functions, and stream ciphers. There is a natural tendency to assume that the security of these underlying primitives will translate to security for the PRNG.

In this paper, we consider several new attacks on PRNGs. Many of these attacks may be considered somewhat academic. However, we believe there are situations that arise in practice in which these attacks are possible. Additionally, we believe that even attacks that are not *usually* practical should be brought to the attention of those who use these PRNGs, to prevent the PRNGs' use in an application that *does* allow the attacks.

Note that in principle, any method of distinguishing between PRNG outputs and random outputs is an attack; in practice, we care much more about the ability to learn the values of PRNG outputs not seen by the attacker, and to predict or control future outputs.

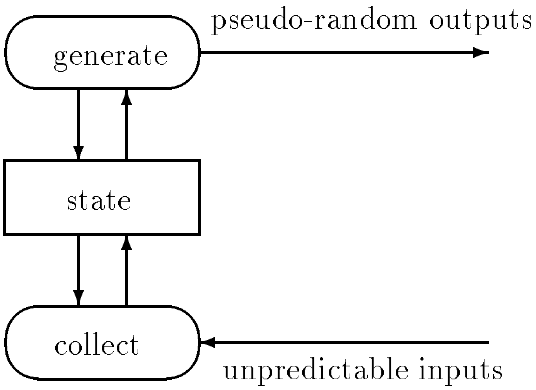
Fig. 1. Black-box view of a PRNG



2.1 Enumerating the Classes of Attacks

1. **Direct Cryptanalytic Attack.** When an attacker is directly able to distinguish between PRNG outputs and random outputs, this is a direct cryptanalytic attack. This kind of attack is applicable to most, but not all, uses of PRNGs. For example, a PRNG used only to generate triple-DES keys may never be vulnerable to this kind of attack, since the PRNG outputs are never directly seen.

Fig. 2. View of internal operations for most PRNGs



2. **Input-Based Attacks.** An input attack occurs when an attacker is able to use knowledge or control of the PRNG inputs to cryptanalyze the PRNG, i.e., to distinguish between PRNG output and random values.
   Input attacks may be further divided into *known-input*, *replayed-input*, and *chosen-input* attacks. Chosen input attacks may be practical against smartcards and other tamper-resistant tokens under a physical/cryptanalytic attack; they may also be practical for applications that feed incoming messages, user-selected passwords, network statistics, etc., into their PRNG as entropy samples. Replayed-input attacks are likely to be practical in the same situations, but require slightly less control or sophistication on the part of the attacker. Known-input attacks may be practical in any situation in which some of the PRNG inputs, intended by the system designer to be hard to predict, turn out to be easily predicted in some special cases. (An obvious example of this is an application which uses hard-drive latency for some of its PRNG inputs, but is being run using a network drive whose timings are observable to the attacker.)
3. **State Compromise Extension Attacks.** A state compromise extension attack attempts to extend the advantages of a previously-successful effort that has recovered S as far as possible. Suppose that, for whatever reason—a temporary penetration of computer security, an inadvertent leak, a cryptanalytic success, etc.—the adversary manages to learn the internal state, S, at some point in time. A state compromise extension attack succeeds when the attacker is able to recover unknown PRNG outputs (or distinguish those PRNG outputs from random values) from before S was compromised, or recover outputs from after the PRNG has collected a sequence of inputs which the attacker cannot guess.
   State compromise extension attacks are most likely to work when a PRNG is started in an insecure (guessable) state due to insufficient starting entropy. They can also work when S has been compromised by any of the attacks

in this list, or by any other method. In practice, it is prudent to assume that occasional compromises of the state S may happen; to preserve the robustness of the system, PRNGs should resist state compromise extension attacks as thoroughly as possible.

(a) **Backtracking Attacks.** A backtracking attack uses the compromise of the PRNG state S at time t to learn previous PRNG outputs.
(b) **Permanent Compromise Attacks.** A permanent compromise attack occurs if, once an attacker compromises S at time t, all future and past S values are vulnerable to attack.
(c) **Iterative Guessing Attacks.** An iterative guessing attack uses knowledge of S at time t, and the intervening PRNG outputs, to learn S at time $t + \epsilon$, when the inputs collected during this span of time are guessable (but not known) by the attacker.
(d) **Meet-in-the-Middle Attacks.** A meet in the middle attack is essentially a combination of an iterative guessing attack with a backtracking attack. Knowledge of S at times t and $t+2\epsilon$ allow the attacker to recover S at time $t + \epsilon$.

3 Attacking Real-World PRNGs

In this section we discuss the strengths and weaknesses of four real-world PRNGs: the ANSI X9.17 PRNG, the DSA PRNG, the RSAREF PRNG, and CryptoLib.

3.1 The ANSI X9.17 PRNG

The ANSI X9.17 PRNG [ANSI85,Sch96] is intended as a mechanism to generate DES keys and IVs, using triple-DES as a primitive. (Of course, it is possible to replace triple-DES with another block cipher.) It has been used as a general-purpose PRNG in many applications.

1. K is a secret triple-DES key generated somehow at initialization time. It must be random and used only for this generator. It is part of the PRNG's secret state which is never changed by any PRNG input.
2. Each time we wish to generate an output, we do the following:
   (a) $T_i = E_K(\text{current timestamp})$.
   (b) $\text{output}[i] = E_K(T_i \oplus \text{seed}[i])$.
   (c) $\text{seed}[i+1] = E_K(T_i \oplus \text{output}[i])$.

This generator is in widespread use in banking and other applications.

Direct Cryptanalytic Attack Direct cryptanalysis of this generator appears to require cryptanalysis of triple-DES (or whatever other block cipher is in use.) As far as we know, this has never been proven, however.

Input-Based Attacks The X9.17 PRNG has a certificational weakness (assuming a 64-bit block size) with respect to replayed-input attacks.

An attacker who can force the T values to freeze can distinguish the PRNG's outputs from random outputs after seeing about 2^{32} 64-bit outputs. In a sequence of random 64-bit numbers, we would expect to see a collision after about 2^{32} outputs. However, with T frozen, we expect a collision from X9.17 to require about 2^{63} outputs. This is a mostly academic weakness, but it may be relevant in some applications.

Otherwise, knowledge or control of inputs does not appear to weaken the PRNG against an attacker that doesn't know K.

State Compromise Extension Attacks The X9.17 PRNG does not properly recover from state compromise. That is, an attacker who compromises the X9.17 triple-DES key, K, can compromise the whole internal state of the PRNG from then on without much additional effort.

Two Design Flaws in X9.17 There are two flaws in the ANSI X9.17 PRNG that become apparent only when the PRNG is analyzed with respect to state compromise extension attacks.

1. Only 64 bits of the PRNG's state, $seed[i]$, can ever be affected by the PRNG inputs. This means that once an attacker has compromised K, the PRNG can never fully recover, even after processing a sequence of inputs the attacker could never guess.
2. The $seed[i+1]$ value is a function of the previous output, the previous T_i, and K. To an attacker who knows K from a previous state compromise, and knows the basic properties of the timestamp used to derive T_i, $seed[i+1]$ is simply not very hard to guess.

The Permanent Compromise Attack: Deriving the Internal State from Two Outputs Consider an attacker who learns K. Much later, after the seed internal variable has become totally different, he is given two successive outputs, output$[i, i+1]$. (He has not seen any intervening outputs from the PRNG.) The attacker's goal will be to learn the value of seed$[i+1]$. Of course, one can trivially mount a 64-bit search and learn the seed value.

However, there is a much more effective way to mount this attack. Suppose that each timestamp value has ten bits that aren't already known to the attacker. (This is a reasonable assumption for many systems. For example, consider a millisecond timer, and an attacker who knows to about the nearest second when an output was generated.) An attacker with two successive outputs can mount a meet-in-the-middle attack to discover the internal seed value, requiring about 2^{11} trial encryptions under the known key K. This works because we have

$$\mathrm{seed}[i+1] = D_K(\mathrm{output}[i+1]) \oplus T_{i+1}$$

$$\mathrm{seed}[i+1] = E_K(\mathrm{output}[i] \oplus T_i)$$

The attacker tries all possible values for T_i, and forms one sorted list of possible seed$[i+1]$ values. He then tries all possible values for T_{i+1}, and forms another sorted list of possible seed[$i+1$] values. The correct seed[$i+1$] value is the one that appears in both lists.

The Iterative Guessing Attack If an attacker knows seed[i], and sees some function of output[$i+1$], he can learn seed[$i+1$] in almost all cases. This is true because the timestamp sample will seldom have much entropy. Using our earlier assumption of ten bits of entropy per timestamp sample, this means the attacker will need only a ten-bit guess. Note that the attacker needs only to see a *function* of the output, not the output itself. This means that a message encrypted with a key derived from the output value is sufficient to mount this attack. (Note the difference between this and the permanent compromise attack, above, in which the attacker needs raw PRNG outputs.)

Backtracking The attacker can move backwards as easily as forward with the iterative guessing attack, assuming he can find functions of the PRNG outputs. Alternatively, he may look for the successive pair of directly available PRNG outputs nearest to the unknown outputs he wants to learn, and mount the permanent compromise attack there.

Meet-in-the-Middle Attack Sometimes, a PRNG may generate a large secret value, and not directly output any bits of it. The attacker may thus know seed[i] and seed[$i+8$], but no intervening values. Since this leaves him with (say) 80 bits of entropy, it might be naively assumed that he cannot recover these output values. However, this isn't necessarily the case, because a meet-in-the-middle attack is available. This works as follows:

1. The attacker mounts the attack described above to learn the PRNG state before and after the run of values that were used together.
2. The attacker carries out a meet-in-the-middle attack, deriving one set of possible values for seed[$i+4$] by guessing $T_{i+1..i+4}$ and deriving a second list by guessing $T_{i+5..i+8}$. If each sequence of four timestamps holds 40 bits of entropy, this will require 2^{41} effort. The correct value of seed[$i+4$] will be present in both lists, so the seed[$i+4$] values that match (there will be about 2^{16} of these) yield the possible sequences of timestamps, and thus, output blocks.
3. The attacker can try all these possible output sequences until he finds the right one. (For example, if the eight output blocks are used as an encryption key, 2^{16} trial decryptions will suffice to eliminate all the false alarms.)

Timer Entropy Issues In the above discussion, we have assumed that individual PRNG inputs have fixed amounts of entropy, and thus, take fixed amounts of effort to guess. In practice, this usually won't be the case. An RSA keypair generation might reasonably use two 512-bit pseudorandom starting points, thus requiring a total of sixteen PRNG output requests. However, these calls will almost certainly be made in rapid succession. Unless the timestamp on which the T_i values are based has a great deal of precision, many of these T_i values will be based on the same or very close timestamp values. This may well make meet-in-the-middle attacks practical, even though it might normally make sense to estimate at least three bits of unpredictability per timestamp.

Summary The ANSI X9.17 key generator appears to be fairly secure from all attacks that *don't* involve either stopping the timer used or compromising the internal triple-DES key. Replaying any timer input about 2^{32} times leads to a certificational weakness: a way to distinguish large numbers of X9.17 PRNG outputs from a truly random sequence of bits. Compromising the internal triple-DES key completely destroys the X9.17 PRNG: it never recovers, even after getting thousands of bits worth of entropy in its sampled timer inputs[2].

For systems that use X9.17, the most obvious way to resist this class of attack is to occasionally use the current X9.17 state to generate a whole new X9.17 state, including a new K and a new starting $seed[0]$.

3.2 The DSA PRNG

The Digital Signature Standard specification [NIST94] also describes a fairly simple PRNG based on SHA (or, alternatively, a DES construction) which was intended for generating pseudorandom parameters for the DSA signature algorithm. Since this generator appears to come with an NSA stamp of approval, it has been used and proposed for applications quite different than those for which it was originally designed.

The DSA PRNG allows an optional user input while generating keys, but not while generating DSA signature parameters. For our purposes, though, we will assume that the PRNG can be given user inputs at any time, as is true with the other PRNGs discussed in this paper. Each time the DSA PRNG generates an output, it may be provided with an optional input, W_i. Note that omitting the input from the PRNG design would guarantee that the PRNG could never recover from a state compromise.

All arithmetic in this PRNG is allowed to be done modulo 2^N, where $160 \leq N \leq 512$. In the remainder of this document, we will assume this modulus to be

[2] Wei Dai's Crypto++ library [Dai97] includes an implementation of a X9.17 variant with increased security against seed compromise attacks. That variant is

1. $T_i = E_K(T_{i-1} \oplus \text{current timestamp})$.
2. $\text{output}[i] = E_K(T_i \oplus \text{seed}[i])$.
3. $\text{seed}[i+1] = E_K(T_i \oplus \text{output}[i])$.

This corresponds to encrypting the timestamps in CBC mode, instead of in ECB mode as is done in the standard X9.17 generator. The timestamp is based on the program's CPU usage, and its resolution is platform-dependent; on Linux, it has a 0.01 second resolution. We have not examined this PRNG closely, but we note that our permanent compromise attack, above, can be extended to work on Crypto++'s X9.17 variant at a cost of requiring a 2^{64} search in the attack

160, since this is the weakest value (with respect to one attack) that is allowed by the design.

The DSA PRNG works as follows:

1. The PRNG maintains an ever-changing state, X_i.
2. The PRNG accepts an optional input, W_i. This may be assumed to be zero if not supplied.
3. The PRNG generates each output as follows:
   (a) $\mathrm{output}[i] = \mathrm{hash}(W_i + X_i \bmod 2^{160})$
   (b) $X_{i+1} = X_i + \mathrm{output}[i] + 1 \pmod{2^{160}}$

Direct Cryptanalytic Attack If the PRNG's hash function is good, then the resulting output sequence appears to be hard to distinguish from a random sequence. It would be nice, from a system designer's point of view, to have some proof of the quality of this PRNG's outputs based on the collision-resistance or one-wayness of the hash function; to our knowledge, no such proof exists.

Input-Based Attacks Consider an attacker who can control the inputs sent into W. If these inputs are sent directly in, there is a straightforward way to force the PRNG to repeat the same output forever. This has a direct relevance if this PRNG is being used in a system in which the attacker may control some of the entropy samples sent into the PRNG. To force the PRNG to repeat, the attacker forms

$$W_i = W_{i-1} - \mathrm{output}[i-1] - 1 \pmod{2^{160}}$$

This forces the seed value to repeat, which forces the output values to repeat. Note, however, that this attack fails quickly when the user hashes his entropy samples before sending them into the PRNG. In practice, this is the natural way to process the inputs, and so we suspect that few systems are vulnerable to this attack.

State Compromise Extension Attacks The DSA PRNG doesn't handle state compromises as well as we might have liked, but it is much better in this regard than ANSI X9.17. Consider an attacker who has somehow compromised the entire internal state of the PRNG, but then lost track of its inputs and outputs for a long period. If enough entropy existed in those samples, then the DSA PRNG will become as strong as ever against attack.

Leaking Input Effects Just as with ANSI X9.17, the DSA PRNG leaks the effects of unguessable inputs in its output. Consider an attacker who has compromised the PRNG's state. The application feeds in an input that the attacker can't guess (e.g., a sample with 90 bits of entropy). If the attacker sees the next output, he doesn't need to guess the sample, because the only effect on future outputs this sample can have is through that output. Note that if the new X_{i+1} depended directly on W_i and X_i, this weakness wouldn't exist. An attacker who knew the state could still try *guessing* the entropy sample, but if he did not guess the right value, he would lose knowledge of the state.

The Iterative Guessing Attack This PRNG is vulnerable to an iterative guessing attack after the state has been compromised. That is, if an attacker knows X_i and knows that W_i has only 20 bits of entropy, he can mount a 2^{20} search, and have a list of 2^{20} 160-bit outputs, one of which is output$[i]$. Note that the attacker needs only a function of the output that he can check, such as a DSA signature made with output$[i]$ as its secret parameter value. Note also that knowledge of the correct value for output$[i]$ also uniquely determines the value of X_{i+1}.

Backtracking If an attacker knows X_i, and output$[i-1]$, then he is clearly able to backtrack to knowledge of X_{i-1}. This doesn't immediately gain him much, since he has to already know output$[i-1]$ to be able to do this. However, in some circumstances, this could turn out to be useful.

Filling in the Gaps Consider a situation in which the attacker knows X_i, X_{i+2}, and output$[i+1]$, but still needs to know output$[i]$. In this case he can solve for output$[i]$ directly:

$$\text{output}[i] = X_{i+2} - X_i - 2 - \text{output}[i+1]$$

Summary The DSA standard's PRNG appears to be quite secure when used in the application for which it was designed: DSA signature parameter generation. However, it doesn't perform well as a general-purpose cryptographic PRNG because it handles its inputs poorly, and because it recovers more slowly from state compromise than it should.

To adapt the DSA PRNG to more general use, the following measures would eliminate most of the attacks we have observed:

1. Require hashing of all PRNG inputs before applying them.
2. Update X by the following formula:

$$X_{i+1} = X_i + \text{hash}(\text{output}[i] + W_i).\ \text{modulo}\ 2^{160}$$

3.3 The RSAREF PRNG

The PRNG included with RSAREF 2.0 is built almost entirely around two operations: MD5 hashing and addition modulo 2^{128}. It is the most conceptually simple design of any we have analyzed. The RSAREF 2.0 PRNG consists of the following:

1. A 128 bit counter, C_i.
2. A method for processing inputs. To process input X_i, we do the following:

$$C_{i+1} = C_i + \text{MD5}(X_i)\ \text{modulo}\ 2^{128}.$$

3. A method for generating outputs. To generate output *output*$[i]$, we do the following:

$$output[i] = \text{MD5}(C_i)\ \text{modulo}\ 2^{128}$$
$$C_{i+1} = C_i + 1\ \text{modulo}\ 2^{128}.$$

Direct Cryptanalytic Attack We will treat MD5 as a random function. While there have been interesting cryptanalytic results on MD5 in the last several years, none of them offer an obvious way to attack the RSAREF PRNG.

Partial Precomputation Attack There is a straightforward attack on a counter-mode generator of this kind: an attacker chooses some number of successive outputs, t, that he expects to see, and then computes the hash of every tth possible counter value. He is guaranteed to see one of these hashes after t outputs; at that point, he knows the whole counter value. This attack is impractical for a 128-bit counter, but it gives an upper bound on the strength of this generator. With 2^{32} outputs, an attacker would need to do a 2^{96} precomputation to mount the attack; with 2^{48} outputs, he would need to do a 2^{80} precomputation. These attacks also require a great deal of memory, though time/memory trade-offs can reduce that.

Timing Attack The C code to add to and increment the 128-bit internal counter has the property that it will leak some information about the resulting 128-bit counter by how many 8-bit add operations the computer must execute. This opens a timing channel for an attacker.

An attacker able to observe the time taken to generate each new output can learn how many zero bytes are in the counter each time it is incremented. This is simply a matter of determining how many bytewise additions had to be done to increment the counter properly. There are two facets to this attack. First, counter values that are all-zero in their low-order few bytes leak a great deal of information through the timing channel; these can be considered a kind of weak state. Second, when combined with the partial precomputation attack discussed above, the timing information can be used to know when to bother checking the PRNG output against precomputed table. This is a small advantage.

Input-Based Attacks We note that several input-based attacks are possible against RSAREF's PRNG. In particular, chosen input attacks exist against the RSAREF PRNG. They become quite powerful when the attacker can also monitor precise timing information from the machine on which the PRNG is running.

Shortening the Cycle with a Chosen-Input Attack An attacker can force the RSAREF PRNG into a shortened cycle by choosing the input value properly. Let input_n be a chosen input for the PRNG such that $\text{MD5}(\text{input}_n)$ has all ones in its low-order n bytes. If an attacker requests a long sequence of outputs by requesting these inputs once per output, he forces the PRNG to cycle much faster, because the low-order n bytes of the counter are fixed. Thus, for $n = 8$, the cycle length is shortened to 2^{64} outputs. Note that the attacker doesn't know what those n bytes are, but he *does* know that they are the same every time the PRNG uses them to generate another output.

A more powerful way to shorten the cycle takes advantage of the birthday paradox. Suppose x_1, x_2 are two chosen inputs such that $\text{MD5}(x_1)+\text{MD5}(x_2)$ has all ones in its low-order n bytes. Then an attacker can feed the periodic sequence $x_1, x_2, x_1, x_2, \ldots$ as inputs to the RSAREF PRNG and observe the outputs; with this procedure, he should see a cycle after about 2^{128-8n} outputs. For example, for the case $n = 16$, it takes about 2^{64} offline work to find suitable x_1, x_2, if an attacker uses an efficient collision search algorithm (see e.g. [OW95,OW96]); this choice of chosen inputs will force the RSAREF generator to repeat immediately[3].

More generally, we can get a simple "time travel" attack: if no new inputs were mixed in during the last j outputs, then the attacker can send the RSAREF PRNG back in time j steps by finding two chosen inputs whose MD5 digests sum to $-j$ (again with the same time complexity).

A Timing + Chosen Input Attack A much more powerful attack is available if the attacker can monitor precise operation timings, and if MD5 operates in constant time. The counter increment operation in the RSAREF source code will leak how many zero bytes are in the resulting counter value by how many 8-bit additions were required, and thus, by how long the counter increment operation took. During the counter increment operation (unlike the add operation used to combine in entropy from a input), detecting n 8-bit additions means that the resulting low-order $n-1$ bytes are zero.

The attack occurs in two stages: in the *precomputation* stage, which is done once, the attacker generates the chosen entropy values he is to use later, and also generates a table of hashed counter values. In the *Execution* stage, which is done each time he wishes to attack some RSAREF PRNG state, he uses those chosen entropy values to force the internal counter to a value that has its low-order 104 bits set to all zeros. The attack requires 2^{48} offline trial hashes and 2000 chosen-entropy requests.

The precomputation stage works as follows:

1. For $n = 1$ to 12, the attacker finds $\text{input}_{0,n}, \text{input}_{1,n}$ such that

   $$\text{MD5}(\text{input}_{0,n}) + \text{MD5}(\text{input}_{1,n})$$

   is all ones in its low-order n bytes, and that its next lowest order byte is even. This is expected to take about 2^{4n} effort using a collision-search algorithm.

The stage of executing the attack works as follows:

1. The attacker watches increment timing values until he knows that the low-order byte of the counter is a zero. (He can see this, because of the extra addition operation, which alters the time taken for the input to be processed.)
2. For $n = 1$ to 12, he does the following:
   (a) He requests update with input_n. This forces the counter value to be all ones in its low n bytes.

[3] We note that MD5 is designed for only 64 bits of collision-resistance, and so perhaps might not be expected to provide more than 64 bits of security. However, this PRNG appears to be in use for generating 1024-bit RSA moduli and establishing triple-DES keys, so it is apparently being trusted for more than 64 bits of security.

(b) He requests an output value, and observes the time taken for the output generation, inferring how many times the PRNG executed an 8-bit add operation in the increment. He keeps requesting the update with input$_n$ and the output, until he gets $n+2$ 8-bit add operations, instead of $n+1$.
(c) At this point, he has forced the low $n+1$-bytes to zeros.

3. At the end of the above loop, the attacker has forced the low-order thirteen bytes of the counter to zero values. He now carries out a brute-force search of the remaining three bytes of C, and breaks the PRNG.

State Compromise Extension Attacks

Losing Entropy in Inputs The PRNG's input-processing mechanism has a potentially dangerous flaw: it is order-independent. That is, updating the PRNG with A, and then with B, is the same as updating it first with B, and then with A. This flaw was originally discovered by Paul Kocher [Koc95,Bal96], but it is still worth noting here. The effect of this is to make the PRNG more likely to start in an insecure state, and also to make the PRNG require considerably more entropy in its inputs before its state is unguessable.

Iterative Guessing The iterative guessing attack works here. That is, if an attacker has compromised C_i, each time the user updates his state with some X_i guessable by an attacker, and then generates an output, $output[i+1]$, which the attacker can see (even if that output is used as a symmetric encryption or authentication key, or as a key or pad encrypted under a public-key) he can maintain his knowledge of the PRNG's state. If the RSAREF PRNG manages to get updated with an unguessable input between a compromised state and a visible output, however, then he loses his knowledge of the state.

Backtracking The RSAREF PRNG is vulnerable to backtracking in a straightforward way. The iterative guessing attack works exactly as well backward as forward, and when an attacker doesn't have new entropy samples, backtracking is exactly as easy as walking the generator forward.

3.4 Summary

The RSAREF 2.0 PRNG is vulnerable to chosen-input attacks which can force it into short cycles, chosen-input timing attacks which can reveal its secret state, and iterative guessing and backtracking attacks which can allow an attacker to extend his knowledge of the secret state backward and forward through time. It also must be used very carefully, due to the fact that inputs affect it in an order-independent way.

To minimize the danger of these attacks, we make the following recommendations:

1. Guard against chosen-input attacks in the design of the system that uses the RSAREF PRNG.

2. Be careful using the RSAREF PRNG in situations where timing information might be leaked.
3. Append a current timestamp and/or a counter to all inputs before sending them into the PRNG, to eliminate the order-independence of PRNG inputs.

3.5 Cryptolib's PRNGs

Cryptolib is a cryptographic library developed primarily by Jack Lacy, Donald Mitchel, William Schnell, and Matt Blaze, and initially described in [LMS93]. The primary source of randomness in Cryptolib is TrueRand, a mechanism for pulling (hopefully) unpredictable values out of the clock skew between different timers available to the system. These values can be used directly (though the documentation warns callers not to rely on more than 16 bits of entropy per 32-bit word), or can be used to seed one of the available pseudorandom number generators, fsrRand or desRand.

fsrRand and desRand are not PRNGs by our definition, but rather are stream ciphers. That is, they do not have defined mechanisms for processing additional inputs "on the fly," but rather are seeded once and then run to generate pseudorandom numbers. This is not unreasonable, given the assumption that TrueRand delivers truly random bits as needed–the system designer can simply generate a whole new state every few minutes, and otherwise needn't worry about entropy collection. When combined, TrueRand and fsrRand or TrueRand and desRand can be analyzed in the same way as the other PRNGs in this paper. That is, we assume that the system initializes the state of either fsrRand or desRand using TrueRand, and uses one of these mechanisms to generate whatever pseudorandom values are needed, and that the whole mechanism is periodically reinitialized from TrueRand. TrueRand is thus the source of PRNG inputs, and fsrRand or desRand is the source of PRNG outputs.

Description of Algorithms

fsrRand fsrRand is described in [LMS93]. Its secret state consists of a secret DES key, K, and an array of seven 32-bit values, $X_{0..6}$, organized as a shift-register. Each time an output is required, two of the 32-bit values are taken and concatenated to form a 64-bit value. This value is encrypted with DES under the secret key. The resulting ciphertext is split into two 32-bit halves; one half is XORed back into one of the 32-bit values (in the same way a shift register value might be updated), the other half is output. The register is then shifted, so that two new values will be used to generate the next output. A more complete description can be found in [LMS93].

desRand desRand appears in the Cryptolib source code (version 1.2). Its secret state consists of a 64-bit counter, C, a secret three-key triple-DES key, K, a secret 20-byte prefix, P, and a secret 20-byte suffix, S. Each new 32-bit output is generated as follows:

1. Use the SHA1 hash function to compute $hash(P, C, S)$.
2. Use triple-DES to compute $E_K(C)$.
3. XOR together the high-order bytes of the hash value with the result from the encryption; output the high-order four bytes of this result.
4. Set $C = C + 1$.

Direct Cryptanalysis

fsrRand There is a direct cryptanalytic attack on fsrRand requiring 2^{89} effort. The attack uses the fact that, once the attacker knows K and any one PRNG output, he can build a table of the 2^{32} possible halves of the DES ciphertext that was used for feedback. For each value, he gets a whole 64-bit ciphertext, which he can decrypt into a 64-bit plaintext, yielding both 32-bit values from the array.

1. The attacker guesses the key, K.
2. The attacker gets the output when the shift register pairs used are (X_i, X_j), (X_j, U), and (V, X_i) for some other U and V. In the pair (A, B), A will be updated with the feedback.
3. For the first two output values, the attacker computes all 2^{32} possible feedback values (the 32-bit half of the DES ciphertext that was not output). This allows him to compute X_j. For each K guess, we expect there to be only one pair of feedback guesses that leads to the same X_j value.
4. The attacker uses the feedback value from the first output (learned in the previous
   step) to compute what the new X_i value should be. He then mounts another 2^{32} guess of the feedback value for the third step, and uses this to derive the current X_i and other register value. If he has the wrong K value, he expects not to find any matching value for X_i; if he has the right K value, he expects to find one value that agrees.

This demonstrates a certificational weakness in fsrRand, at most; the computational requirements are very probably outside the reach of any attacker right now[4].

desRand We are not aware of any direct cryptanalytic attacks available on desRand. The desRand design appears to us to be very conservative, and unlikely to be attacked in the future. Note that nothing like the timing attack on RSAREF's PRNG is available here, despite the use of a counter.

[4] It could be argued that since DES has only 56 bits of strength, this construction was intended for no more strength than that. We find this argument unconvincing. fsrRand was clearly an attempt to get more than 56 bits of strength from DES; otherwise, DES in OFB- or counter-mode would have been used.

Input-Based Attacks These systems accept input only once, and accept it directly from TrueRand or a buffer provided by the caller. This (re)initializes the PRNG. In the context of the following discussion, a known-input attack means that the attacker has learned how to predict some TrueRand values. Clearly, if the attacker can know all the TrueRand values, there is no cryptanalysis to be performed. An interesting result occurs if the PRNG becomes weak with only a small number of predictable TrueRand values.

fsrRand An attacker who knows any two X values used as a plaintext block for DES can mount a keysearch attack, and reduce the possible number of keys to about 2^{24}. He must then wait until the first of those values makes it into the DES input again, and carry out an additional 2^{32} search per candidate key; this will determine the key uniquely. This requires a total of about 2^{57} trial encryptions, and about 2^{24} blocks of memory. From this point, the attacker can quickly recover the remaining state of the PRNG. An attacker who can guess any two such X values with 2^t work can mount the same attack with 2^{57+t} trial encryption and 2^{24} blocks of memory.

An attacker who knows the key, K can recover the remaining PRNG state with a 2^{33} effort, using the same method described for direct cryptanalysis of the PRNG, above.

A more subtle concern might involve flaws in the quality of seed values from TrueRand. Consider an attacker who knows, for a given system, that only 2^8 32-bit outputs from TrueRand are possible. If fsrRand is reseeded directly from TrueRand, this leads to fairly simple attack: fsrRand's DES key must come from TrueRand, and the attacker can quickly list all possible 56-bit values that could have been generated, getting about 2^{16} of them. He can then carry out the attack described above. In general, if there are 2^m possible values for the fsrRand's DES key to get, then the attack will take 2^{m+33} trial decryptions. This is an improvement for $m < 56$, naturally.

Note that this demonstrates that fsrRand doesn't profit from the full entropy it receives during reseeding; In the example above, fsrRand would get 8 bits of entropy per 32-bit word used to reseed it, for a total of 112 bits of entropy.

desRand We are aware of no reasonable known-input attacks on desRand. An attacker with knowledge of C, P, and S, but not K, appears to have no chance to defeat the PRNG; similarly, and attacker with knowledge of C and K, but not P or S, appears to have no chance to defeat the PRNG.

State Compromise Extension Attacks The desRand and fsrRand generators don't process inputs, and so can never recover from state compromise. However, if TrueRand is used to generate a whole new state every few minutes, the scope of a state compromise is made very small. It is worth noting that both desRand and fsrRand allow an attacker in possession of their current state to go backward as well as forward, learning all values ever generated by them. That is, if the PRNG state ever is compromised, the attacker can learn every output ever generated by that state. If the system is designed to reinitialize its PRNG with TrueRand values once every hour, then this means a compromise of all PRNG outputs for that hour. If the system reinitializes the PRNG more frequently, then the attacker learns fewer outputs; if less frequently, then the attacker learns more outputs.

Summary Assuming TrueRand is a good source of unpredictable values, the PRNGs built by putting it together with either fsrRand or desRand appear to be of reasonable strength, but desRand appears to be more resistant to various attacks than fsrRand. Note, however, that nearly all of these attacks require keysearching DES or doing some similarly computationally expensive task.

We make the following recommendations:

1. System designers should verify both by statistical analysis and by an analysis of their target systems' designs whether TrueRand will reliably provide unpredictable numbers on their systems. (This holds true for every source of unpredictable inputs, for every PRNG.)
2. In environments where TrueRand's outputs may be suspect (perhaps due to malicious actions by the attacker), we recommend that desRand, rather than fsrRand, be employed.

4 Summary, Conclusions, and Open Problems

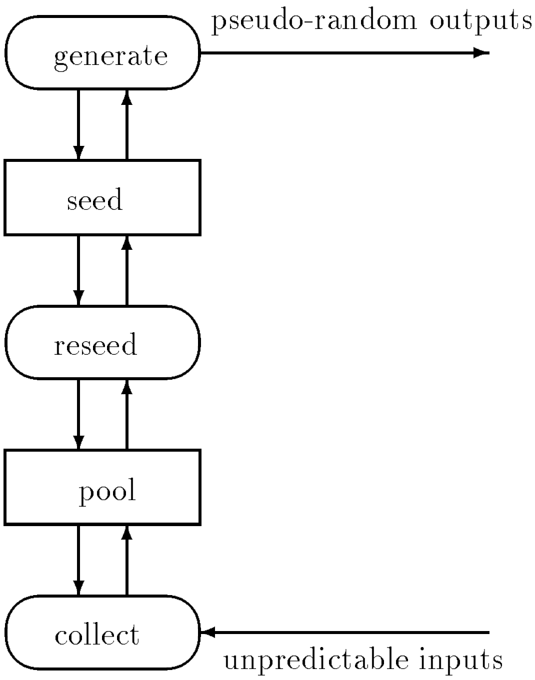
In this paper, we have argued for treating PRNGs as their own kind of cryptographic primitive, distinct from stream ciphers, hash functions, and block ciphers. We have discussed the requirements for a PRNG, developed abstract attacks against an idealized PRNG, and then demonstrated those attacks against four real-world PRNGs.

4.1 Guidelines for Using Vulnerable PRNGs

In the earlier sections, we discussed possible countermeasures for many of the attacks we had developed. Here, we propose a list of ways to protect a PRNG against each of the classes of attacks we discussed.

1. **Use a hash function to protect vulnerable PRNG outputs.** If a PRNG is suspected to be vulnerable to direct cryptanalytic attack, then outputs from the PRNG should be preprocessed with a cryptographic hash function. Note that not all possible flawed PRNGs will be secure even after hashing their outputs, so this doesn't guarantee security, but it makes security much more likely.
2. **Hash PRNG inputs with a counter or timestamp before use.** To prevent most chosen-input attacks, the inputs should be hashed with a timestamp or counter, before being sent into the PRNG. If this is too expensive to be done every time an input is processed, the system designer may want to only hash inputs that could conceivably be under an attacker's control.

Fig. 3. Generalized PRNG, with periodic reseeding



3. **Occasionally generate a new starting PRNG state.** For PRNGs like ANSI X9.17, which leave a large part of their state unchangeable once initialized, a whole new PRNG state should occasionally be generated from the current PRNG. This will ensure that any PRNG can fully reseed itself, given enough time and input entropy.
4. **Pay special attention to PRNG starting points and seed files.** The best way to resist all the state-compromise extension attacks is simply never to have the PRNG's state compromised. While it's not possible to guarantee this, system designers should spend a lot of effort on starting their PRNG from an unguessable point, handling PRNG seed files intelligently, etc. (See [Gut98] for several ways that this can be done.)

4.2 Guidelines for Designing a PRNG

Having described a set of possible attacks on PRNGs, it is reasonable to try to discuss ways to develop new PRNGs that will resist them. We propose the following guidelines for developing new PRNGs:

1. **Base the PRNG on something strong.** The PRNG should be designed so that a successful direct cryptanalytic attack implies a successful attack

on some cryptographic primitive that's believed to be strong. Ideally, this would be proven.

2. **Make sure the whole PRNG state changes over time.** The whole secret internal state should change over time. This prevents a single state compromise from being unrecoverable.
3. **Do "catastrophic reseeding" of the PRNG.** The part of the internal state that is used to generate outputs should be separate from the entropy pool. The generation state should be changed only when enough entropy has been collected to resist iterative guessing attacks, according to a conservative estimate. Figure 3 depicts a possible architecture for implementing catastrophic reseeding.
4. **Resist backtracking.** The PRNG should be designed to resist backtracking. Ideally, this would mean that output t was unguessable in practice to an attacker who compromised the PRNG state at time $t+1$. It may also be acceptable to simply pass the PRNG's state through a one-way function every few outputs, limiting the possible scope of any backtracking attack.
5. **Resist Chosen-Input Attacks.** The inputs to the PRNG should be combined into the PRNG state in such a way that, given an unguessable sequence of inputs, an attacker who starts knowing the PRNG state but not the input sequence, and an attacker who starts knowing the input sequence but not the state, are both unable to guess the ending state. This provides some protection against both chosen-input and state compromise extension attacks.
6. **Recover from Compromises Quickly.** The PRNG should take advantage of every bit of entropy in the inputs it receives. An attacker wanting to learn the effect on the PRNG state of a sequence of inputs should have to guess the entire input sequence.

4.3 Open Problems

In this paper, we've begun the process of systematically analyzing PRNGs. However, there are several interesting areas we haven't dealt with here:

1. **Dedicated PRNG Designs.** Early in this paper, we made the assertion that PRNGs are a distinct kind of cryptographic primitive. Existing PRNGs are almost all built out of existing cryptographic primitives. This raises the question of whether it makes sense to build dedicated PRNG algorithms. Typically, the motivation for building a dedicated algorithm is to improve performance. Are there applications where the PRNG's performance is a serious enough issue to merit a new algorithm?
2. **Security Proofs.** Since most currently-fielded PRNGs are based on existing cryptographic primitives, it would be nice to see some security proofs, demonstrating that mounting some class of attack on the PRNG is equivalent to breaking an underlying block cipher, stream cipher, or hash function.
3. **Starting Points.** One likely way for an attacker to compromise the PRNG state is for the PRNG to be started in a guessable state. This raises the issue of how a designer can ensure that his system always starts its PRNG at an

unguessable state. We would like to see more discussion of these issues in the literature.

4. **Seed Compromise.** We would like to see more discussion of how to resist state compromises in fielded systems. This is an enormous practical issue, which has received relatively little attention in the literature.
5. **Analyzing Other PRNGs.** There are many PRNGs we have not discussed here, mainly due to time and space constraints. In particular, we would like to see a complete discussion of the class of PRNG used in PGP and Gutmann's Cryptlib, among other places. These PRNGs fit into our model, but look very different than any of the systems we have reviewed here: they typically maintain a considerably larger state (or "pool"), in hopes of accumulating large amounts of entropy.
6. **Developing New PRNGs.** We have discussed flaws in existing PRNGs. We are interested in seeing new designs proposed that resist our attacks. A PRNG of our own is currently under development; details will be posted to `http://www.counterpane.com` as they become available.

5 Acknowledgements

The authors would like to thank Greg Guerin, Peter Gutmann, and Adam Shostack for helpful conversations and comments on early drafts of this paper, and Ross Anderson and several anonymous referees for helpful suggestions on improving the paper's presentation.

References

[Agn88] G.B. Agnew, "Random Source for Cryptographic Systems," *Advances in Cryptology — EUROCRYPT '87 Proceedings,* Springer-Verlag, 1988, pp. 77–81.

[ANSI85] ANSI X 9.17 (Revised), "American National Standard for Financial Institution Key Management (Wholesale)," American Bankers Association, 1985.

[Bal96] R.W. Baldwin, "Proper Initialization for the BSAFE Random Number Generator," *RSA Laboratories Bulletin*, n. 3, 25 Jan 1996.

[Dai97] W. Dai, Crypto++ library, `http://www.eskimo.com/~weidai/cryptlib.html`.

[DIF94] D. Davis, R. Ihaka, and P. Fenstermacher, "Cryptographic Randomness from Air Turbulience in Disk Drives," *Advances in Cryptology — CRYPTO '94 Proceedings,* Springer-Verlag, 1994, pp. 114–120.

[ECS94] D. Eastlake, S.D. Crocker, and J.I. Schiller, "Randomness Requirements for Security," RFC 1750, Internet Engineering Task Force, Dec. 1994.

[FMK85] R.C. Fairchild, R.L. Mortenson, and K.B. Koulthart, "An LSI Random Number Generator (RNG)," *Advances in Cryptology: Proceedings of CRYPTO '84,* Springer-Verlag, 1985, pp. 203–230.

[Gud85] M. Gude, "Concept for a High-Performance Random Number Generator Based on Physical Random Noise," *Frequenz*, v. 39, 1985, pp. 187–190.

[Gut98] P. Gutmann, "Software Generation of Random Numbers for Cryptographic Purposes," Proceedings of the 1998 Usenix Security Symposium, 1998, to appear.

[Koc95] P. Kocher, post to sci.crypt Internet newsgroup (message-ID pckDIr4Ar.L4z@netcom.com), 4 Dec 1995.

[LMS93] J.B. Lacy, D.P. Mitchell, and W.M. Schell, "CryptoLib: Cryptography in Software," *USENIX Security Symposium IV Proceedings,* USENIX Association, 1993, pp. 237–246.

[NIST92] National Institute for Standards and Technology, "Key Management Using X9.17," NIST FIPS PUB 171, U.S. Department of Commerce, 1992.

[NIST93] National Institute for Standards and Technology, "Secure Hash Standard," NIST FIPS PUB 180, U.S. Department of Commerce, 1993.

[NIST94] National Institute for Standards and Technology, "Digital Signature Standard," NIST FIPS PUB 186, U.S. Department of Commerce, 1994.

[OW95] P.C. van Oorschot and M.J. Wiener, "Parallel collision search with application to hash function and discrete logarithms," *2nd ACM Conf. on Computer and Communications Security*, New York, NY, ACM, 1994.

[OW96] P.C. van Oorschot and M.J. Wiener, "Improving implementable meet-in-the-middle attacks by orders of magnitude," *CRYPTO '96*, Springer-Verlag, 1996.

[Plu94] C. Plumb, "Truly Random Numbers, *Dr. Dobbs Journal*, v. 19, n. 13, Nov 1994, pp. 113-115.

[Ric92] M. Richterm "Ein Rauschgenerator zur Gweinnung won quasi-idealen Zufallszahlen fur die stochastische Simulation," Ph.D. dissertation, Aachen University of Technology, 1992. (In German.)

[RSA94] RSA Laboratories, RSAREF cryptographic library, Mar 1994, ftp://ftp.funet.fi/pub/crypt/cryptography/asymmetric/rsa/rsaref2.tar.gz.

[SV86] M. Santha and U.V. Vazirani, "Generating Quasi-Random Sequences from Slightly Random Sources," *Journal of Computer and System Sciences,* v. 33, 1986, pp. 75–87.

[Sch96] B. Schneier, *Applied Cryptrography,* John Wiley & Sons, 1996.

[Zim95] P. Zimmermann, *The Official PGP User's Guide,* MIT Press, 1995.