Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator

John Kelsey, Bruce Schneier, and Niels Ferguson

Counterpane Systems; 101 E Minnehaha Parkway, Minneapolis, MN 55419, USA;
{*kelsey,schneier,niels*}*@counterpane.com*

Abstract. We describe the design of Yarrow, a family of cryptographic pseudo-random number generators (PRNG). We describe the concept of a PRNG as a separate cryptographic primitive, and the design principles used to develop Yarrow. We then discuss the ways that PRNGs can fail in practice, which motivates our discussion of the components of Yarrow and how they make Yarrow secure. Next, we define a specific instance of a PRNG in the Yarrow family that makes use of available technology today. We conclude with a brief listing of open questions and intended improvements in future releases.

1 Introduction

Random numbers are critical in every aspect of cryptography. Cryptographers design algorithms such as RC4 and DSA, and protocols such as SET and SSL, with the assumption that random numbers are available. Even as straightforward an application as encrypting a file on a disk with a passphrase typically needs random numbers for the salt to be hashed in with the passphrase and for the initialization vector (IV) used in encrypting the file. To encrypt e-mail, digitally sign documents, or spend a few dollars worth of electronic cash over the internet, we need random numbers.

Specifically, random numbers are used in cryptography in the following applications:

- Session and message keys for symmetric ciphers, such as triple-DES or Blowfish.
- Seeds for routines that generate mathematical values, such as large prime numbers for RSA or ElGamal-style cryptosystems.
- Salts to combine with passwords, to frustrate off-line password guessing programs.
- Initialization vectors for block cipher chaining modes.
- Random values for specific instances of many digital signature schemes, such as DSA.
- Random challenges in authentication protocols, such as Kerberos.
- Nonces for protocols, to ensure that different runs of the same protocol are unique; e.g., SET and SSL.

Some of those random numbers will be sent out in the clear, such as IVs and random challenges. Other of those random numbers will be kept secret, and used as keys for block ciphers. Some applications require a large quantity of random numbers, such as a Kerberos server generating thousands of session keys every hour, and others only a few. In some cases, an attacker can even force the random generator to generate thousands of random numbers and send them to him.

Unfortunately, random numbers are very difficult to generate, especially on computers that are designed to be deterministic. We thus fall back on *pseudorandom*[1] numbers. These are numbers that are generated from some (hopefully random) internal values, and that are very hard for an observer to distinguish from random numbers.

Given the importance of generating pseudo-random numbers for cryptographic applications, it is somewhat surprising that little formal cryptanalysis of these generators exist. There are methodologies for generating randomness on computer systems [DIF94,ECS94], and ad hoc designs of generators [Gut98], but we are aware of only one paper cryptanalyzing these designs [KSWH98a].

[1] It is important to distinguish between the meaning of pseudorandom numbers in normal programming contexts, where these numbers merely need to be reasonably random-looking, and in the context of cryptography, where these numbers must be indistinguishable from real random numbers, even to observers with enormous computational resources.

1.1 What is a Cryptographic PRNG?

In our context, a random number is a number that cannot be predicted by an observer before it is generated. If the number is to be in the range $0 \ldots 2^n - 1$, an observer cannot predict that number with probability any better than $1/2^n$. If m random numbers are generated in a row, an observer given any $m-1$ of them still cannot predict the m'th with any better probability than $1/2^n$. More technical definitions are possible, but they amount to the same general idea.

A cryptographic pseudorandom number generator, or PRNG, is a cryptographic mechanism for processing somewhat-unpredictable inputs, and generating pseudorandom outputs. If designed, implemented, and used properly, even an attacker with enormous computational resources should not be able to distinguish a sequence of PRNG outputs from a random sequence of bits.

There are a great many PRNGs in use in cryptographic applications. Some of them (such as Peter Gutmann's PRNG in Cryptlib [Gut98], or Colin Plumb's PRNG in PGP [Zim95]) are apparently pretty well designed. Others (such as the RSAREF 2.0 PRNG [RSA94], or the PRNG specified in ANSI X9.17 [NIST92]) are appropriate for some applications, but fail badly when used in other applications [KSWH98a].

A PRNG can be visualized as a black box. Into one end flow all the internal measurements (samples) which the system designer believed might be unpredictable to an attacker. Out of the other end, once the PRNG believes it is in an unguessable state, flow apparently random numbers. An attacker might conceivably have some knowledge or even control over some of the input samples to the PRNG. An attacker might have compromised the PRNG's internal state at some point in the past. An attacker might have an extremely good model of the "unpredictable" values being used as input samples to the PRNG, and a great deal of computational power to throw at the problem of guessing the PRNG's internal state.

Internally, a PRNG needs to have a mechanism for processing those (hopefully) unpredictable samples, a mechanism for using those samples to update its internal state, and a mechanism to use some part of its internal state to generate pseudorandom outputs. In some PRNG designs, more-or-less the same mechanism does all three of these tasks; in others, the mechanisms are clearly separated.

1.2 Why Design a New PRNG?

We designed Yarrow because we are not satisfied with existing PRNG designs. Many have flaws that allowed attacks under some circumstances (see [KSWH98a] for details on many of these). Most of the others do not seem to have been designed with attacks in mind. None implement all the defenses we have worked out over the last two years of research into PRNGs.

Yarrow is an enhancement of a proprietary PRNG we designed several years ago for a client. We kept improving our design as we discovered new potential attacks.

1.3 A Guide to the Rest of the Paper

The remainder of this paper is as follows: In Section 2 we discuss the reasons behind our design choices for Yarrow. In Section 3 we discuss the various ways that cryptographic PRNGs can fail in practice. Then, in Section 4, we will discuss the basic components of Yarrow, and show how they resist the kinds of failures listed earlier. Section 5 gives the generic design ideas and their rationale. Finally, we will consider open questions relating to Yarrow, and plans for future releases.

In the full paper we will define Yarrow-160, a precisely defined PRNG, and discuss entropy calculation.

2 Yarrow Design Principles

Our goal for Yarrow is to make a PRNG that system designers can fairly easily incorporate into their own systems, and that is better at resisting the attacks we know about than the existing, widely-used alternatives.

We pose the following constraints on the design of Yarrow:

1. Everything is reasonably efficient. There is no point in designing a PRNG that nobody will use, because it slows down the application too much.
2. Yarrow is so easy to use that an intelligent, careful programmer with no background in cryptography has some reasonable chance of using the PRNG in a secure way.
3. Where possible, Yarrow re-uses existing building blocks.

Yarrow was created using an attack-oriented design process. This means we designed the PRNG with attacks in mind from the beginning. Block ciphers are routinely designed in this way, with structures intended to optimize their strength against commonly-used attacks such as differential and linear cryptanalysis. The Yarrow design was very much focused on potential attacks. This had to be tempered with other design constraints: performance, flexibility, simplicity, ease of use, portability, and even legal issues regarding the exportability of the PRNG were considered. The result is still a work-in-progress, but it resists every attack of which we are aware, while still being a usable tool for system designers.

We spent the most time working on a good framework for entropy-estimation and reseeding, because this is so critical for the ultimate security of the PRNG, and because it is so often done badly in fielded systems. Our cryptographic mechanisms are nothing very exciting, just various imaginative uses of a hash function and a block cipher. However, they do resist known attacks very well.

2.1 Terminology

At any point in time, a PRNG contains an internal state that is used to generate the pseudorandom outputs. This state is kept secret and controls much of the processing. Analogous to ciphers we call this state the key of the PRNG.

To update the key the PRNG needs to collect inputs that are truly random, or at least not known, predictable or controllable by the attacker. Often used examples include the exact timing of key strokes or the detailed movements of the mouse. Typically, there are a fairly large number of these inputs over time, and each of the input values is fairly small. We call these inputs the samples.

In many systems there are several sources that each produce samples. We therefore classify the samples according to the source they came from.

The process of combining the existing key and new sample(s) into a new key is called the reseeding.

If a system is shut down and restarted, it is desirable to store some high-entropy data (such as the key) in non-volatile memory. This allows the PRNG to be restarted in an unguessable state at the next restart. We call this stored data the seed file.

3 How Cryptographic PRNGs Fail

In this section, we consider some of the ways that a PRNG can fail in a real-world application. By considering how a PRNG can fail, we are able to recognize ways to prevent these failures in Yarrow. In other cases, the failures cannot be totally prevented, but we can make them less likely. In still other cases, we can only ensure a quick recovery from the compromised state.

3.1 How PRNGs are Compromised

Once the key of a PRNG is compromised, its outputs are predictable; at least until it gets enough new samples to derive a new, unguessable key. Many PRNGs have the property that, once compromised, they will never recover, or they will recover only after a very long time.

For these reasons, it makes sense to consider how a PRNG's key can be compromised, and how, once keys are compromised, they may be exploited.

Entropy Overestimation and Guessable Starting Points We believe that this is the most common failing in PRNGs in real-world applications. It is easy to look at a sequence of samples that appears random and has a total length of 128 bits, feed it into the PRNG, and then start generating output. If that sequence of samples turns out only to have 56 bits of entropy, then an attacker could feasibly perform an exhaustive search for the starting point of the PRNG.

This is probably the hardest problem to solve in PRNG design. We tried to solve it by making sure that the entropy estimate is very conservative. While it is still possible to seriously overestimate the starting entropy, it is much less likely to happen, and when it does the estimate is likely to be closer to the actual value. We also use a computationally-expensive reseeding process to raise the cost of attempting to guess the PRNG's key.

Mishandling of Keys and Seed Files Keys and seed files are easy to mishandle in various ways, such as by letting them get written to the swap file by the operating system, or by opening a seed file, but failing to update it every time it is used. The Yarrow design provides some functions to simplify the management of seed files. An excellent discussion of some methods for avoiding key compromise appears in [Gut98].

Implementation Errors Another way that the key of the PRNG can be compromised is by exploiting some implementation error. Errors in the implementation are impossible to prevent. The only preventative measures we found for Yarrow was to try to make the interface reasonably simple so that the programmer trying to use Yarrow in a real-world product can use it securely without understanding much about how the PRNG works.

This is an area we are still working on. It is notoriously difficult to make security products easy to use for most programmers, and of course, it is very hard to be certain there are no errors in the Yarrow generator itself.

One thing we can do is to make it easy to verify the correct implementation of a Yarrow PRNG. We have carefully designed Yarrow to be portable and precisely defined. This allows us to create test vectors that can be used to verify that a Yarrow implementation is in fact working correctly. Without such test vectors an implementor would never be able to ensure that her Yarrow implementation was indeed working correctly.

Cryptanalytic Attacks on PRNG Generation Mechanisms Between reseedings, the PRNG output generation mechanism is basically a stream cipher. Like any other stream cipher, it is possible that the one used in a PRNG will have some cryptanalytic weakness that makes the output stream somewhat predictable or at least recognizable. The process of finding weaknesses in this part of the PRNG is the same as finding them in a stream cipher.

We have not seen a lot of PRNGs that were easily vulnerable to this kind of attack. Most PRNGs' generation mechanisms are based on strong cryptographic mechanisms already. Thus, while this kind of attack is always a concern, it usually does not seem to break the PRNG. To be safe, we have designed Yarrow to be based on a block cipher; if the block cipher is secure, then so is the generation mechanism. This was done because there are quite a number of apparently-secure block ciphers available in the public domain.

Side-Channel Attacks Side-channel attacks are attacks that use additional information about the inner workings of the implementation [KSWH98b]: timing attacks [Koc96], and power analysis [Koc98] are typical examples. Many PRNGs that are otherwise secure fall apart when any additional information about their internal operations are leaked. One example of this is the RSAREF 2.0 PRNG, which can be implemented in a way that is vulnerable to a timing attack.

It is probably not possible to protect against side-channel attacks in the design of algorithms. However, we do try to avoid obvious weaknesses, specifically any data-dependent execution paths.

Chosen-Input Attacks on the PRNG An attacker is not always limited to just observing PRNG outputs. It is sometimes possible to gain control over some of the samples sent into the PRNG, especially in a tamper-resistant token. Some PRNGs, such as the RSAREF 2.0 PRNG, are vulnerable to such attacks. In the worst case the attacker can mount an adaptive attack in which the samples are selected based on the output that the PRNG provides. To avoid this kind of attack in Yarrow, all samples are processed by a cryptographic hash function, and are combined with the existing key using a secure update function.

3.2 How Compromises are Exploited

Once the key is compromised, it is interesting to consider how this compromise is exploited. Since it is not always possible to prevent an attacker from learning the key, it is reasonable to spend some serious time and effort making sure the PRNG can recover its security from a key compromise.

Permanent Compromise Attacks Some PRNGs, such as the one proposed in ANSI X9.17, have the property that once the key has been compromised, an attacker is forever after able to predict their outputs. This is a terrible property for a PRNG to have, and we have made sure that Yarrow can recover from a key compromise.

Iterative Guessing Attacks If the samples are mixed in with the key as they arrive, an attacker who knows the PRNG key can guess the next "unpredictable" sample, observe the next PRNG output, and test his guess by seeing if they agree. This means that a PRNG which mixes in samples with 32 bits of entropy every few output words will not recover from a key compromise until the attacker is unable to see the effects of three or four such samples on the outputs. This is called an iterative guessing attack, and the only way to resist it is to collect entropy samples in a pool separate from the key, and only reseed the key when the contents of the entropy pool is unguessable to any real-world attacker. This is what Yarrow does.

Backtracking Attacks Some PRNGs, such as the RSAREF 2.0 PRNG, are easy to run backwards as well as forward. This means that an attacker that has compromised the PRNG's key *after* a high-value RSA key pair was generated can still go back and learn that high-value key pair. We include a mechanism in Yarrow to limit backtracking attacks to a limited number of output bytes.

Compromise of High-Value Keys Generated From Compromised Key Of course, the biggest cost of a compromised PRNG is that it leads to compromised system-keys if the key generation process uses the PRNG. If the key that is being generated is very valuable, the harm to the system owner can be very large. As we mentioned, the iterative guessing attacks require us to collect entropy in a pool before reseeding the generator with it. When we are about to generate a very valuable key, it is preferable to have whatever extra entropy there is in the PRNG's key. Therefore, the user can request an explicit reseed of the generator. This feature is intended to be used rarely and only for generating high-value secrets.

4 The Yarrow Design: Components

In this section, we discuss the components of Yarrow, and how they interact. A major design principle of Yarrow is that its components are more-or-less independent, so that systems with various design constraints can still use the general Yarrow design.

The use of algorithm-independent components in the top level design is a key concept in Yarrow. Our goal is not to increase the number of security primitives that a cryptographic system is based on, but to leverage existing primitives as much as possible. Hence, we rely on one-way hash functions and block ciphers, two of the best-studied and most widely available cryptographic primitives, in our design.

There are four major components:

1. An **Entropy Accumulator** which collects samples from entropy sources, and collects them in the two pools.
2. A **Reseed Mechanism** which periodically reseeds the key with new entropy from the pools.
3. A **Generation Mechanism** which generates PRNG outputs from the key.
4. A **Reseed control** that determines when a reseed is to be performed.

Below, we specify each component's role in the larger PRNG design, we discuss the requirements for each component in terms of both security and performance, and we discuss the way each component must interact with each other component. Later in this paper, we will discuss specific choices for these components.

4.1 Design Philosophy

We have seen two basic design philosophies for PRNGs.

One approach assumes that it is usually possible to collect and distill enough entropy from the samples that each of the output bits should have one bit of real entropy. If more output is required than entropy has been collected from the samples, the PRNG either stops generating outputs or falls back on a cryptographic mechanism to generate the outputs. Colin Plumb's PGP PRNG and Gutmann's Cryptlib PRNG both fall into this category. In this kind of design, entropy is accumulated to be immediately reused as output, and the whole PRNG mechanism may be seen as a mechanism to distill and measure entropy from various sources on the machine, and a buffer to store this entropy until it is used.

Yarrow takes a different approach. We assume that we can accumulate enough entropy to get the PRNG into an unguessable state (without such an assumption, there is no point designing a PRNG). Once at that starting point, we believe we have cryptographic mechanisms that will generate outputs an attacker cannot distinguish from random outputs. In our approach, the purpose of accumulating entropy is to be able to recover from PRNG key compromises. The PRNG is designed so that, once it has a secure key, even if all other entropy accumulated is predictable by, or even under the control of, an attacker, the PRNG is still secure. This is also the approach taken by the RSAREF, DSA, and ANSI X9.17 PRNGs.

The strength of the first approach is that, if properly designed, it is possible to get unconditional security from the PRNG. That is, if the PRNG really does accumulate enough entropy to provide for all its outputs, even breaking some strong cipher like triple-DES will not be sufficient to let an attacker predict unknown PRNG outputs. The weakness of the approach is that the strength of the PRNG is based in a critical way on the mechanisms used to estimate and distill entropy. While this is inevitably true of all PRNGs, with a design like Yarrow we can afford to be far more conservative in our entropy estimates, since we are not expecting to be able to distill enough entropy to provide

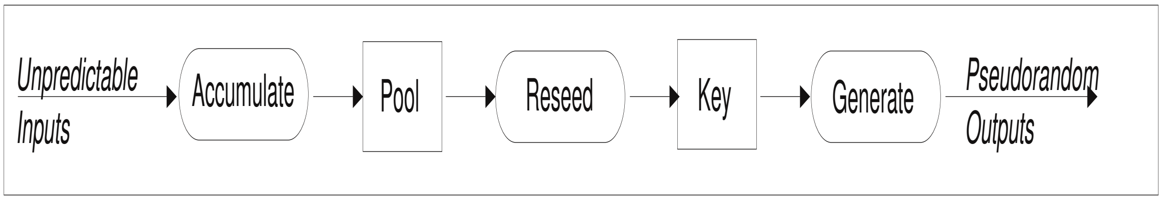

Fig. 1. Generic block diagram of Yarrow

for all our outputs. In our opinion, entropy estimation is the hardest part of PRNG design. By contrast, the design of a generation mechanism that will resist cryptanalysis is a relatively easy task, making use of available cryptographic primitives such as a block cipher.

Practical cryptographic systems rely on the strength of various algorithms, such as block ciphers, stream ciphers, hash functions, digital signature schemes, and public key ciphers. We feel that basing the strength of our PRNG on well-trusted cryptographic mechanisms is as reasonable as basing the strength of our systems on them.

This approach raises two important issues, which should be made explicit:

1. Yarrow's outputs are cryptographically derived. Systems that use Yarrow's outputs are no more secure than the generation mechanism used. Thus, unconditional security is not available in systems like one-time pads, blind signature schemes, and threshold schemes. Those mechanisms are capable of unconditional security, but an attacker capable of breaking Yarrow's generation mechanism will be able to break a system that trust Yarrow outputs to be random. This is true even if Yarrow is accumulating far more entropy from the samples than it is producing as output.

2. Like any other cryptographic primitive, a Yarrow generator has a limited strength which we express in the size of the key. Yarrow-160 relies on the strength of three-key triple-DES and SHA-1, and has an effective key size of about 160 bits. Systems that have switched to new cryptographic mechanisms (such as the new AES cipher, when it is selected) in the interests of getting higher security should also use a different version of Yarrow to rely on those new mechanisms. If a longer key is necessary, then a future "larger" version of Yarrow should be used; it makes no sense to use a 160-bit PRNG to generate a 256-bit key for a block cipher, if 256 bits of security are actually required.

4.2 Entropy Accumulator

Entropy Accumulation Entropy accumulation is the process by which a PRNG acquires a new, unguessable internal state. During initialization of the PRNG, and for reseeding during operation, it is critical that we successfully accumulate entropy from the samples. To avoid iterative guessing attacks and still regularly reseed the PRNG it is important that we correctly estimate the amount of entropy we have collected thus far. The entropy accumulation mechanism must also resist chosen-input attacks, in the sense that it must not be possible for an attacker who controls some of the samples, but does not know others, to cause the PRNG to lose the entropy from the unknown samples.

In Yarrow, entropy from the samples is collected into two *pools*, each a hashing context. The two pools are the *fast* pool and the *slow* pool; the fast pool provides frequent reseeds of the key, to ensure that key compromises have as short a duration as possible when our entropy estimates of each source are reasonably accurate. The slow pool provides rare, but extremely conservative, reseeds of the key. This is intended to ensure that even when our entropy estimates are very optimistic, we still eventually get a secure reseed. Alternating input samples are sent into the fast and slow pools.

Each pool contains the running hash of all inputs fed into it since it was last used to carry our a reseed.

In Yarrow-160, the pools are each SHA-1 contexts, and thus are 160 bits wide. Naturally, no more than 160 bits of entropy can be collected in these pools, and this determines the design strength of Yarrow-160 to be no greater than 160 bits.

The following are the requirements for the entropy accumulation component:

1. We must expect to accumulate nearly all entropy from the samples, up to the size of a pool, even when the entropy is distributed in various odd ways in those samples, e.g., always in the last bit, or no entropy in most samples, but occasional samples with nearly 100 bits of entropy in a 100-bit sample, etc.
2. An attacker must not be able to choose samples to undo the effects of those samples he does not know on a pool.
3. An attacker must not be able to force a pool into any kind of weak state, from which it cannot collect entropy successfully.
4. An attacker who can choose which bits in which samples will be unknown to him, but still has to allow n unknown bits, must not be able to narrow down the number of states in a pool to substantially fewer than 2^n.

Note that this last condition is a very strong requirement. This virtually requires the use of a cryptographic hash function.

Entropy Estimation Entropy estimation is the process of determining how much work it would take an attacker to guess the current contents of our pools. The general method of Yarrow is to group the samples into sources and estimate the entropy contribution of each source separately. To do this we estimate the entropy of each sample separately, and then add these estimates of all samples that came from the same source.

The assumption behind this grouping into sources is that we do not want our PRNG's reseeding taking place based on only one source's effects. Otherwise, one source which appears to provide lots of entropy, but instead provides relatively little, will keep causing the PRNG to reseed, and will leave it vulnerable to an iterative guessing attack. We thus allow a single fast source to cause frequent reseeding from the fast pool, but not the slow pool. This ensures that we reseed frequently, but if our entropy estimates from our best source are wildly inaccurate, we still will eventually reseed from the slow pool, based on entropy estimates of a different source. Recall that samples from each source alternate between the two pools.

Implementors should be careful in determining their sources. The sources should not be closely linked or exhibit any significant correlations.

The entropy of each sample is measured in three ways:

- The programmer supplies an estimate of entropy in a sample when he writes the routine to collect data from that source. Thus, the programmer might send in a sample, with an estimate of 20 bits of entropy.
- For each source a specialized statistical estimator is used to estimate the entropy of the sample. This test is geared towards detecting abnormal situations in which the samples have a very low entropy.
- There is a system-wide maximum “density” of the sample, by considering the length of the sample in bits, and multiplying it by some constant factor less than one to get a maximum estimate of entropy in the sample. Currently, we use a multiplier of 0.5 in Yarrow-160.

We use the smallest of these three estimates as the entropy of the sample in question.

The specific statistical tests used depends on the nature of the source and can be changed in different implementations. This is just another component, which can be swapped out and replaced by better-suited components in different environments.

4.3 Generating Pseudorandom Outputs

The Generation Mechanism provides the PRNG output. The output must have the property that, if an attacker does not know the PRNG's key, he cannot distinguish the PRNG's output from a truly random sequence of bits.

The generation mechanism must have the following properties:

- Resistant to cryptanalytic attack,
- efficient,
- resistant to backtracking after a key compromise,
- capable of generating a very long sequence of outputs securely without reseeding.

4.4 Reseed Mechanism

The Reseed Mechanism connects the entropy accumulator to the generating mechanism. When the reseed control determines that a reseed is required, the reseeding component must update the key used by the generating mechanism with information from one or both of the pools being maintained by the entropy accumulator, in such a way that if either the key or the pool(s) are unknown to the attacker before the reseed, the key will be unknown to the attacker after the reseed. It must also be possible to make reseeding computationally expensive to add difficulty to attacks based on guessing unknown input samples.

Reseeding from the fast pool uses the current key and the hash of all inputs to the fast pool since the last reseed (or since startup) to generate a new key. After this is done, the entropy estimates for the fast pool are all reset to zero.

Reseeding from the slow pool uses the current key, the hash of all inputs to the fast pool, and the hash of all inputs to the slow pool, to generate a new key. After this is done, the entropy estimates for both pools are reset to zero.

4.5 Reseed control

The Reseed Control mechanism must weigh various considerations. Frequent reseeding is desirable, but it makes an iterative guessing attack more likely. Infrequent reseeding gives an attacker that has compromised the key more information. The design of the reseed control mechanism is a compromise between these goals.

We keep entropy estimates for each source as the samples have gone into each pool. When any source in the fast pool has passed a threshhold value, we reseed from the fast pool. In many systems, we would expect this to happen many times per hour. When any k of the n sources have hit a higher threshhold in the slow pool, we reseed from the slow pool. This is a much slower process.

For Yarrow-160, the threshhold for the fast pool is 100 bits, and for the slow pool, is 160 bits. At least two different sources must be over 160 bits in the slow pool before the slow pool reseeds, by default. (This should be tunable for different environments; environments with three good and reasonably fast entropy sources should set $k = 3$.)

5 The Generic Yarrow Design and Yarrow-160

In this section, we describe the generic Yarrow design. This is a generic description, using an arbitrary block cipher and hash function. If both algorithms are secure, and the PRNG gets sufficient starting entropy, our construction results in a strong PRNG. We also discuss the specific parameters and primitives used in Yarrow-160.

We need two algorithms, with properties as follows:

- A one-way hash function, $h(x)$, with an m-bit output size,
- A block cipher, $E()$, with a k-bit key size and an n-bit block size.

The hash function is assumed to have the following properties:

- Collision intractable.
- One-way.
- Given any set M of possible input values, the output values are distributed as $|M|$ selections of the uniform distribution over m-bit values.

The last requirements implies several things. Even if the attacker knows most of the input to the hash function, he still has no effective knowledge about the output unless he can enumerate the set of possible inputs. It also makes it impossible to control any property of the output value unless you have full control over the input.

The block cipher is assumed to have the following properties:

- It is resistant to known-plaintext and chosen-plaintext attacks, even those requiring enormous numbers of plaintexts and their corresponding ciphertexts,
- Good statistical properties of outputs, even given highly patterned inputs.

The strength (in bits) of the resulting PRNG is limited by $\min(m, k)$. In practice even this limit will not quite be reached. The reason is that if you take an m bit random value and apply a hash function that produces m bits of output, the result has less than m bits of entropy due to the collisions that occur. This is a very minor effect, and overall results in the loss of at most a few bits of entropy. We ignore this small constant factor, and say that the PRNG has a strength of $\min(m, k)$ bits.

Yarrow-160 uses the SHA1 hash function for $h()$, and three-key triple-DES for $E_K()$.

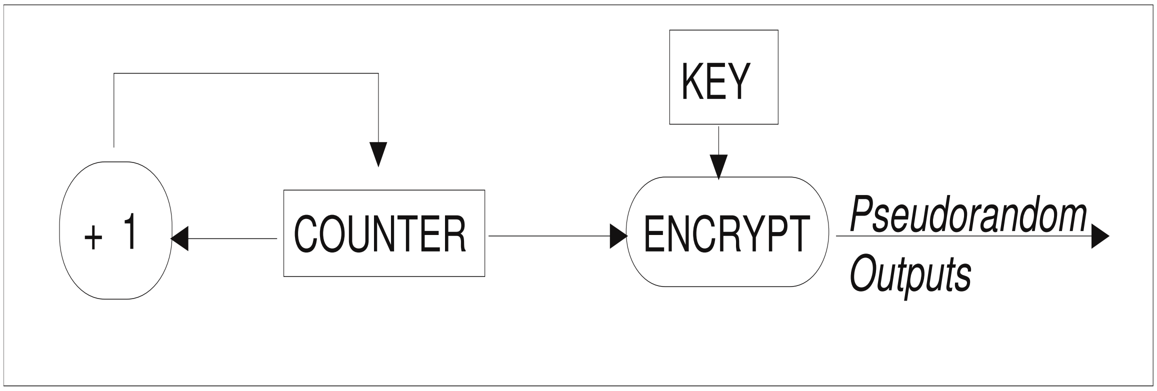

Fig. 2. Generation mechanism

5.1 Generation Mechanism

Figure 2 shows the generator which is based on using the block cipher in counter mode.

We have an n-bit counter value C. To generate the next n-bit output block, we increment C and encrypt it with our block cipher, using the key K. To generate the next output block we thus do the following:

$$C \leftarrow (C+1) \bmod 2^n$$
$$R \leftarrow E_K(C)$$

where R is the next output block and K is the current PRNG key.

If the key is compromised at a certain point in time, the PRNG must not leak too many 'old' outputs that were generated before the compromise. It is clear that this generation mechanism has no inherent resistance to this kind of attack. For that reason, we keep count of how many blocks we have output. Once we reach some limit P_g (a system security parameter, $1 \leq P_g \leq 2^{n/3}$), we generate k bits of PRNG output, and use them as the new key.

$$K \leftarrow \text{Next } k \text{ bits of PRNG output}$$

We call this operation a generator gate. Note that this is not a reseeding operation as no new entropy is introduced into the key.

In the interests of keeping an extremely conservative design, the maximum number of outputs from the generator between reseedings is limited to $\min(2^n, 2^{k/3}P_g)$ n-bit output blocks. The first term in the minimum prevents the value C from cycling. The second term makes it extremely unlikely that K will take on the same value twice. In practice, P_g should be set much lower than this, e.g. $P_g = 10$, in order to minimize the number of outputs that can be learned by backtracking.

In Yarrow-160, we use three-key triple-DES in counter mode to generate outputs, and plan to apply the generator gate every ten outputs. (That is, $P_g = 10$.)

Security Arguments

Normal Operations Consider an attacker who can, after seeing a long sequence of outputs from this generator under the same key K, extract the key. This can be converted into a chosen plaintext attack on the cipher to extract its key.

Consider an attacker who can, after seeing a long sequence of outputs from this generator under the same key, predict a single future or past output value. The algorithm used by the attacker performs a chosen-plaintext attack on the underlying block cipher, allowing the prediction of (part of) one ciphertext after some number of encryptions of chosen plaintexts have been seen. This is enough of a demonstrated weakness to rule the cipher out for many uses, e.g. in CBC-MAC.

Backtracking Protection Consider an attacker who can use the outputs after a generator gate has taken place to mount an attack on the data generated before the generator gate. The same attacker can mount his attack on the generator without the generator gate by using k known bits of the generator output to form a new key, using that key to generate a sequence of outputs, and then applying the attack. (This is possible as the counter value C is assumed to be known to the attacker.) Thus, a generator gate cannot expose previous output values to attack without also demonstrating a weakness in the generation mechanism in general.

Consider an attacker who compromises the current key of the PRNG somehow. Suppose he can learn a previous key from the current key. To do this, he must be able to extract the key of the block cipher given a small number of bits of the generator's output. Thus, the attacker must defeat the generator mechanism to defeat the generator gate mechanism.

Consider an attacker who can predict the next key generated by the generator gate. The same method he uses to do this can be used to predict the next PRNG output, if the generator is used without generator gate.

Limits on Generator Outputs As the number of output blocks from the basic generator available to the attacker grows closer to and beyond $2^{n/2}$ it becomes easier and easier to distinguish the cipher's outputs from a real random sequence. A random sequence should have collisions in some n-bit output blocks, but there will be no repetitions of output blocks in the output from running a block cipher in counter mode. This means that a conservative design should re-key long before this happens. This is the reason why we require the generator gate to be used at least once every $2^{n/3}$ output blocks. Note that P_g is a configurable parameter and can be set to smaller values. Smaller values of P_g increase the number of generator gates and thus decrease the amount of old data an attacker can retrieve if he were to find the current key. The disadvantage of very small P_g values is that performance suffers, especially if a block cipher is used that has an expensive key schedule.

Each time we use the generator gate, we generate a new key from the old key using a function that we can assume to behave as a random function. This function is not the same function for each generator gate, as the counter C changes in value. There are therefore no direct cycles for K to fall into. Any cycle would require C to wrap around, which we do not allow between reseedings. To be on the safe side we do restrict the number of generator gate operations to $2^{k/3}$ which makes it extremely unlikely that the same value K will be used twice between reseedings.

Implementation ideas The use of counter mode allows several output blocks to be computed together, or even in parallel. A hardware implementation can exploit this parallelism using a pipelined design, and software implementations could use a bit-sliced implementation of the block cipher for higher performance. Even for simple software implementations it might very well be more efficient to produce many blocks at a time and to buffer the output in a secure memory area. This improves the locality of the code, and can improve the cache-hit ratio of the program.

5.2 Entropy Accumulator

To accumulate the entropy from a sequence of inputs, we concatenate all the inputs. Once we have collected enough entropy we apply the hash function h to the concatenation of all inputs. We alternate applying samples from each source to each pool.

In Yarrow-160, we use the SHA1 hash function to accumulate inputs in this way. We alternate feeding inputs from each source into the fast and slow pools; each pool is its own SHA1 hash context, and thus effectively contains the SHA1 hash of all inputs fed into that pool.

Security Arguments If we believe that an attacker cannot find collisions in the hash function, then we must also believe that an attacker cannot be helped by any collisions that exist.

Consider the situation of an attacker trying to predict the whole sequence of inputs to be fed into the user's entropy accumulator. The attacker's best strategy is to try to generate a list of the most likely input sequences, in order of decreasing probability. If he can generate a list that is feasible for him to search through which has a reasonable probability (say, a 10^{-6} chance) of containing the actual sequence of samples, he has a worthwhile attack. Ultimately, an attacker in this position cannot be resisted effectively by the design of the algorithm, though we do our best. He can only be resisted by the use of better entropy sources, and by better estimation of the entropy in the pool.

Now, how can the entropy accumulator help the attacker? Only by reducing the total number of different input sequences he must test. However, in order for the attacker to see a single pair of different input sequences that will lead to the same entropy pool contents he must find a pair of distinct input sequences that have the same hash value.

Implementation ideas All common hash functions can be computed in an incremental manner. The input string is usually partitioned into fixed size blocks, and these blocks are processed sequentially by the hash function. This allows an implementation to compute the hash of the sequence of inputs on the fly. Instead of concatenating all inputs and applying the hash function in one go (which would require an unbounded amount of memory) the software can use a fixed size buffer and compute the hash partially whenever the buffer is full.

As with the generator mechanism, the locality of the code can be improved by using a buffer that is larger than one hash function input block. The entropy accumulator would thus accumulate several blocks worth of samples before hashing the entire buffer.

The entropy accumulator should be careful not to generate any overflows while adding up the entropy estimates. As there is no limit on the number of samples the accumulator might have to process between two reseeds the implementation has to handle this case.

5.3 Reseed Mechanism

The reseeding mechanism generates a new key K for the generator from the entropy accumulator's pool and the existing key. The execution time of the reseed mechanism depends on a parameter $P_t \geq 0$. This parameter can either be fixed for the implementation or be dynamically adjusted.

The reseed process consists of the following steps:

1. The entropy accumulator computes the hash on the concatenation of all the inputs into the fast pool. We call the result v_0.
2. Set $v_i := h(v_{i-1}|v_0|i)$ for $i = 1, \ldots, t$.
3. Set $K \leftarrow h'(h(v_{P_t}|K), k)$.
4. Set $C \leftarrow E_K(0)$.
5. Reset all entropy estimate accumulators of the entropy accumulator to zero.
6. Wipe the memory of all intermediate values
7. If a seed file is in use, the next $2k$ bits of output from the generator are written to the seed file, overwriting any old values.

Step 1 gathers the output from the entropy accumulator. Step 2 uses an iterative formula of length P_t to make the reseeding computationally expensive if desired. Step 3 uses the hash function h and a function h', which we will define shortly, to create a new key K from the existing key and the new entropy value v_{P_t}. Step 4 defines the new value of the counter C.

The function h' is defined in terms of h. To compute $h'(m, k)$ we construct

$$\begin{aligned} s_0 &:= m \\ s_i &:= h(s_0|\ldots|s_{i-1}) \qquad i = 1, \ldots \\ h'(m,k) &:= \text{first } k \text{ bits of } (s_0|s_1|\ldots) \end{aligned}$$

This is effectively a 'size adaptor' function that converts an input of any length to an output of the specified length. If the input is larger than the desired output, the function takes the leading bits of the input. If the input is the same size as the output the function is the identity function. If the input is smaller than the output the extra bits are generated using the hash function. This is a very expensive type of PRNG, but for the small sizes we are using this is not a problem.

There is no security reason why we would set a new value for the counter C. This is done to allow more implementation flexibility and still maintain compatibility between different implementations. Setting the counter C makes it simple for an implementation to generate a whole buffer of output from the generator at once. If a reseed occurs, the new output should be derived from the new seed and not from the old output buffer. Setting a new C value makes this simple: any data in the output buffer is simply discarded. Simply re-using the existing counter value is not compatible as different implementations have different sizes of output buffers, and thus the counter has been advanced to different points. Rewinding the counter to the virtual 'current' position is error-prone.

To reseed the slow pool, we feed the hash of the slow pool into the fast pool, and then do a reseed. In general, this slow reseed should have P_t set as high as is tolerable.

In Yarrow-160, this is done as described above, but using SHA1 and triple-DES. We generate a three-key triple-DES key from the hash of the contents of the pool or pools used, and the current key.

Security Arguments Consider an attacker who starts out knowing the generator key but not the contents of the entropy pool hash v_0. The value v_{P_t} is a pure function of v_0, so the attacker has no real information about v_{P_t}. This value is then hashed with K, and the result is size-adjusted to be the new key. As the result of the hash has as much entropy as v_{P_t} has, the attacker loses his knowledge about K.

Consider an attacker in the opposite situation: he starts out knowing the samples that have been processed, but not the current generator key. The attacker thus knows v_{P_t}. However, an attacker with no knowledge of the key K cannot predict the result of the hash, and thus ends up knowing nothing about the new key.

5.4 Reseed control

The reseed control module determines when a reseed is to be performed. An explicit reseed occurs when some application explicitly asks for a reseed operation. This is intended to be used only rarely, and only by applications that generate very high-valued random secrets. Access to the explicit reseed function should be restricted in many cases.

The reseed periodically occurs automatically. The fast pool is used to reseed whenever any of its sources have an entropy estimate of over some threshhold value. The slow pool is used to reseed whenever at least two of its sources have entropy estimates above some other threshhold value.

In Yarrow-160, the fast pool threshhold is 100 bits, and the slow pool threshhold is 160 bits. Two sources must pass the threshhold for the slow pool to reseed.

6 Open Questions and Plans for the Future

Yarrow-160, our current construction, is limited to at most 160 bits of security by the size of its entropy accumulation pools. Three-key triple-DES has known attacks considerably better than brute-force; however, the backtracking prevention mechanism changes keys often enough that the cipher still has about 160 bits of security in practice.

At some point in the future, we expect to see a new block cipher standard, the AES. Yarrow's basic design can easily accommodate a new block cipher. However, we will also have to either change hash functions, or come up with some special hash function construction to provide more than 160 bits of entropy pool. For AES with 128 bits, this will not be an issue; for AES with 192 bits or 256 bits, it will have to be dealt with. We note that the generic Yarrow framework will accomodate the AES block cipher and a 256-bit hash function (perhaps constructed from the AES block cipher) with no problems.

In practice, we expect any weaknesses in Yarrow-160 to come from poorly estimating entropy, not from cryptanalysis. For that reason, we hope to continue to improve the Yarrow entropy estimation mechanisms. This is the subject of ongoing research; as better estimation tools become available, we will upgrade Yarrow to use them.

We still have to create a reference implementation of Yarrow-160, and create test vectors for various parameter sets. These test vectors will test all aspects of the generator. This will probably require the use of Yarrow-160 versions with different parameters then the ones used in Yarrow-160; the details of this remain to be investigated.

The reseed control rules are still an ad-hoc design. Further study might yield an improves set of reseed control rules. This is the subject of ongoing research.

7 On the Name "Yarrow"

Yarrow is a flowering perennial with distinctive flat flower heads and lacy leaves, like Queen Anne's Lace or wild carrot. Yarrow stalks have been used for divination in China since the Hsia dynasty, in the second millennium B.C.E. The fortuneteller would divide a set of 50 stalks into piles, then repeatedly use modulo arithmetic to generate two bits of random information (but with a nonuniform distribution).

Here is the full description of the method: The most notable things are: one, it takes an amazing amount of effort to generate two random bits; and two, it does not produce a flat output distribution, but, apparently, 1/16 - 3/16 - 5/16 - 7/16.

The oracle is consulted with the help of yarrow stalks. These stalks are short lengths of bamboo, about four inches in length and an eighth inch in diameter. Fifty stalks are used for this purpose. One is put aside and plays no further part. The remaining 49 stalks are first divided into two random heaps. One stalk is then taken from the right-hand heap and put between the ring finger and the little finger of left hand. Then the left-hand heap is placed in the left hand, and the right hand takes from it bundles of 4, until there are 4 or fewer stalks remaining. This remainder is placed between the ring finger and the middle finger of the left hand. Next the right-hand heap is counted off by fours, and the remainder is placed between the middle finger and the forefinger of the left hand. The sum of the stalks now between the fingers of the left hand is either 9 or 5. (The various possibilities are $1 + 4 + 4$, or $1 + 3 + 1$, or $1 + 2 + 2$, or $1 + 1 + 3$; it follows that the number 5 is easier to obtain than the number 9.)

At this first counting off of the stalks, the first stalk—held between the little finger and the ring finger—is disregarded as supernumerary, hence one reckons as follows: 9 = 8, or 5 = 4. The number 4 is regarded as a complete unit, to which the numerical value 3 is assigned. The number 8, on the other hand, is regarded as a double unit and is reckoned as having only the numerical value 2.Therefore, if at the first count 9 stalks are left over, they count as 2; if 5 are left, they count as 3. These stalks are now laid aside for the time being.

Then the remaining stalks are gathered together again and divided anew. Once more one takes a stalk from the pile on the right and places it between the ring finger and the little finger of the left hand; then one counts off the stalks as before. This time the sum of the remainders is either 8 or 4, the possible combinations being 1 + 4 + 3, or 1 + 3 + 4, or 1 + 1 + 2, or 1 + 2 + 1, so that this time the chances of obtaining 8 or 4 are equal. The 8 counts as a 2, the 4 counts as a 3. The procedure is carried out a third time with the remaining stalks, and again the sum of the remainders is 8 or 4.

Now from the numerical values assigned to each of the three composite remainders, a line is formed with a total value of 6, 7, 8, or 9.

Yarrow stalks are still used for fortunetelling in China, but with a greatly simplified method: shake a container of 100 numbered yarrow stalks until one comes out. This random number is used as an index into a table of fortunes.

8 Acknowledgements

We wish to thank Christopher Allen, Steve Bellovin, Matt Blaze, Jon Callas, Bram Cohen, Dave Grawrock, Alexey Kirichenko, David Wagner, and James Wallner for useful comments on the Yarrow design. We would also like to thank Ari Y. Benbasat for implementing the preliminary Windows version of Yarrow.

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.

[BDR+96] M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener, "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security," January 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*, USENIX Association, 1998, pp. 243–257.

[Kah67] D. Kahn, *The Codebreakers, The Story of Secret Writing*, Macmillan Publishing Co., New York, 1967.

[Koc95] P. Kocher, post to `sci.crypt` Internet newsgroup (message-ID pck-DIr4Ar.L4z@netcom.com), 4 Dec 1995.

[Koc96] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Advances in Cryptology—CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 104–113.

[Koc98] P. Kocher, "Differential Power Analysis," available online from http://www.cryptography.com/dpa/.

[KSWH98a] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic Attacks on Pseudorandom Number Generators," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 168–188.

[KSWH98b] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *ESORICS '98 Proceedings*, Springer-Verlag, 1998, pp. pp 97–110.

[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.

[Luc98] S. Lucks, Private Communication, 1998.

[NIS80] National Institute of Standards and Technology. *DES Modes of Operation*, December 2, 1980. FIPS PUB 81, available from http://www.itl.nist.gov/div897/pubs/fip81.htm.

[NIS93] National Institute of Standards and Technology. *Data Encryption Standard (DES)*, December 30, 1993. FIPS PUB 46-2, available from http://www.itl.nist.gov/div897/pubs/fip46-2.htm.

[NIS95] National Institute of Standards and Technology. *Secure Hash Standard*, April 17, 1995. FIPS PUB 180-1, available from http://www.itl.nist.gov/div897/pubs/fip180-1.htm.

[NIS99] National Institute of Standards and Technology. *Data Encryption Standard (DES)*, 1999. DRAFT FIPS PUB 46-3.

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

[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 Cryptography,* John Wiley & Sons, 1996.

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