Black-Box Assessment of Pseudorandom Algorithms

Derek Soeder
dsoeder@cylance.com

Christopher Abad
cabad@cylance.com

Gabriel Acevedo
gacevedo@cylance.com

Cylance, Inc.
http://cylance.com/

ABSTRACT

We present a survey of the non-cryptographic pseudorandom number generators provided to applications by a variety of highly prevalent platforms. Most of the pseudorandom number generators examined exhibit properties that enable various attacks and techniques, including forward and reverse prediction, seeking, and the recovery of internal state from pseudorandom application output many orders of magnitude more quickly than naive brute-force. We describe the attacks and present examples and algorithms to illustrate their implementation.

1. INTRODUCTION

Despite user sentiment to the contrary, computers are rigidly deterministic and struggle to appear unpredictable when such behavior is intended. A computer's random behavior is therefore termed *pseudorandom*, and the source of this pseudorandomness is typically a pseudorandom number generator, or PRNG.

The core requirements for a PRNG are a) that its output appear unpredictable to a human and b) that it always produce the same sequence of output when initialized with a particular *seed* value. Other desirable properties include a long period—how many numbers the PRNG can yield before its output sequence begins repeating itself—and an absence of obvious correlations between numbers in the sequence. Egregious examples of such correlations have previously been demonstrated in various PRNGs ([14][15][16]). The PRNGs we examined are expected to exhibit a uniform distribution, meaning every value that the PRNG is capable of producing should have equal probability of being produced, although PRNGs that yield nonuniform distributions have also been constructed and statistically analyzed[3].

It is widely recognized in the security community that PRNGs are not adequate for generating random data for use in security-sensitive applications[11], in particular because they accept a small seed initially (typically 32 to 64 bits, often derived from the current time) and by design operate deterministically thereafter, never gathering further entropy. In fact, this problem of a guessable seed has occasionally subjected even a cryptographically secure random number generator (CSRNG) to successful attack[5][12]. Our research contrasts with these historical attacks in that it demonstrates guessing the seed or internal state based on an application's pseudorandom output rather than from foreknowledge of how the seed is derived. We also present a number of attacks that speed this guessing by many orders of magnitude compared to naive brute-force. For some PRNGs we demonstrate the prediction of future pseudorandom output even though the PRNG's complete internal state cannot be efficiently recovered.

Although there have been decades of research exploring attacks on PRNG-based systems, we feel that much of the analysis presented here is novel, and we hope that its embodiment in a relatively simple utility for use in black-box scenarios will advance practitioners' and researchers' ability to practically identify and remediate insecure usage of PRNGs. The solution, of course, is to use a CSRNG in security-relevant code—or as a simpler rule, use CSRNGs whenever possible. This has long been an established best practice, but unfortunately general security awareness among developers tends to lag behind until prompted to improve.

2. PRIOR WORK

We have tried to acknowledge much of the relevant prior work around attacking applications' usage of PRNGs, but last year's Black Hat USA presentation by Argyros and Kiayias[1] deserves particular mention. Their work focused specifically on the PHP PRNGs and on various open-source PHP applications' use of them, and to mount their attacks, they give consideration both to techniques for ascertaining the values used to construct the seed and to state recovery attacks which operate on the target application's pseudorandom output. Our research considers only attacks that operate on pseudorandom output, we assess a somewhat wider variety of prevalent PRNGs, and we present rather different attacks with generally very low computation, memory, and output length requirements.

Readers interested in analyses of more secure alternatives to the PRNGs we consider are recommended to review an examination of the Windows CSRNG by Dorrendorf et al.[8] and inspections of various Java PRNGs presented earlier this

Black Hat USA 2013, Las Vegas, Nevada

year at RSA Conference USA by Meyer et al.[2].

3. HOW APPLICATIONS USE PRNGS

We consider a PRNG to be a simple algorithm for transforming a state stored internally by the PRNG into another state and for producing a pseudorandom number as its output. The state transformation is determinate, as is the process of computing the output from a state, and the state itself is a fixed size for a given PRNG. Computation of the output could take place before, during, or after the state transformation, although every PRNG we have reviewed computes its output after transforming its state.

The internal state of a PRNG is initialized, or *seeded*, from a seed typically derived from external values or supplied by the application. Every PRNG we studied accepts a seed effectively ranging from 31 bits to 64 bits in size. In many cases there is no distinction between seed and initial state; if the seed is the same size as the PRNG's internal state, it will typically be assigned with little or no modification. However in other cases, particularly when the state is larger than the seed, some algorithm is used to initialize the entire state from successive transformations of the seed.

In effect, the PRNG is a component of an application that only requires a seed as input for initialization in order to generate as many pseudorandom numbers as desired. The range of possible values that each output number can take varies among PRNGs, but they tend to be either 15 to 32-bit integers or floating-point numbers in the interval [0.0, 1.0).

Given such a pseudorandom number, the application then needs to process it into a form suitable for the application's intended use. For the cases in which we are most interested, the application wants to emit a sequence of symbols chosen pseudorandomly from a fairly small alphabet. To accomplish this, the application must derive a position in the alphabet from the output number. We will refer to this position as the ordinal value. Ordinal values are typically derived in one of two ways, which we describe below.

For conciseness, we define the term *limit* to mean the count of all possible ordinal values, and we assume throughout that the ordinal values are the set of distinct integers in the interval [0, limit). In summary, then, the application uses a PRNG to generate a pseudorandom number, reduces that number to an ordinal, and maps that ordinal to a symbol via an alphabet.

3.1 Modular or Take-from-Bottom

The simplest way for an application to process a potentially large integer into a comparatively small ordinal is to divide it by the limit and use the remainder as the ordinal. This approach is performed simply by using the modulo operator (typically represented as %), and therefore we refer to it as *modular*. Another way of thinking about the approach is that some least-significant portion of the pseudorandom number is being consumed and the rest discarded, so we preferentially use the more evocative term *take-from-bottom*.

One observation about this approach is that it can introduce detectable biases in a subrange of ordinal values. For example, consider an application using a PRNG that outputs numbers from 0 to 32767 in order to itself output a string of English alphabet letters represented internally by the 26 ordinal values 0 through 25. 32768 is not evenly divisible by 26−there is a remainder of eight (32768 % 26 = 8)−so the first eight letters are slightly favored over the others. Assuming a uniform distribution of PRNG outputs, each of the first eight letters occurs with probability 1261/32768 (3.848%), while each of the other eighteen occurs with probability 1260/32768 (3.845%). With enough output and favorable application behavior, it might be possible to detect such a bias with the purpose of identifying the PRNG and discerning something about the alphabet.

A potentially much more serious consequence of the take-from-bottom approach is what we term a *subgenerator attack*; this and other implementation-dependent attacks are discussed in Section 4.

3.2 Multiplicative or Take-from-Top

Another common approach for converting pseudorandom numbers into ordinals is to multiply the limit by a pseudorandom numerator and divide by the integer just beyond the highest possible pseudorandom number. Put another way, the pseudorandom number is normalized to be a real number in the interval [0.0, 1.0)−a more traditional form preferred by many PRNGs−and it is then multiplied by the limit to yield the interval [0.0, limit). The mantissa is then discarded (the Floor function is applied), leaving an integer in the desired range of ordinal values. We refer to this approach as *multiplicative*, although we hope the reader will find it intuitive if we use the term *take-from-top* to emphasize that the most-significant portion of the pseudorandom number is consumed, and to contrast with the take-from-bottom approach.

Like the take-from-bottom approach, take-from-top can also introduce biases, but spread over the range of ordinal values rather than clumped in the lowest values. These biases arise from the normalized pseudorandom numbers not conforming to a completely uniform distribution over [0.0, 1.0), because in practice, the normalized number is computed as a ratio of two integers of finite size. Continuing the example from Section 3.1, a pseudorandom number from 0 to 32767, when normalized through division by 32768 and then multiplied by 26, will yield ordinal values 0, 3, 6, 9, 13, 16, 19, and 22 each with probability 1261/32768, versus probability 1260/32768 for each of the others. Here, too, eight of the 26 ordinal values are favored, but a mostly different eight from those favored by the take-from-bottom approach.

4. PRNG ANALYSES

The following sections present analyses of a variety of PRNGs to elucidate properties that could be useful to an adversary when attacking an application's pseudorandom behaviors. These analyses are not intended to be exhaustive.

4.1 Microsoft Visual C Runtime (MSVCRT)

The MSVCRT C PRNG, embodied in the srand (initialization) and rand (pseudorandom number generation) functions of VC/crt/src/rand.c, is a simple linear congruential generator (LCG) which resembles the ANSI C suggested implementation. The PRNG can be fully expressed as shown in Listing 1.

The constant 214013 is specified as the LCG's multiplier term, while 2531011 is its increment term. Due to 32-bit integer truncation, its modulus term is implicitly 4294967296 (2^{32}). The constant 65536 (2^{16}) is what we call the *discard divisor*, not present in every LCG but commonly employed to exclude the least significant portion of the state from the output, while 32768 (2^{15}) is what we refer to as the *output modulus*.

Listing 1: MSVCRT PRNG.

```
UInt32 State;

void srand(UInt32 seed)
{
    State = seed;
}

Int16 rand()
{
    // 32-bit integer truncation may occur
    State = (State * 214013) + 2531011;
    return (State / 65536) % 32768;
}
```

This PRNG has a 32-bit internal state, to which the seed is assigned exactly during initialization, and it outputs a non-negative integer from 0 to 32767 inclusive. How this output is consumed is application-dependent: Perl normalizes the output to a real number in the interval [0.0, 1.0) by dividing by 32768, the take-from-top approach discussed in Section 3.2, but we believe it's more common for applications to apply a modulus in the manner of the *take-from-bottom* approach of Section 3.1.

One observation about this PRNG is that its modulus (2^{32}) is divisible by the discard divisor times the output modulus ($2^{16} \cdot 2^{15} = 2^{31}$), and therefore it has an effective modulus of 2147483648 (2^{31}). In other words, the most-significant bit of the state never affects the output. This halves the maximum number of states that need to be considered when attempting to guess which seed could have produced a given output sequence. Some of the attacks described in the following sections will reduce the number of candidate states many orders of magnitude further.

Note that we have not evaluated any of the C++ TR1 PRNGs implemented in VC/include/random.

4.1.1 Biases

The potential for bias mentioned in Sections 3.1 and 3.2 applies particularly well to the MSVCRT PRNG due to its small output modulus of 32768. (The reader might notice we used that very output modulus in the examples.)

If an attacker is able to collect a large amount of pseudorandom output (typically at least tens of thousands of ordinals), and if the limit is not a power of two, then a noticeable bias should eventually emerge. How quickly the bias becomes pronounced, the number of ordinal values exhibiting bias, and the distribution of those biased values should provide clues as to the PRNG's output modulus, the count of ordinal values and their mapping to symbols (especially if multiple ordinal values map to the same symbol), and whether the application employs a take-from-bottom or take-from-top approach.

4.1.2 Subgenerator Attack

Because of the MSVCRT PRNG's choice of parameters, it is potentially susceptible to what we term a *subgenerator attack* when consumed using the take-from-bottom approach. A *subgenerator* is like a PRNG within a PRNG: the subgenerator's modulus (the *submodulus*), which defines the size of its internal state, is a factor of the full PRNG's modulus, and therefore the subgenerator's period is a fraction of the full PRNG's period.

The subgenerator phenomenon occurs when (discard divisor · output modulus) divides the modulus and GCD(limit, modulus / (discard divisor · output modulus)) > 1. In these cases, the submodulus is the product of the GCD times (discard divisor · output modulus). The higher portion of the full PRNG's state has no effect on the output of the subgenerator, which is a sequence of pseudorandom numbers in the interval [0, GCD). Therefore, the subgenerator's initial state can be brute-forced to "fix" the least-significant portion of the full PRNG's initial state, and then the most-significant portion can be brute-forced separately.

Consider again the example of an application emitting a string of pseudorandom letters using the take-from-bottom approach with the MSVCRT PRNG. Since the limit 26 is divisible by two, the least-significant bit of each letter's ordinal value (assuming a standard A-through-Z alphabet) can be considered as though it was generated by a similar PRNG with a modulus of two times the discard divisor−that is, 131072 (2^{17}). This means that only 131072 seeds need to be tested in order to determine which seeds if any produce the observed sequence of least-significant bits; for this step, all the more-significant bits of each ordinal can be ignored. For each matching subgenerator seed, the higher portion of the seed can then be brute-forced to determine which if any produces the observed letter sequence in full. This attack reduces the search space from 2^{31} seeds to $2^{17} + 2^{14}$ seeds, given output of a length sufficient to ensure that only one candidate subgenerator seed is found.

4.1.3 Reversing

If the PRNG's complete state is known, it can be easily reversed to the previous state by subtracting the increment and multiplying by the multiplier's inverse modulo the modulus. For the modulus 2147483648, the multiplicative inverse of the multiplier 214013 is 968044885.

As an example, if the PRNG's current state is 12345678, its next state is

$$((12345678 \cdot 214013) + 2531011)\%2147483648 = 733229785$$

That state's previous state can be computed as

$$((733229785-2531011)\cdot 968044885)\%2147483648 = 12345678$$

Reversing an LCG-type PRNG's state was previously demonstrated by Kamkar[6].

4.1.4 Seeking

The PRNG's state can be efficiently advanced by an arbitrary number of states through computation of the following:

$$state[n] = ((state[0] \cdot A^n) + (C \cdot (A^{n-1})/(A-1)))\%M$$

Where n is the number of states by which to advance, state[0] is an arbitrary state to which the seek is relative, A is the LCG's multiplier constant, C is the LCG's increment constant, and M is the LCG's modulus constant. This equation appears as Formula 3.2.1-6 in Knuth's *Seminumerical Algorithms*[7]. Our implementation uses exponentiation by squaring and computes the right term modulo (M · (A - 1)) prior to dividing by (A - 1). This extends L'Ecuyer's "jumping ahead" technique[9] from multiplicative LCGs to LCGs with a nonzero increment.

To seek backwards using the same function, supply the multiplicative inverse (A^{-1}) for the A parameter, and supply the additive inverse times the multiplicative inverse ($-C \cdot A^{-1}$) for the C parameter.

4.1.5 Partial State Fixation

Each piece of an application's pseudorandom output reveals a portion of its PRNG's internal state as it was at the time the output was generated. Typically the amount of information divulged is small compared to the entirety of the internal state, and the arithmetic performed on the much greater unknown portion obscures the past and future states. However, given one ordinal we can again "fix" a portion of the state and thereby divide the number of possibilities by the limit, as the fixed portion is known to take one specific ordinal from the set of all values. Unlike the subgenerator attack of Section 4.1.2, this attack applies to an LCG-type PRNG without regard to its choice of parameters or the limit being used to process pseudorandom numbers into ordinals.

For example, suppose again that an application outputs a string of pseudorandom A-through-Z letters using straightforward ordinal values. For the string "TQKBKKUDWEVTJYAB" the first ordinal (of 'T') is 19, and so we know that after the PRNG generated that value, its state was such that either (state / 65536) % 26 = 19 (for take-from-bottom) or $((state/65536) \cdot 26)/32768 = 19$ (for take-from-top). Armed with this knowledge, we can then brute-force the state that produces the suffix "QKBKKUDWEVTJYAB" while skipping approximately 25/26 (96%) of the possibilities. Once we find the right state, we can then reverse using the technique of Section 4.1.3 to compute the state that begets the full string. The code shown in Listing 2 illustrates a take-from-bottom implementation of the attack.

Listing 2: MSVCRT take-from-bottom.

```
for (top = 19; top < (2147483648 / 65536); top += 26)
    for (bottom = 0; bottom < 65536; bottom++)
    {
        seed = (top * 65536) + bottom;
        if (string_from_seed(seed) == "QKBKKUDWEVTJYAB")
        {
            seed = ((seed − 2531011) * 968044885) % 2147483648;
            found_seed(seed);
        }
    }
```

The attack could be implemented for a take-from-top application as shown in Listing 3.

Listing 3: MSVCRT take-from-top.

```
start = (((19 * 32768 + 25) / 26) * 65536); // +25 to round up
end = (((20 * 32768 + 25) / 26) * 65536) − 1;

for (seed = start; seed <= end; seed++)
    if (string_from_seed(seed) == "QKBKKUDWEVTJYAB")
    {
        seed = ((seed − 2531011) * 968044885) % 2147483648;
        found_seed(seed);
    }
```

When the subgenerator attack applies, it can be combined with this partial state fixation attack to further accelerate PRNG state recovery.

4.1.6 Improved Partial State Fixation

There is an improvement we can make to the partial state fixation attack that divides the reduced search space by the limit again, but only in a take-from-top application. Because the multiplier is only a fraction of the modulus, incrementing a given candidate state by one—which has the effect of adding the multiplier into the next state—usually does not influence the next state enough to produce an ordinal different from what is produced from the unincremented candidate state. More specifically, each block of approximately (M / A / limit) sequential candidate states will cause the same ordinal to be output. For modulus 2147483648, multiplier 214013, and a limit of 26, this means that we can add as much as 385 or 386 to skip a block of candidate states that does not yield the correct next ordinal. Each time we reach a desirable block, we test its candidate states serially, after which we can skip the next (limit - 1) blocks.

This technique usually cannot be extended to an ordinal beyond the second, because typical choices of LCG parameters specify a multiplier large enough that its powers are not small fractions of the modulus. As a result, for further distances into the output the aforementioned blocks become unitary or generally too small and too scattered to enable recursive application of the attack.

4.2 Java

The Java PRNG implemented in the java.util.Random class is essentially an LCG but with a few idiosyncrasies. It can be expressed as shown in Listing 4.

Java's LCG uses a multiplier of 0x5DEECE66D (the hexadecimal representation of 25214903917), an increment of 0xB (11), and a modulus of 0x1000000000000 (2^{48}). The Next() method, which returns an unconstrained pseudorandom integer, applies a discard divisor of 0x10000 (65536 or 2^{16}) and an implicit output modulus of 2^{32}. Because this method can return negative integers, we feel that programmers are more likely to use the variant depicted as Next(Int32), which accepts an argument expressing the exclusive upper bound for the desired range of pseudorandom integers—what we have referred to more succinctly as the limit. In this way, the Java PRNG spares the programmer the decision of whether to transform the raw pseudorandom numbers via take-from-bottom or take-from-top; in fact, the Next(Int32) method uses both.

Listing 4: Java PRNG.

```
class Random
{
    UInt64 State;

    void Seed(UInt64 seed)
    {
        State = (seed ^ 0x5DEECE66D) % 0x1000000000000;
    }

    Int32 Next()
    {
        State = ((State * 0x5DEECE66D) + 0xB) % 0x1000000000000;
        return (Int32)(State / 0x10000);
    }

    Int32 Next(Int32 limit)
    {
        State = ((State * 0x5DEECE66D) + 0xB) % 0x1000000000000;
        Int32 randint = (State / 0x20000);

        if (limit is a power of 2)
            return ((Int64)randint * limit) / 0x80000000;

        // reject 'randint' in top non-multiple of 'limit' to avoid bias
        while (randint >= 0x80000000 / limit * limit)
        {
            State = ((State * 0x5DEECE66D) + 0xB) %
                    0x1000000000000;
            randint = (State / 0x20000);
        }

        return (randint % limit);
    }
}
```

For a power-of-two limit, the most-significant bits are used in a take-from-top approach, ostensibly because it is known[9] that the less-significant bits have a shorter period given a power-of-two modulus. For other limits, a take-from-bottom approach is used with a discard divisor of 0x20000 (2^{17}), although numbers drawn from the highest portion of the 48-bit number space are skipped to avoid introducing even tiny biases into the PRNG's output.

4.2.1 Subgenerator Attack

The subgenerator attack of Section 4.1.2 applies to the Java PRNG in cases where the chosen limit is a multiple of two but not a power of two. Given sufficient output and a limit which shares a greatest common divisor of two with the modulus, the subgenerator attack should reduce the search space from 2^{48} to $2^{18} + 2^{30}$.

4.2.2 Reversing

The Java PRNG's multiplier and modulus are coprime, so the multiplier has a multiplicative inverse, namely 0xDFE05BCB1365 (246154705703781), which can be used to reverse to the previous PRNG state. However, because the Random.Next(Int32) method in its take-from-bottom mode skips states that would bias the output, a reversing implementation must similarly skip any such states it encounters in order to accurately recover past output.

4.2.3 Seeking

The seeking technique of Section 4.1.4 also applies to the Java PRNG, although as noted above, certain biasing states may need to be accounted for manually, which complicates long-distance seeks.

4.2.4 Partial State Fixation

The partial state fixation attack of Section 4.1.5 is equally applicable to all modes of use of the Java PRNG, with the caveat that the recovered intermediate state must be reversible as described in Section 4.2.2.

4.2.5 Improved Partial State Fixation

The partial state fixation attack improvement described in Section 4.1.6 can be applied to the Java PRNG when the limit is a power of two, because such a limit engages the take-from-top logic which is a prerequisite of the improved attack.

4.2.6 Timing

The special logic in the Java PRNG's take-from-bottom code path, meant to avoid biasing the PRNG's distribution very slightly in favor of the lowest values, consequently causes pseudorandom number generation to take slightly longer when a biasing state is encountered. If the timing difference were observed after generation of a single pseudorandom value, it would reveal that the skipped state was in the interval $[(2^{48}$ - $(2^{31}\%limit) \cdot 2^{17}), 2^{48})$. We don't expect the timing difference to be detectable in realistic Internet-borne attack scenarios, especially when considering that applications typically emit pseudorandom output in chunks. In general, the subgenerator attack (when the limit is even) and the partial state fixation attack are far more useful in those cases where the limit is not a power of two.

4.3 BSD libc

The modern BSD libc PRNG, which is also used on Mac OS X, is a simple multiplicative LCG with a multiplier of 16807 (7^5), a modulus of 2147483647 (2^{31} - 1), and an increment of zero, which is the distinguishing characteristic of a multiplicative LCG. It can be expressed in code as shown in Listing 5.

Listing 5: BSD libc PRNG.

```
UInt32 State;

void srand(UInt32 seed)
{
    if (seed == 0) // don't allow an initial state of zero,
        State = 123459876; // or else every future state will be zero
    else State = seed;
}

Int32 rand()
{
    // avoid 32-bit truncation, as that conflicts with modulus
    State = ((UInt64)State * 16807) % 2147483647;
    return State;
}
```

Due to the choice of parameters, this PRNG exhibits the maximum period of 2147483646 when the seed is nonzero (A seed of zero will cause it, like all multiplicative LCGs, to remain at the zero state forever). Because the modulus is prime, no subgenerator attack is possible.

There also exists an older BSD libc PRNG based on an ANSI C rand() recommendation, an LCG with multiplier 1103515245, increment 12345, modulus 2147483648 (2^{31}), no discard divisor, and an output modulus of 32768 (2^{15}). Analysis of this PRNG is left as an exercise for the reader.

4.3.1 Biases

Because the modulus 2147483647 is prime, the PRNG's distribution is slightly biased for most choices of limit less than the modulus, and a further extremely minor bias is introduced due to the PRNG's inability to output zero. These biases actually cancel out when the limit divides 2147483646 (i.e., when 2147483647 % limit = 1), which for reference factorizes as follows:

$$2147483646 = 2 \cdot 3 \cdot 3 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$$

For any other limit, (2147483647 % limit) - 1 ordinal values occur with probability Ceiling(2147483647 / limit) / 2147483647, while all other ordinal values occur with probability Floor(2147483647 / limit) / 2147483647. The amount of application output required to detect such a bias is clearly prohibitive.

4.3.2 Reversing

The BSD libc's PRNG can be reversed using a multiplicative inverse of 1407677000. Its additive inverse, like its increment, is zero.

4.3.3 Seeking

Seeking a multiplicative LCG is even simpler than seeking an LCG with a nonzero increment, as only modular exponentiation is involved. Compute the n'th state relative to some current state as follows:

$$state[n] = (state[0] \cdot A^n)\%M$$

4.3.4 Partial State Fixation

The partial state fixation attack described previously applies to the BSD libc PRNG.

4.3.5 Improved Partial State Fixation

The improved partial state fixation attack is also effective against the BSD libc PRNG, if the PRNG is being used in a take-from-top application.

4.4 Microsoft VBScript and ASP

Microsoft's VBScript processor (which executes Active Server Pages) implements the VisualBASIC Rnd function as an LCG with a multiplier of 0x00FD43FD (the hexadecimal representation of 16598013 or -179203), an increment of 0x00C39EC3 (12820163 or -3957053), and a 24-bit state, meaning an effective modulus of 0x01000000 (16777216 or 2^{24})−and therefore a period of 16777216 as well. The PRNG can be represented in code as shown in Listing 6.

Listing 6: Microsoft VBScript and ASP PRNG.

```
UInt32 State;

Double Rnd()
{
    // 32-bit integer truncation may occur
    State = ((State * 0x00FD43FD) + 0x00C39EC3) % 0x01000000;
    return (Double)State * 0.000000059604645; // roughly 1/2**24
}
```

Multiplying by 0.000000059604645 is almost the same as dividing by the modulus, but it is different enough that attempting the latter as a shortcut will occasionally produce incorrect output.

Because the PRNG is designed to always return a normalized real number, it can be assumed to operate in the take-from-top mode only, which precludes a subgenerator attack but enables the partial state fixation attack on what is in any case a small seed space.

4.4.1 Biases

The smaller output size of the VBScript PRNG means it exhibits more pronounced biases for non-power-of-two limits, although these biases are still relatively small. For example, given a limit of 26, fourteen (16777216 % 26 = 14) of the possible ordinal values will appear with probability 645278/16777216 (3.846157%), while the other twelve will appear with probability 645277/16777216 (3.846151%).

4.4.2 Reversing

Given the full state of the PRNG, it can be reversed in the typical manner for mixed LCGs described in Section 4.1.3, using a multiplicative inverse of 0x00093155 (602453 or -16174763 in decimal).

4.4.3 Seeking

Seeking to an arbitrary future or past state can be accomplished as described in Section 4.1.4.

4.4.4 Partial State Fixation

The VBScript PRNG is susceptible to the partial state fixation attack.

The improved partial state fixation attack does not significantly apply to the VBScript PRNG due to the large size of the multiplier relative to the modulus. The expression (M / A / limit) of Section 4.1.6 evaluates to zero for any meaningful choice of limit, meaning there are not large blocks of consecutive states that can be skipped.

4.5 Microsoft SQL Server and PHP

Both Microsoft SQL Server's RAND and PHP's session ID generation code use a PRNG that combines the output of two multiplicative LCGs. The first LCG has multiplier 40014 and modulus 2147483563, while the second has multiplier 40692 and modulus 2147483399; both moduli are prime. Effectively, the PRNG has an approximately 62-bit internal state comprising the internal states of both LCGs, and its period should be the product of the two LCGs' moduli, each reduced by one.

Internally, the PRNG computes the first LCG's output minus the second LCG's output, wraps the difference to 2147483562 or less if nonpositive (not the same as modulo 2147483563), and then multiplies the result by 4.656613e-10 (equivalently, divides by 2147483589.46728, which is somewhat greater than the upper bound) to produce pseudorandom floating-point numbers in the approximate interval (0.0, 0.9999999872]. The code in Listing 7 illustrates the algorithm.

4.5.1 Biases

The difference between 1.0 and the top of the raw output interval introduces a slight bias against the highest ordinal value, but this bias is not believed to be detectable in realistic attack scenarios. The exclusion of zero reduces the frequency of the lowest ordinal value by a tiny amount.

Listing 7: Microsoft SQL Server and PHP PRNG.

```
UInt32 State1, State2;

Double rand()
{
    // avoid 32−bit truncation, as that conflicts with modulus
    State1 = ((UInt64)State1 * 40014) % 2147483563;
    State2 = ((UInt64)State2 * 40692) % 2147483399;

    // confine to the interval [1, 2147483562]
    Int32 diff = (Int32)State1 − (Int32)State2;
    if (diff <= 0) diff += 2147483562;

    return (Double)diff * 4.656613e−10;
}
```

4.5.2 Reversing

If the PRNG's complete state is known, it can be simply reversed by applying the standard technique for reversing a multiplicative LCG to each of the two LCG's separately−that is, multiply the first LCG's state by its multiplicative inverse of 2082061899 (modulo 2147483563) and the second LCG's state by its multiplicative inverse of 1481316021 (modulo 2147483399).

4.5.3 Seeking

Likewise, the PRNG can be advanced or reversed by an arbitrary number of states using the exponential approach of Section 4.3.3 on each LCG separately.

4.5.4 Partial State Fixation

We construct a partial state fixation attack for this PRNG by first computing lower and upper bounds on the difference of the two constituent LCG's outputs. Continuing the example of a limit of 26 and a first output ordinal of 19, we obtain the following:

Listing 8: Partial State Fixation.

```
difflo = ((19 * 2147483589 + 25) / 26); // +25 to round up
diffhi = ((20 * 2147483590 + 25) / 26) − 1;
```

Notice that the 2147483589.46728 divisor is rounded down when computing the lower bound and rounded up when computing the upper bound. This is a very slight overcompensation to allow us to retain the convenience of integer arithmetic without a risk of missing a viable candidate seed.

With this range computed, we then brute-force the full state space of the first LCG, but for each possible value we only need to brute-force a range of size (diffhi - difflo + 1) of the second LCG's state space. The beginning of that range is offset so that the difference of each pair of candidate states falls within [difflo, diffhi], and therefore features a fixed top portion corresponding to the first ordinal of the output. This attack effectively divides the complete state space of the PRNG (approximately 2^{62}) by the limit, meaning a brute-force search is significantly accelerated but remains relatively computationally expensive.

4.5.5 Improved Partial State Fixation

Because this PRNG appears to only be consumed in take-from-top mode, the improved attack described in Section 4.1.6 can always be applied, further dividing the reduced search space by the limit.

Listing 9: Google V8 PRNG.

```
UInt32 State1, State2;

Double random()
{
    State1 = (18273 * (State1 & 0xFFFF)) + (State1 >> 16);
    State2 = (36969 * (State2 & 0xFFFF)) + (State2 >> 16);

    // discards the top 14 bits of State1 and of State2
    return ((State1 << 14) + (State2 & 0x3FFFF)) / 4294967296.0;
}
```

4.6 Google V8

The PRNG used in the Math.random() implementation of Google's V8 JavaScript engine (the function random_base in v8/trunk/src/v8.cc) is described as "using George Marsaglia's MWC [multiply-with-carry] algorithm". The code in Listing 9 depicts the algorithm. (For reference, we will refer to the integer derived from portions of the two internal states as the *composite output.*)

This algorithm has some unusual properties which we'll examine here and in the following sections. Perhaps the most useful observation is that the second MWC generator influences the output only rarely. The first MWC generator is used to populate the 18 most significant bits of the 32-bit result, while the second populates the 18 least significant bits, so the two overlap in the middle four bits. Because the two are combined using addition, it's possible for a bit to be carried out of the overlapping four bits (the first state's bits 3..0 plus the second state's bits 17..14), with a probability of approximately 47%. Each bit position the carry must propagate across in order to affect the output ordinal further attenuates the second MWC generator's influence by 50%. So for example, given a limit of 26, the odds of the second MWC generator affecting the output ordinal is 0.07%. We consider the application of this observation in the next section.

Although both states are 32 bits in size and are initialized with 32 bits of external data, both MWC generators use roughly a quarter to a half of the 32-bit number space. Specifically, each generator has a maximum recurring value of (multiplier $\cdot 2^{16}$ - 1). Initial values higher than the maximum will return to this range after at most two transformations and cannot leave it thereafter.

Both MWC generators contain four cycles: two cycles with a period of one, at zero and at (multiplier $\cdot 2^{16}$ - 1); and two cycles with period (multiplier $\cdot 2^{15}$ - 1), one with a lowest value of one and the other with a lowest value of five. These periods do not include the extracyclic values discussed previously. Note that the first MWC generator's higher state with a period of one, 0x4760FFFF, can be reached from two other extracyclic states which are its multiples, 0x8EC1FFFE and 0xD622FFFD. Because the second MWC generator's higher state with a period of one (0x9068FFFF) is greater than half of the 32-bit space, it has no such multiples.

4.6.1 State Reduction

As mentioned above, the second MWC generator can safely be ignored for more than 99% of all ordinals when brute-forcing the PRNG's internal state. If we substitute zero for the effective portion of the second generator's state, those rare discrepancies will always be one value greater (modulo the limit) than expected; if we substitute the midpoint value 0x20000, the discrepancies could be one value greater or lesser but will occur half as frequently.

In any case, ignoring the second MWC generator shrinks the raw state space we need to brute-force from 64 bits to 32 bits, or from approximately 61 bits to approximately 30 bits when excluding extracyclic values.

4.6.2 *Biases*

The most significant bits (positions 17 and 16) of the state portions used to derive the pseudorandom output are slightly biased. Because the highest recurring value is (multiplier $\cdot 2^{16}$ - 1) for both generators, and in both cases (multiplier % 4) = 1, bit positions 17 and 16 in these state portions both favor zero with a bias of (1 / multiplier).

4.6.3 *Reversing*

The MWC generator can always be reversed, but not necessarily to the correct previous state. Because the generators' states can be initialized to any 32-bit value, the first and even second states may be extracyclic before the generator enters a cycle. Most intracyclic states of either generator can be reached from one, two, or three extracyclic states in addition to its predecessor intracyclic state.

In cases where the goal is to recover the original seed values, reversing may at worst provide a few possible previous states (or fail, if no state could have been transformed into the current state). Practically speaking though, this ambiguity only matters when the generator is initialized to one extracyclic state and transitions to a second before entering a cycle, which means the very first pseudorandom number will be derived from an extracyclic state, and therefore we would need to test all predecessor states when reversing to determine which actually generated the observed output. Forward prediction, and reverse prediction up to the point of initialization, should not be impacted.

Our reversing algorithm for either of the MWC generators can be expressed as shown in Listing 10.

Listing 10: Reverse Algorithm for MWC.

```
if (State <= (Multiplier * 0x10000) − 1)
    return ((State % Multiplier) * 0x10000) + (State / Multiplier);
else if (State <= (Multiplier * 0xFFFF) + 0xFFFF)
    return ((State − (Multiplier * 0xFFFF)) * 0x10000) + 0xFFFF;
else fail();
```

The first case reverses one intracyclic state to its preceding intracyclic state, while the second reverses an extracyclic state to another extracyclic state that could have produced it. States not handled by either case cannot be produced from another state and therefore cannot be reversed.

4.6.4 *Partial State Fixation*

Like the partial state fixation attack of Section 4.5.4, our attack on the V8 PRNG fixes a portion of the composite output prepared from portions of the two MWC generators' states, which makes it slightly more convoluted than most of the other partial state fixation attacks. Because this composite output is used to produce ordinals in take-from-top fashion, we restrict its possible values to the interval capable of producing the first observed ordinal. This has the effect of dividing the search space for bits 17..0 of the first MWC generator's state by the limit, but we still need to brute-force bits 31..18 of the first state separately. We leave the second MWC generator's state at zero, for the reasons discussed in Section 4.6.1.

Revisiting our preferred example of a string of pseudorandom A-through-Z letters (say "TJJJJINTVJUQLTBM") using straightforward ordinal values, with a first ordinal of 19, we compute a range for the composite output and brute-force it as shown in Listing 11.

Listing 11: Brute-force of composite output.

```
complo = ((19 << 32 + 25) / 26) >> 14; // +25 to round up
comphi = ((20 << 32 + 25) / 26) >> 14 − 1;

for (s1lo = complo; s1lo <= comphi; s1lo++)
    // top 16 bits of intracyclic states are always < multiplier,
    // and s1hi is top 14 bits, so that's why (18273 / 4)
    for (s1hi = 0; s1hi < 18273 / 4; s1hi++)
    {
        s1 = (s1hi << 18) + s1lo;
        if (string_from_states(s1, 0) == "JJJJINTVJUQLTBM")
        {
            s1 = previous_state(s1, 18273);
            found_states(s1, 0);
        }
    }
```

4.6.5 *Improved Partial State Fixation*

With some adaptation, we can apply the improved partial state fixation attack to the V8 PRNG. Consider that incrementing the high 16 bits of the first MWC generator's state adds one to the succeeding state (the state used to derive the next output), while incrementing the low 16 bits adds the multiplier to the state. Because the effective portion of the first MWC generator's state covers bits 31..14 of the composite output, incrementing the high 16 bits of the state adds 0x4000 (2^{14}) to the composite output, while incrementing the low 16 bits adds 0x11D84000 ($18273 \cdot 2^{14}$) to the composite output. Notice that this means adding the multiplier to the high 16 bits also adds 0x11D84000 ($18273 \cdot 2^{14}$) to the composite output. Therefore, if we imagine the set of all candidate states in $[0, \mathit{multiplier} \cdot 2^{16})$ sorted in ascending order first by the low 16 bits, then by the high 16 bits, we can identify blocks of consecutive states that will all result in the same second ordinal. The size of each block is (0x100000000 / 0x4000 / limit), which is essentially the (M / A / limit) expression of Section 4.1.6.

There is one slight complication in skipping these blocks: our algorithm increments the top 18 bits of the state, not the top 16 bits, and this has four times the effect—i.e., it adds 0x10000 (2^{16}) to the composite output instead of 0x4000 (2^{14}). For an example limit of 26, then, instead of encountering blocks of size (0x100000000 / 0x4000 / 26) = 10082 or 10083, we will observe four times as many blocks, each of size 2520 or 2521. If the difference (modulo the limit) of the expected second ordinal minus the next ordinal to be produced from a candidate seed is not 0 or 1, we skip the next (difference - 1) blocks of candidate states. We include the -1 term to skip very conservatively, not wanting to skip into the middle of a block of viable candidate states. After completing a block of candidate states that will produce the expected ordinal, we can then skip the next (limit - 1) blocks.

4.7 Microsoft .NET

The Microsoft .NET Framework's PRNG, implemented in System.Random (see Random.cs in the SSCLI[10]), is copied almost verbatim from the ran3 implementation of *Numerical Recipes in C, 2nd Edition*[13], except for its choice of modulus (the prime 2147483647 versus ran3's 1000000000) and its offset between indices (31 versus 21), and a couple details of its seeding implementation. It is described as a "subtractive" PRNG, at the heart of which is a 55-integer array accessed cyclically using a pair of indices with a fixed separation of 21 modulo 55 (similar to the "lag" described by Kahaner et al.[4]). When a pseudorandom number is requested, the indices are incremented, the element at the second index is subtracted (modulo 2147483647) from the element at the first index, and the difference is both stored at the first index and returned.

The .NET Framework provides both System.Random.Next() and System.Random.Next(Int32) methods, so it is unclear whether application developers will prefer to use the former in a take-from-bottom or take-from-top approach, or use the latter which is inherently take-from-top in its implementation. We consider both modes in the following sections.

Admittedly, we have no highly effective attacks to present for this PRNG. Since the PRNG's internal state is so large (approximately 1700 bits) compared to the size of its seed (31 bits), even brute-forcing the entire seed space will not reach all possible states. Outputs produced after some use of the PRNG following seeding would not be matched, because both seeding and stepping are necessary to arrive at the state that produced that output. Brute-forcing the 1700-bit state space is clearly impossible.

4.7.1 Biases

The .NET Framework PRNG, like most we reviewed, exhibits very small biases which should not be realistically detectable. Because the PRNG's modulus is prime, there is no limit less than the modulus such that all ordinal values occur with equal probability.

4.7.2 Forward Prediction

A very important property of the .NET Framework PRNG is that it can be expressed as the following recurrence relation:

$$x_{i+55} = x_i - x_{i+21}$$

To avoid confusion, x refers to the ordinal at the indicated position in the pseudorandom stream, and not to an element of the internal array.

For each pair x_i and x_{i+21} we obtain, we can predict the future output x_{i+55}. If we obtain 55 consecutive outputs, we should be able to predict all future output. This is equally true in the take-from-top and take-from-bottom modes.

Unfortunately, it's not completely true. At least half of all ordinals predicted in this way will differ from the actual ordinal by a fixed constant, and this effect compounds after the 34th prediction, when inaccurate predictions become the bases for further predictions.

In a take-from-top application, the actual value might be less than the predicted value by one, due to the effects of the hidden "fractional" portion of each element of the internal array—that is, the mantissa of multiplying the element by the limit and dividing by the modulus. If an element with a greater fractional part is subtracted from an element with a lesser fractional part, the actual output value will be lower than expected.

In a take-from-bottom application, the actual value might be greater than the predicted value by (modulus % limit), which signifies that an integer underflow occurred during the subtraction and caused the difference to wrap around the integer ring defined by the modulus. Because the modulus is prime, such errors occur for all choices of limit.

(Note that when we refer to one integer in a ring being "greater" or "less" than another by some amount, we mean that the first integer equals the second integer plus or minus the amount, not that the integer is actually greater or lesser in any absolute, number-line sense.)

4.7.3 Reverse Prediction

The recurrence relation describing the .NET Framework PRNG can be equivalently expressed as follows:

$$x_{i-55} = x_{i-34} + x_i$$

This permits us to predict past output ordinals by summing pairs of known values. With 55 consecutive values, we should be able to predict all past output as well, but the same complications arising in forward prediction apply here in reverse. Each actual value of take-from-top-generated output might be greater than the predicted value by one, due to carry from the sum of the two known elements' hidden fractional parts. In a take-from-bottom mode, the actual value might be less than the predicted value by (modulus % limit) if an integer underflow occurred when the actual value was generated. In both cases, the inaccuracies start to compound after 21 predictions.

4.7.4 Reversing

Given the full state of the internal array, the .NET Framework PRNG is easily reversible with complete accuracy by adding the element at the second index to the element at the first index and then decrementing the indices.

4.7.5 Seeking

Although it might not be immediately obvious, it is possible to efficiently seek the PRNG by an arbitrary number of states in either direction. The key is to construct a 55 x 55 matrix that represents the effects of 55 consecutive pseudorandom number generations. Conceptually, the internal state array comprises 55 variables, and each row of the matrix contains the coefficients for a 55-variable equation to compute the value that the corresponding element will take after the transformation represented by the matrix occurs.

To seek by 'n' states, raise this matrix—or its inverse when seeking backward—to the (n / 55)th power, and multiply the matrix by the state vector (the current contents of the internal array) to perform a "coarse seek" by a multiple of 55 states (Note that the matrix arithmetic must also be modular). For the "fine seek," step forward or backward up to 54 times to reach the desired state.

Because matrix exponentiation is expensive, one suggested optimization involves reducing the number of matrix multiplications at the cost of some amount of overshooting or undershooting.

4.7.6 Seed Computation

With the complete state of the internal array and the approximate position of that state (i.e., how many pseudorandom numbers have been generated since the PRNG was seeded), it is possible to compute the seed value that was used to initialize the array. First, compute the array for two reference seeds, 161803398 (which becomes zero due to a subtraction from the constant MSEED in the initialization code) and 161803397 (which becomes one) and seek both to the given position. Next, subtract the first array from the second to compute what we'll call the delta vector. The elements of this vector represent the contribution−specific to this position−made by increasing the seed from the first seed to the second.

Now subtract the first array from the array of interest to make the latter relative to the former. We'll now essentially divide each element by the contributions found in the delta vector. Compute the multiplicative inverse of each element of the delta vector (for instance, using the extended Euclidean algorithm), and multiply each element in the now-relative array of interest by the computed inverse. If the approximate position is sufficiently close, most or all of the products will equal the difference of the seed of interest minus the reference seed. Finally, subtract the first reference seed from the resulting seed, and subtract that difference from 161803398 to arrive at the original seed.

In summary, we compute the following:

Listing 12: Seed Computation.

```
DeltaVector = ReferenceVector1 − ReferenceVector0

SeedVector[i] = 161803398 − (StateVector[i] − ReferenceVector0[i]) ·
        DeltaVector[i]^−1
```

If most or all elements of SeedVector agree, the result is the original seed. (Note that all arithmetic is modulo 2147483647.)

4.7.7 Minor Partial State Recovery

In Sections 4.7.2 and 4.7.3, we mentioned the difficulty of accurately predicting past and future output because of arithmetic underflow or borrow causing actual output to diverge from our predictions. However, if we can collect hundreds of consecutive ordinals of output, we can compare each actual value to what we would have expected based on the recurrence relation, and the errors might reveal some small amount of information about the PRNG's internal state at the time the output was generated.

We identify chains in the sequence of ordinals where errors do not occur and, based on the length of the chain, infer a distribution for the hidden portion of the corresponding state element. For example, given $x_i, x_{i+(k\cdot 55)+21}$, and $x_{i+(k\cdot 55)+55}$ for some choice of i, x_i is the head of the chain, each $x_{i+(k\cdot 55)+21}$ is a subtrahend, and each $x_{i+(k\cdot 55)+55}$ is a difference. For each difference where no error is observed, it must be the case that no underflow or borrow occurred when subtracting the corresponding internal state elements. If k successive differences fail to exhibit an error, then the head of the chain can be said to have "survived" k subtractions, and therefore it is more likely that the corresponding state (take-from-bottom) or its fractional part (take-from-top) was large. This information can then be used to refine the distributions of values likely to be taken by the "associated" states at (i-55), (i-21), (i+21), and (i+55).

As of this writing, we have only tested very basic proofs-of-concept of this attack; we have not yet prepared a more formal treatment.

4.8 glibc (type 3)

The GNU C Library default rand() implementation, type 3, is an array-based PRNG with some similarities to the .NET Framework PRNG, but also some crucial differences. It comprises a 31-element array and two indices, with the front index leading the rear by three elements. Each time a pseudorandom number is generated, the element at the rear index is added to the element at the front index, and the sum is both stored at the front index and returned, although the value to be returned−but not the value to be stored−is first divided by a discard divisor of two. The indices are incremented only after the sum is computed and stored.

The discard divisor effectively gives each array element a hidden "fractional" bit which causes significant unpredictability; this bit's contribution is particularly important because the PRNG operates with a modulus of 2^{32}.

The seed computation technique of Section 4.7.6 does not apply to the glibc PRNG because it applies a modulus of 2147483647 during the first phase of seeding and a modulus of 2^{32} otherwise.

4.8.1 Forward Prediction

The glibc PRNG can be expressed as the following recurrence relation:

$$x_{i+31} = x_i + x_{i+28}$$

Where x represents the ordinal at the stated position in the pseudorandom stream, plus a hidden fractional bit which is not reflected in the output.

For each pair x_i and x_{i+28} we obtain, we can predict the future output x_{i+31}, both in take-from-top and in take-from-bottom modes; with 31 consecutive outputs, we can predict all future output. However, all such predictions are subject to errors similar to those described in Section 4.7.2, arising in take-from-bottom mode from arithmetic overflows when the limit is not a power of two, and in both modes due to carry from summing hidden fractional portions. These errors begin compounding after the third prediction.

Carry-based errors manifest as the actual ordinal being greater than the predicted ordinal by one, while arithmetic overflow-based errors appear as the actual ordinal being less than predicted by ((modulus / 2) % limit). Each ordinal generated through a take-from-bottom approach can exhibit neither, either, or both errors. Because (modulus / 2) - 1 =

2147483647 is prime, no practical choice of limit can result in ((modulus / 2) % limit) = 1, meaning the two types of error cannot interfere destructively. The only choices of limit that yield ((modulus / 2) % limit) = (limit - 1) are 3 and 715827883, and therefore the two error types should practically always be distinguishable.

4.8.2 *Reverse Prediction*

The recurrence relation of the preceding section can equivalently be rendered as:

$$x_{i-28} = x_{i+3} - x_i$$

Using this relation, we can predict past ordinals as the differences of pairs of known ordinals, subject to the same errors as described in the previous section, but inverted. These errors start to compound after the 28th prediction.

4.8.3 *Reversing*

Given the complete state of its internal array and indices, the glibc PRNG can be easily reversed by decrementing the indices and then subtracting the element at the rear index from the element at the front index.

4.8.4 *Seeking*

The glibc PRNG can be advanced or reversed by an arbitrary number of states using the same technique described in Section 4.7.5, except with a 31 x 31 matrix and coarse seeking in multiples of 31 states.

4.8.5 *Partial State Recovery*

When the glibc PRNG is used in a take-from-bottom approach, and we have access to scores or hundreds of consecutive ordinals of pseudorandom output, we can deduce the fractional bits' states by detecting the errors they induce. Once we know 31 consecutive fractional bits, we can predict with complete certainty all future values they will take.

For each output ordinal triple x_i, x_{i+28}, and x_{i+31}, if x_{i+31} is greater than expected by one after accounting for arithmetic overflow-based error, then the triple exhibits a carry-based error. If we detect a carry-based error, we know that x_i and x_{i+28} both have fractional bits of one, and therefore x_{i+31} has a fractional bit of zero; otherwise, we don't learn anything.

Once we know that an x_i has a nonzero fractional bit, we can assign the fractional bits of x_{i-28} and x_{i+3} based on the presence or absence of a carry-based error at x_{i+3} (1+1=**10** if an error was detected, or 0+1=**0**1 if not), and likewise for x_{i+28} and x_{i+31} according to the carry-based error at x_{i+31} (1+1=**10** or 1+0=**0**1). Once any pair of x_i, x_{i+28}, and x_{i+31} is known, the unknown fractional bit can be computed as an XOR of the known fractional bits. By repeatedly applying this algorithm to pseudorandom output, we can eventually determine 31 consecutive fractional bit values, enabling forward and reverse prediction with certainty of fractional bits and the N least-significant bits of ordinals such that 2^N is a factor of the limit.

4.8.6 *Position Recovery*

As mentioned in the glibc PRNG source code, the internal state's 31 fractional bits form a linear feedback shift register (LFSR) which exhibits the maximal period of 2147483647 (2^{31} - 1) assuming that at least one bit is nonzero. With complete knowledge of two LFSR states−for instance, the LFSR state immediately after the PRNG is initialized with a particular seed, and the LFSR state recovered some time later using the algorithm of Section 4.8.5−we can compute the distance modulo 2147483647 between the two corresponding points in the pseudorandom stream. This effectively allows us to quickly compute the position of pseudorandom output relative to a known initial state, provided that the output is long enough to let us recover 31 consecutive fractional bits.

Our implementation for computing position simply advances an LFSR state until it matches one of a sorted list of "waypoint" states, and then it subtracts the number of iterations from the waypoint's position retrieved from a parallel list. The distance between two LFSR states is the difference modulo 2147483647 of their positions identified in this way.

4.8.7 *Seed and Position Recovery*

We now describe an attack that allows us to serially brute-force the position in a pseudorandom stream at which a sample of output occurs and simultaneously learn the seed from which that pseudorandom stream arises. At each candidate position, we must test at most 16 seeds; for each seed, we initialize the glibc PRNG, seek to the candidate position, and compare the PRNG's output to the sample. When the output matches the sample, the tested seed and position can then be used to immediately reconstruct the full internal state of the application's PRNG, which enables forward and reverse prediction with complete accuracy.

However, this attack has two significant drawbacks. One serious limitation is that the attack requires knowledge of 31 consecutive fractional bits, which we term an *LFSR state*. The partial state recovery attack of Section 4.8.5 can be used to reveal this information, although it tends to require on the order of a hundred consecutive output ordinals, a high but not impossible number. We suspect that attacks capable of operating on discontinuous samples are also possible.

A second drawback is paying for the time-memory tradeoff: we have to precompute a huge amount of data in order to efficiently translate any given LFSR state back into the seed values that could produce it. In our present implementation, this data totals 24 gigabytes, a 16 GB table organizing all of the seeds that produce each of the 2147483648 possible LFSR states (there can be from zero to 16 seeds per LFSR state), and an 8 GB table translating LFSR states to indices into the first table. To conserve space, we let the range in which an index falls denote the number of seeds stored at that index. For example, if an LFSR state maps to an index in the interval [0x22A5B043, 0x6826A3EA], then there are two seeds that produce it, while an index in the interval [0x6826A3EB, 0xADA75392] indicates that three seeds produce the associated LFSR state.

The 24 GB of tables can be precomputed once and used forever after, but their size and the character of their contents make even operating upon them resource-intensive. Specifically, the attack involves repeatedly accessing small pieces of data at unpredictable offsets within the tables, which hobbles the efficacy of caching portions of the table as they are read from disk and causes chaotic disk seeking. For

Listing 13: Seed and Position Recovery attack.

```
lfsr = recover_fractional_bits(output, length);

for (pos = 0; pos < cutoff; pos++)
{
    index = LfsrToIndex[lfsr];
    seedcount = index_to_seed_count(index);

    for (i = 0; i < seedcount; i++)
    {
        seed = IndexedSeeds[index + i];

        glibc3_srand(seed);
        glibc3_seek(pos);

         if (glibc3_rand_output(length) == output)
            found_seed_and_position(seed, pos);
    }

    // compute previous fractional bits
    // reverse: o[0] = o[31] − o[28]
    lfsr <<= 1;
    lfsr |= ((lfsr >> 31) ^ (lfsr >> 28)) & 1;
    lfsr &= 0x7FFFFFFF;
}
```

performance, we recommend that the tables be consumed on a 64-bit system with enough RAM to keep substantially all of both tables in memory—a non-negligible requirement by today's standards—and that they be read from beginning to end into memory in preparation for the attack.

Assuming that the tables are fully resident in memory, our attack can be expressed as shown in Listing 13.

As suggested by the placeholder "cutoff" above, it is not possible to exhaustively test all possible positions, only some arbitrary reasonable number of them, up to the maximal period of the PRNG. For reference, when the PRNG is seeded with a value of one, we have observed that its period has an upper bound of 0x143E438A0426502B921B3DC0C576769A5 A9A7DE2F5B5995A9439E51775C6E66411F6CDCC903B19 D53897F94B0AC8A684C5C134A1FF2D03E4D7519D49B09 6DA864B16FDE9634A18406373C1801BB754331731C35DE 04C22CC791476F64E24C62CAEF80000000, or approximately $2.16 \cdot 10^{243}$.

4.9 Perl

Perl's no-argument rand() implementation simply invokes the underlying platform's rand() and divides it by the platform's output modulus (RAND_MAX + 1) to produce a normalized pseudorandom number in the interval [0.0, 1.0). If an argument is supplied to rand(), the normalized pseudorandom number is multiplied by that argument. In either case, the Perl PRNG amounts to a take-from-top approach to generating pseudorandom output, using a platform-provided PRNG which is otherwise, presumably, typically operated in a take-from-bottom capacity.

We did not examine any of the other PRNGs offered by Perl.

5. CONCLUSIONS

We have presented analyses of a number of mainstream PRNGs and demonstrated a variety of what we believe to be novel attacks to facilitate recovery of a PRNG's state and/or seed based on an application's pseudorandom output. We also offer various techniques—some old, some new—for efficiently transforming a PRNG's internal state. Updates to this work will be featured on www.cylance.com, and it is our hope that this work will inspire improvements and contributions from other researchers. Most of all, we hope that the insecurity of non-cryptographic PRNGs will become common knowledge for all developers.

6. ACKNOWLEDGMENTS

The authors would like to thank Shelly Siebert for her many improvements to the article, Matt Wolff for providing valuable feedback, and Glenn Chisholm for suggesting and prototyping GPU-based parallelization.

7. REFERENCES

[1] G. Argyros and A. Kiayias. PRNG: Pwning Random Number Generators. `https://media.blackhat.com/bh-us-12/Briefings/Argyros/BH_US_12_Argyros_PRNG_WP.pdf`, Black Hat USA 2012.

[2] C. Meyer et al. Randomly Failed! The State of Randomness in Current Java Implementations. `http://www.rsaconference.com/writable/presentations/file_upload/cryp-w25.pdf`, RSA Conference USA 2013.

[3] J. D. Cook. *Beautiful Testing.* O'Reilly, 2010. Chapter 10: How to Test a Random Number Generator.

[4] D. Kahaner et al. *Numerical Methods and Software.* Prentice Hall, 1989. Chapter 10: Simulation and Random Numbers.

[5] I. Goldberg and D. Wagner. Randomness and the Netscape Browser. *Dr. Dobb's Journal*, January 1996.

[6] S. Kamkar. phpwn. `http://samy.pl/phpwn/`, 2009.

[7] D. E. Knuth. *The Art of Computer Programming*, volume Volume 2: Seminumerical Algorithms. Addison-Wesley, 1981.

[8] L. Dorrendorf et al. Cryptanalysis of the Random Number Generator of the Windows Operating System. `http://eprint.iacr.org/2007/419.pdf`, November 2007.

[9] P. L'Ecuyer. Random numbers for simulations. *Communications of the ACM*, 33(10), October 1990. `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/cacm90.pdf`.

[10] Microsoft Corporation. Shared Source Common Language Infrastructure 2.0 Release. `http://www.microsoft.com/en-us/download/details.aspx?id=4917`, 2006.

[11] MITRE Corporation. CWE-330: Use of insufficiently random values. `http://cwe.mitre.org/data/definitions/330.html`, 2008.

[12] Software in the Public Interest, Inc. SSLkeys. `http://wiki.debian.org/SSLkeys`, 2011.

[13] W. Press et al. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 2nd edition, October 1992. pp. 283.

[14] Wikipedia. RANDU. `https://en.wikipedia.org/wiki/RANDU`, 2013.

[15] M. Zalewski. Strange Attractors and TCP/IP Sequence Number Analysis. `http://lcamtuf.coredump.cx/oldtcp/`, 2001.

[16] M. Zalewski. Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later. `http://lcamtuf.coredump.cx/newtcp/`, 2002.