ISAAC

Robert J. Jenkins Jr.

Abstract

A sequence of new pseudorandom number generators are developed: IA, IBAA, and ISAAC. No efficient method is known for deducing their internal states. ISAAC requires an amortized 18.75 instructions to produce a 32-bit value. There are no cycles in ISAAC shorter than 2^{40} values. The expected cycle length is 2^{8295} values. Tests show that scaled-down versions of IBAA are unbiased for their entire cycle length. No proofs of security are given.

1 Introduction

The purpose of this paper is to introduce the new random number generators IA, IBAA, and ISAAC. IA (Indirection, Addition) is slightly biased but it appears to be secure. It is immune to Gaussian elimination. IBAA (Indirection, Barrelshift, Accumulate and Add) eliminates the bias in IA without damaging security. ISAAC (Indirection, Shift, Accumulate, Add, and Count) is faster than IBAA, guarantees no bad seeds or short cycles, and makes orderly states disorderly faster.

IA was designed to satisfy these goals:

- Deducing the internal state from the results should be intractable.
- The code should be easy to memorize.
- It should be as fast as possible.

More requirements were added for IBAA:

- It should by cryptographically secure [1] [12].
- No biases should be detectable for the entire cycle length.
- Short cycles should be astronomically rare.

A generator was found that had the appropriate levels of bias. It used an accumulator and barrelshifts. IBAA was formed by combining it with IA without introducing bias or reducing the security of IA. (Any unbreakable unbiased generator which has long cycles must be cryptographically secure.)

ISAAC took away the requirement of easy memorization but added more:

- The C code should be optimized for speed.
- Orderly states should become disorderly quickly.

Figure 1: C code for IA

```
typedef  unsigned int  u4;   /* unsigned four bytes, 32 bits */
#define ALPHA      (8)        /* log of number of terms in m */
#define SIZE       (1<<ALPHA)      /* 1 time 2 to the 8, 256 */
#define ind(x)     (x&(SIZE-1))     /* low order 8 bits of x */

static void ia(m,r,bb)
u4 *m;   /* Memory: array of SIZE ALPHA-bit terms */
u4 *r;   /* Results: the sequence, same size as m */
u4 *bb;  /* the previous result */
{
  register u4 b,x,y,i;

  b = *bb;
  for (i=0; i<SIZE; ++i)
  {
    x = m[i];
    m[i] = y = m[ind(x)] + b;              /* set m */
    r[i] = b = m[ind(y>>ALPHA)] + x;       /* set r */
  }
  *bb = b;
}
```

- There should be no short cycles at all.

ISAAC is similar in form and function to the alleged RC4 [11], although the generators were developed independently. ISAAC is three times faster, less biased, and has longer minimum and average cycle lengths. ISAAC requires an amortized 18.75 machine instructions to produce a 32-bit value. ISAAC should be useful as a stream cipher, for simulations, and as a general purpose pseudorandom number generator.

The sections of this paper describe IA, IBAA, test results for IBAA, and ISAAC.

2 IA

The new generator IA was designed to be secure, fast and easy to memorize. C code for IA is given in figure 1.

IA operates on a secret array `m` of 256 values. The values in `m` should contain at least 2`ALPHA` bits. IA uses pseudorandom indirection to determine its results. The results given by IA are the sum of values in `m`, not actual values in it. IA does

not swap values in `m`, instead it walks through the array adding pseudorandomly chosen terms to the old terms.

IA is reversible: every internal state has exactly one predecessor. The average cycle length of all elements in all reversible mappings of s states is about $s/2$, while the average cycle length of all elements in all irreversible mappings is about $\sqrt{s}$ [5] [9]. In addition to having every internal state on some cycle, reversible generators tend to have over half the states on the same cycle, giving the sequences a very uniform distribution.

Notice that when `x` is added into `r[i]=b`, `x` is no longer in `m`. Therefore `x` came from a different pool of values than the pseudorandom term that is added in with it. If this were not the case, IA would not be reversible and the results would be biased in favor of even values.

The two indirections bracket the user's result. `r[i]` is the old value of `m[i]`, but with a pseudorandomly chosen value added. The new value of `m[i]` is the user's previous result, but with a different pseudorandomly chosen value added. There is no equation which does not contain a new pseudorandomly chosen value. If the pseudorandom values are treated as unknowns, this is enough to thwart Gaussian elimination. Guessing what the choice was means guessing 8 bits of information per value.

There are windows into the internal state of IA. The relationship `ind(m[i])` = `ind(r[i]-i)` is 1/256 too probable, as is `ind(m[i-SIZE]>>8)` = `ind((r[i]>>8)-i)`. They happen when a pseudorandom indirection chooses itself. Each relationship holds 1/128 of the time.

It is possible to avoid these windows by limiting each pseudorandom choice to the half of the array which does not include the value used for the pseudorandom choice (`x` or `y`). This would leave only 128 values for each pseudorandom choice, giving 256 relationships that are correct 1/128 of the time (as opposed to the two relationships we have now). The proposed modification also makes the code slower, more complicated, and more biased, so it was not done.

Biases can be detected in IA using the correlated gap test. These biases are similar in nature to those seen in lagged-Fibonacci and add-with-carry generators [7]. The biases are smaller than the previously noted windows into the internal state.

No efficient attack is known against IA. The guess-and-generate attack, which applies the equations of IA to an arbitrary initial guess but sets `b` to the real results of IA, converges to the true state of IA after about 2^{17} values when `ALPHA` = 3. The attack cannot be extended to `ALPHA` = 4, let alone `ALPHA` = 8. Attacks on the alleged RC4 [11] usually can be applied to IA, and vice versa.

Proving the security of IA would require showing that no algorithm could efficiently deduce its internal state. No algorithm examined so far can deduce its internal state, and Gaussian elimination is one of the algorithms that has been examined. This is not a proof by any means, but it is a start.

3 IBAA

IA was extended to IBAA. In addition to being fast, easy to memorize, and immune to Gaussian elimination, IBAA was required to have no detectable bias for the entire cycle length. Short cycles must be very rare. C code for IBAA is given in figure 2. The next section gives the results of statistical tests on IBAA; it does seem to be unbiased.

The lack of bias in IBAA comes from the accumulator `a`. The term added to `a` is `m[ind(i+(SIZE/2))]`. Why were `x` or `y` not used instead, seeing how they are already in registers? This decision was made based on a single series of tests. (See the testing section for more detailed descriptions of the terms and methods here.) The tests were on IBAA, except `m[ind(x)]` and `m[ind(y>>ALPHA)]` were replaced with `0` and `0`. The generator was scaled down to have 8 terms (not 256) of 3 bits apiece (not 32). With a total of 30 bits of state, it had a maximum cycle length of 2^{30} calls. `ind(i+(SIZE/2))` was replaced with `ind(i+j)`, for each $j \in 0 \ldots 7$. Each of these eight generators produced a sequence of 2^{27} calls, or 2^{30} values. No cycles were detected. The low-order bit was removed from each value, leaving sequences of 2-bit values. The gap test was applied to each of these sequences, tracking gaps of length $0 \ldots 63$. The expected χ^2 result was 63, but the actual results (ordered by j) were 684, 412, 208, 201, 212, 203, 682, and 13584. The difference from 63 is proportional to the amount of bias detected. In all cases the first bad gap was of length 10. No other tests detected significant amounts of bias, so the decision had to be based on this alone. It appears that the bias decreases with the distance from either endpoint, so `m[ind(i+(SIZE/2))]` was chosen.

`barrel(a)` is a permutation of `a`, and is nonlinear when combined with addition. Permutations help assure that all values are equally likely. Nonlinear systems are less prone than linear systems to mixing values then spontaneously unmixing them after they have been churned for awhile. The security of IBAA, however, does not depend upon this nonlinearity. The security depends upon the indirections `m[ind(x)]` and `m[ind(y>>ALPHA)]`.

If `m[i]`, `m[ind(x)]` and `m[ind(y>>ALPHA)]` are treated as separate unknowns, then every set of equations has at least 4/3 as many unknowns as equations. Let a set of $3n$ equations (n setting `a`, n setting `m`, and n setting `r`) be given. It will produce at least $4n$ unknowns: n each of `a`, `m[i]`, `m[ind(x)]`, and `m[ind(y>>ALPHA)]`. Eliminating any subset of these equations only increases the ratio of unknowns to equations.

If an arbitrary reversible mapping has N possible values, then the chance of an arbitrary starting point being on a cycle of length N/x or less is 1/x. The number of internal states of IBAA is 2^{8264}, so the chances of arbitrarily choosing a cycle shorter than 2^{40} are about 2^{-8224}. About 2^{140} protons could fit in the known universe [10]. The state of all zeros forms a cycle of length 256 though; after `i` passes through $0 \ldots 255$ the state maps back to all zeros.

Figure 2: C code for IBAA

```
/*
 * ^ means XOR, & means bitwise AND, a<<b means shift a by b.
 * barrel(a) shifts a 19 bits to the left, and bits wrap around
 * ind(x) is (x AND 255), or (x mod 256)
 */
typedef  unsigned int  u4;   /* unsigned four bytes, 32 bits */
#define ALPHA      (8)
#define SIZE       (1<<ALPHA)
#define ind(x)     ((x)&(SIZE-1))
#define barrel(a)  (((a)<<19)^((a)>>13)) /* beta=32,shift=19 */

static void ibaa(m,r,aa,bb)
u4 *m;   /* Memory: array of SIZE ALPHA-bit terms */
u4 *r;   /* Results: the sequence, same size as m */
u4 *aa;  /* Accumulator: a single value */
u4 *bb;  /* the previous result */
{
  register u4 a,b,x,y,i;

  a = *aa; b = *bb;
  for (i=0; i<SIZE; ++i)
  {
    x = m[i];
    a = barrel(a) + m[ind(i+(SIZE/2))];    /* set a */
    m[i] = y = m[ind(x)] + a + b;          /* set m */
    r[i] = b = m[ind(y>>ALPHA)] + x;       /* set r */
  }
  *bb = b; *aa = a;
}
```

4 Tests

Tests run against random number generators with 256-term internal states often will not fail no matter how long they are run. The cycle lengths of such random number generators are more than astronomical. In order for statistical tests to be of use, generators need to be **scaled down**. The number of terms in the array and the size of the terms must be reduced. This has a number of advantages.

- Flaws are magnified because boundary cases occupy a larger percentage of the total number of states.
- The tests run faster because the arrays are shorter.
- If the internal state is small enough, all internal states can be enumerated and cycle lengths can be reached. There is clearly no point in running tests longer than the cycle length.

The tests run were Knuth's frequency, gap, and run tests [3]. The frequency test counts how many times each value appears. The gap test measures the gaps between occurances of values in the results. For example, the sequence "abcdeaf" has a gap of 4 between occurances of "a". The gap test measured gaps up to four times the length of the internal array. The run test counts the lengths of strictly increasing subsequences. The expected distribution of values for a truly random sequence is known for each of these tests, and was compared against the sample distributions using the standard χ^2 formula [3].

Two types of values were used, "normal" and "correlated". Random number generators are designed to produce lots of random values. These are the "normal" values. "Correlated" values were derived from groups of normal values. There is one correlated value per call to the generator; it has as many bits as the normal values but is composed of the low-order bit of the first few normal values. Correlated values could identify patterns that occurred between calls.

The initial seed in all cases was `m[i]=i, a=1, b=1`. Each generator was warmed up by making ten calls before statistics were gathered. ALPHA (`a`) is the log of the length of `m`, BETA (`b`) is the number of bits in each value, and SHIFT (`s`) is the amount of the barrelshift (relevant only to IBAA). The normal values are either the whole values in `r` or the low-order ALPHA bits of each `r` value.

In the scaled-down versions of IBAA, SHIFT was chosen to be the integer closest to the golden ratio (.618) times BETA [4]. These shift values seem to work well. No reason is known for why they should work well. The scaled-down versions still are not quite IBAA, because the values usually had fewer than `2ALPHA` bits. Many bits of `ind(y>>ALPHA)` were always zero, so the pseudorandom choices were very restricted.

Figure 3: Test results for IBAA

IBAA	correlated frequency	normal frequency	correlated gap	normal gap	normal run	number of calls
a1b1s1	1:0	1:0	7:4	7:13	1:2	5
a1b2s1	3:2	3:1	7:4	7:3	3:0	12
a1b3s2	3:0	3:0	7:5	7:11	3:7	3164
a1b4s2	3:3	3:6	7:6	7:3	3:0	10441
a1b5s3	3:0	3:0	7:14	7:5	3:6	235491
a1b6s4	3:5	3:7	7:5	7:2	3:3	1869951
a1b7s4	3:0	3:0	7:1	7:7	3:0	221862935
a2b2s1	3:0	3:1	15:7	15:11	3:1	1407
a2b3s2	7:6	7:7	15:21	15:8	7:3	29382
a2b4s2	15:9	15:11	15:16	15:7	7:9	6146999
a2b5s3	15:12	15:12	15:15	15:12	7:7	9507107
a3b3s2	7:3	7:0	31:44	31:49	7:5	886828921
a8b32s19	255:238	255:215	1023:949	1023:1016	7:5	2^26

A result 15:9 means expected 15, actually got 9. A test is said to pass if the actual result differs from the expected result by less than four times the square root of the expected result.

The normal gap test was questionable for IBAA a3b3s2.

The number of calls was the cycle length, except for a3b32s19. The cycle length for IBAA a2b5s3 was unusually short.

Test results are given in figure 3. If a test would have taken more than a day to run and tests on smaller generators had failed to detect any bias, then the test was not run.

A common requirement of cryptographically secure random number generators is that all detectable biases b decrease exponentially with some polynomial function f of the size s of the internal state: $b < 2^{-f(s)}$ [1] [12]. No significant bias was detected in IBAA, so it might satisfy this requirement or it might not.

Tests suggest that all consecutive 256-value strings are equally likely results from IBAA, 256 being the number of terms in `r`. No tests on samples of that size or smaller ever failed, even for IA which has known biases. The gap and run tests in particular only fail if they look at subsequences of more than $2^{\texttt{ALPHA}}$ values [3]. All 8192-bit strings are equally likely in `m`; there are 2^{64} such states

for every string (one for each possible value of `a` and `b`).

George Marsaglia's DIEHARD test suite [8] was found shortly before this paper went to print. Two samples each from full-scale IBAA and ISAAC were tested. Although each sample had some test return questionable results, no test had questionable results for both samples for either generator. Separate experiments have seen IBAA develop small biases that fade away as sequences grow longer. Bias peaked in subsequences with about 2^{21} values. ISAAC does not seem to have this problem with short term bias.

5 ISAAC

IBAA was extended to be leaner, meaner, and have no short cycles at all – at the expense of being easy to memorize. The result is ISAAC, shown in figure 4. If the initial internal state is all zero, after ten calls the values of aa, bb, and cc in hexadecimal will be `d4d3f473`, `902c0691`, and `0000000a`.

rngstep() The macro `rngstep()` is essentially the inner loop of IBAA. Repeating it four times (unrolling the loop) reduced the loop overhead. This does not affect the results.

***m++** Replacing `m[i]` with `*m++`, `r[i]` with `*r++`, and `m[i(SIZE/2)]` with `*m2++` reduced the cost of looking up terms in predictable array positions. `m` is a pointer, `*` gets the term it points at, and `++` moves the pointer up one to the next term. This does not affect the results.

a$\hat{}$(mix) The barrelshifts of IBAA were replaced with a sequence of four functions: `a^(a<<13)`, `a^(a>>6)`, `a^(a<<2)`, and `a^(a>>16)`. `^` means XOR and `<<` and `>>` are shifts. Each call to `rngstep()` does one of these functions. When machines have no barrelshift instruction, this saves one instruction per `rngstep()`. This sequence of functions also cause `a` to achieve avalanche [6] in twelve `rngstep()`s. That causes orderly states to become disorderly faster, reducing short term biases. It should be noted that each of these functions is a permutation of `a`.

cc A counter was included which is used (and incremented) only once per call. This was suggested by Bill Chambers [2]. `cc` and `i` together guarantee a minimum cycle length of 2^{40} values. No cycles are known which are that short. No bad initial states exist, not even the state of all zeros. Tests have shown that adding independent things to `b` does not greatly affect the generator's bias or security.

ind(x) The indirection bits used in ISAAC are $2 \ldots 9$ for `x` and $10 \ldots 17$ for `y`. (IBAA used $0 \ldots 7$ and $8 \ldots 15$.) This shaved another instruction off each indirect lookup. Scaled-down tests suggest that the choice of indirection bits does not affect security or bias, providing no bit is used twice.

Figure 4: C code for ISAAC

```
/* & is bitwise AND, ^ is bitwise XOR, a<<b shifts a by b   */
/* ind(mm,x) is bits 2..9 of x, or (floor(x/4) mod 256)*4   */
/* in rngstep barrel(a) was replaced with a^(a<<13) or such */
typedef  unsigned int  u4;  /* unsigned four bytes, 32 bits */
typedef  unsigned char u1;  /* unsigned one  byte,  8  bits */
#define ind(mm,x)  (*(u4 *)((u1 *)(mm) + ((x) & (255<<2))))
#define rngstep(mix,a,b,mm,m,m2,r,x) \
{ \
  x = *m;  \
  a = (a^(mix)) + *(m2++); \
  *(m++) = y = ind(mm,x) + a + b; \
  *(r++) = b = ind(mm,y>>8) + x; \
}

static void isaac(mm,rr,aa,bb,cc)
u4 *mm;  /* Memory: array of SIZE ALPHA-bit terms */
u4 *rr;  /* Results: the sequence, same size as m */
u4 *aa;  /* Accumulator: a single value */
u4 *bb;  /* the previous result */
u4 *cc;  /* Counter: one ALPHA-bit value */
{
  register u4 a,b,x,y,*m,*m2,*r,*mend;
  m=mm; r=rr;
  a = *aa; b = *bb + (++*cc);
  for (m = mm, mend = m2 = m+128; m<mend; )
  {
    rngstep( a<<13, a, b, mm, m, m2, r, x);
    rngstep( a>>6 , a, b, mm, m, m2, r, x);
    rngstep( a<<2 , a, b, mm, m, m2, r, x);
    rngstep( a>>16, a, b, mm, m, m2, r, x);
  }
  for (m2 = mm; m2<mend; )
  {
    rngstep( a<<13, a, b, mm, m, m2, r, x);
    rngstep( a>>6 , a, b, mm, m, m2, r, x);
    rngstep( a<<2 , a, b, mm, m, m2, r, x);
    rngstep( a>>16, a, b, mm, m, m2, r, x);
  }
  *bb = b; *aa = a;
}
```

All told, ISAAC requires an amortized 18.75 instructions to produce each 32-bit value. (With the same optimizations, IA requires an amortized 12.56 instructions to produce each 32-bit value.) There are no cycles in ISAAC shorter than 2^{40} values. There are no bad initial states. The internal state has 8288 bits, so the expected cycle length is 2^{8287} calls (or 2^{8295} 32-bit values). Deducing the internal state appears to be intractable, and the results of ISAAC are unbiased and uniformly distributed.

6 Summary

A sequence of new pseudorandom number generators were developed: IA, IBAA, and ISAAC. Their speed and lack of bias should make them useful for simulations and cryptography. The reader is invited to prove their security (or lack thereof).

Thanks go to Colin Plumb for rephrasing an early version of IBAA, and Niels Jorgen Kruse who found a horrible flaw in a slightly later irreversible version. Thanks go to Hal Finney, Paul Crowley, Peter Boucher, John Kelsey, and the other readers of sci.crypt. Thanks go to Bill Chambers for reviewing a preliminary draft and suggesting a way to guarantee cycle lengths. Thanks go to Manuel Blum for introducing me to cryptography in the first place. All mistakes are my own.

References

[1] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13:850–864, 1984.

[2] W. G. Chambers. private communication. udee205@bay.cc.kcl.ac.uk.

[3] D. Knuth. *Seminumerical Methods*, volume 2, chapter 3. Addison Wesley, 1981.

[4] D. Knuth. *Seminumerical Methods*, volume 3, chapter 5. Addison Wesley, 1981.

[5] V. F. Kolchin. Random mappings. *Optimization Software Inc.*, 1986.

[6] S. Lloyd. Counting binary functions with certain cryptographic properties. *Journal of Cryptology*, 5:107–131, 1992.

[7] G. Marsaglia. A new class of random number generators. *The Annals of Applied Probability*, 1:462–480, 1991.

[8] G. Marsaglia. Diehard. ftp stat.fsu.edu/pub/diehard/diehard.zip, 1995.

[9] A. M. Odlyzko P. Flajolet. Random mapping statistics. *Lecture Notes in Computer Science*, 434:329–354, 1990.

[10] W. Poundstone. *Labyrinths of Reason*. Anchor Press, 1988.

[11] An0nYm0Us UsEr. Rc4 ? sci.crypt, 1994.

[12] A. Yao. Theory and applications of trapdoor functions. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.