# Disproof of Li An-Ping's claims regarding Salsa20

Daniel J. Bernstein [⋆]

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607–7045
snuffle@box.cr.yp.to

## 1 Introduction

In a message posted 28 August 2005 to the ECRYPT Stream Cipher Project forum, and in a paper posted 2 September 2005 titled "Linear approximating for the Cipher Salsa20," Li An-Ping claims that the Salsa20 output blocks are noticeably biased in two ways.

This document demonstrates, by computer experiments with small word sizes, that the two claimed biases do not exist. See Sections 3 and 4. One of the virtues of Salsa20 is that it easily supports this type of experiment.

This document also pinpoints two fundamental errors in An-Ping's argument for these claimed biases. See Sections 5 and 6. There may be additional errors in An-Ping's "analysis," but I see no reason to waste further time with this garbage.

## 2 Generalizing Salsa20

Salsa20-$w/r$ is a variant of Salsa20 with $w$-bit words and $r$ rounds. Here $w \in \{2, 4, 6, 8, \ldots, 32\}$ and $r \in \{2, 4, 6, 8, \ldots, 20\}$. In particular, Salsa20-32/20 is the same as Salsa20.

The point of varying $w$ is that small values of $w$ can easily be studied on the computer. Each key produces $2^{4w}$ output blocks; these blocks can be enumerated when $w$ is small. Clear patterns in output-block statistics for small values of $w$ and $r$ can be safely extrapolated to larger values of $w$ and $r$.

Please note that I am not recommending (e.g.) Salsa20-2/2 as a cipher for general use. On the contrary: Salsa20-2/2 is trivially breakable; among other problems, it has only 256 possible 4-word keys, and only 65536 possible 8-word keys. However, given a hypothesis regarding Salsa20, the cryptanalyst should generalize the hypothesis to Salsa20-$w/r$ and then study a range of pairs $(w, r)$, including breakable pairs.

I make no claims of novelty for this type of generalization. Most block ciphers consist of a large number of similar rounds, and have obvious generalizations to

[⋆] The author was supported by the Alfred P. Sloan Foundation. Date of this document: 2005.09.05.

any number of rounds; studying the effectiveness of attacks on a small number of rounds is an extremely common cryptanalytic technique. Some ciphers also have obvious generalizations to arbitrary word sizes; studying the effectiveness of attacks on small word sizes is a moderately common cryptanalytic technique.

## Definition of Salsa20-$w/r$

Salsa20-$w/r$ uses $w$-bit words wherever Salsa20 uses 32-bit words. The encoding of $w$-bit words as byte strings is not relevant to this document.

The Salsa20-$w/r$ key can be either 4 words or 8 words. The Salsa20-$w/r$ nonce is 2 words. The Salsa20-$w/r$ block counter is 2 words.

Salsa20-$w/r$ starts with a 16-word array filled with the key words, nonce words, block-counter words, and constants. The constants are the same (modulo $2^w$) as the initial constants in Salsa20.

The pattern of add-rotate-xor in each round of Salsa20-$w/r$ is the same as the pattern in each round of Salsa20. The rotation distances (modulo $w$) are $7, 9, 13, 18$, as in Salsa20.

After $r$ rounds, Salsa20-$w/r$ adds its 16 words to the original words, exactly as in Salsa20.

## Software for Salsa20-$w/r$

Here is a reference implementation of Salsa20-$w/r$ for 4-word keys. The first file is `salsa20_wr.h`:

```
#ifndef SALSA20_WR_H
#define SALSA20_WR_H

extern unsigned int salsa20_wr_bits;
extern unsigned int salsa20_wr_max;
extern unsigned int salsa20_wr_rounds;

extern void salsa20_wr_4(unsigned int *,unsigned int *,
  unsigned int *,unsigned int *);

#endif
```

The second file is `salsa20_wr.c`, to be compiled with constant `bits` and `rounds` macros giving the values of $w$ and $r$:

```
#include "salsa20_wr.h"

#define max ((unsigned int) ((2 << (bits - 1)) - 1))

unsigned int salsa20_wr_bits = bits;
unsigned int salsa20_wr_max = max;
unsigned int salsa20_wr_rounds = rounds;
```

```
#define WORD(b) (max & (b))
#define XOR(a,b) ((a) ^ (b))
#define PLUS(a,b) ((a) + (b))
#define ROTATE(a,b) (((a) << ((b) % bits)) \
  | (WORD(a) >> (bits - ((b) % bits))))

static unsigned int tau[4] = {
  'e' + 256 * ('x' + 256 * ('p' + 256 * 'a'))
, 'n' + 256 * ('d' + 256 * (' ' + 256 * '1'))
, '6' + 256 * ('-' + 256 * ('b' + 256 * 'y'))
, 't' + 256 * ('e' + 256 * (' ' + 256 * 'k'))
} ;

void salsa20_wr_4(unsigned int out[16],unsigned int k[4],
  unsigned int iv[2],unsigned int pos[2])
{
  unsigned int in[16];
  unsigned int x[16];
  int i;
  in[0] = tau[0];
  in[1] = k[0]; in[2] = k[1]; in[3] = k[2]; in[4] = k[3];
  in[5] = tau[1];
  in[6] = iv[0]; in[7] = iv[1];
  in[8] = pos[0]; in[9] = pos[1];
  in[10] = tau[2];
  in[11] = k[0]; in[12] = k[1]; in[13] = k[2]; in[14] = k[3];
  in[15] = tau[3];
  for (i = 0;i < 16;++i) x[i] = in[i];
  for (i = rounds;i > 0;i -= 2) {
    x[ 4] = XOR(x[ 4],ROTATE(PLUS(x[ 0],x[12]), 7));
    x[ 8] = XOR(x[ 8],ROTATE(PLUS(x[ 4],x[ 0]), 9));
    x[12] = XOR(x[12],ROTATE(PLUS(x[ 8],x[ 4]),13));
    x[ 0] = XOR(x[ 0],ROTATE(PLUS(x[12],x[ 8]),18));
    x[ 9] = XOR(x[ 9],ROTATE(PLUS(x[ 5],x[ 1]), 7));
    x[13] = XOR(x[13],ROTATE(PLUS(x[ 9],x[ 5]), 9));
    x[ 1] = XOR(x[ 1],ROTATE(PLUS(x[13],x[ 9]),13));
    x[ 5] = XOR(x[ 5],ROTATE(PLUS(x[ 1],x[13]),18));
    x[14] = XOR(x[14],ROTATE(PLUS(x[10],x[ 6]), 7));
    x[ 2] = XOR(x[ 2],ROTATE(PLUS(x[14],x[10]), 9));
    x[ 6] = XOR(x[ 6],ROTATE(PLUS(x[ 2],x[14]),13));
    x[10] = XOR(x[10],ROTATE(PLUS(x[ 6],x[ 2]),18));
    x[ 3] = XOR(x[ 3],ROTATE(PLUS(x[15],x[11]), 7));
    x[ 7] = XOR(x[ 7],ROTATE(PLUS(x[ 3],x[15]), 9));
    x[11] = XOR(x[11],ROTATE(PLUS(x[ 7],x[ 3]),13));
```

```
      x[15] = XOR(x[15],ROTATE(PLUS(x[11],x[ 7]),18));
      x[ 1] = XOR(x[ 1],ROTATE(PLUS(x[ 0],x[ 3]), 7));
      x[ 2] = XOR(x[ 2],ROTATE(PLUS(x[ 1],x[ 0]), 9));
      x[ 3] = XOR(x[ 3],ROTATE(PLUS(x[ 2],x[ 1]),13));
      x[ 0] = XOR(x[ 0],ROTATE(PLUS(x[ 3],x[ 2]),18));
      x[ 6] = XOR(x[ 6],ROTATE(PLUS(x[ 5],x[ 4]), 7));
      x[ 7] = XOR(x[ 7],ROTATE(PLUS(x[ 6],x[ 5]), 9));
      x[ 4] = XOR(x[ 4],ROTATE(PLUS(x[ 7],x[ 6]),13));
      x[ 5] = XOR(x[ 5],ROTATE(PLUS(x[ 4],x[ 7]),18));
      x[11] = XOR(x[11],ROTATE(PLUS(x[10],x[ 9]), 7));
      x[ 8] = XOR(x[ 8],ROTATE(PLUS(x[11],x[10]), 9));
      x[ 9] = XOR(x[ 9],ROTATE(PLUS(x[ 8],x[11]),13));
      x[10] = XOR(x[10],ROTATE(PLUS(x[ 9],x[ 8]),18));
      x[12] = XOR(x[12],ROTATE(PLUS(x[15],x[14]), 7));
      x[13] = XOR(x[13],ROTATE(PLUS(x[12],x[15]), 9));
      x[14] = XOR(x[14],ROTATE(PLUS(x[13],x[12]),13));
      x[15] = XOR(x[15],ROTATE(PLUS(x[14],x[13]),18));
    }
    for (i = 0;i < 16;++i) out[i] = WORD(PLUS(x[i],in[i]));
  }
```

## 3 The first claimed bias

Given a 16-word block $z = (z[0], z[1], \ldots, z[15])$, define $b_1(z) \in \{0, 1\}$ as the xor of the $9w$ bits in the 9 words $z[0], z[1], z[2], z[4], z[5], z[6], z[8], z[9], z[10]$.

For each Salsa20-$w/r$ key $k$, consider the $2^{4w}$ Salsa20-$w/r$ output blocks for that key, and define $B_1(k)$ as the average value of $b_1(z)$ for these blocks $z$.

This section reports computer calculations of $B_1(k)$ for many keys $k$. The computer calculations show that the deviation of $B_1(k)$ from 0.5 is very close to $0.5/2^{2w}$, as one would expect for a strong cipher. This section then discusses An-Ping's incorrect claims that $B_1(k)$ has a much larger deviation from 0.5, namely $2^{-(w+1)}$.

### Computer calculations

I considered each pair $(w, r)$ with $w \in \{2, 4, 6\}$ and $r \in \{2, 4, 6, 10, 14, 20\}$, For each $(w, r)$, I used FreeBSD's `/dev/urandom` cryptographic random number generator to create 1000 Salsa20-$w/r$ keys, each having $4w$ bits. I computed the exact value of $B_1(k)$ for each key $k$ by computing all of the $2^{4w}$ Salsa20-$w/r$ output blocks for $k$.

Consider, for example, $w = 6$ and $r = 20$, and consider the key $k$ with words `26,3b,21,2c` hexadecimal. There are $2^{4w} = 16777216$ output blocks for this key, and 8389434 of these blocks $z$ have $b_1(z) = 1$, so $B_1(k) = 8389434/16777216 = 0.500119388103485107421875$. The computer output says

```
6 bits, 20 rounds: B1(26,3b,21,2c) - 0.5 =
0.0000492334365844727
```

indicating that $B_1(k)$ is approximately $0.5 + 0.0000492334365844727$. The only approximations here are in a floating-point division of 8389434 by 16777216, and in printing out the resulting floating-point number; the value 8389434 was calculated exactly, by computation of all 16777216 output blocks. The key is recorded to allow separate verification of each line.

Then, for each $(w, r)$, I averaged $(B_1(k) - 0.5)^2$ for these 1000 keys $k$, to estimate the average of $(B_1(k) - 0.5)^2$ for all keys $k$, i.e., to estimate the squared deviation of $B_1(k)$ from 0.5. The computer output says, for example,

```
4 bits, 20 rounds: B1 deviation estimate 2^(-9.004712)
using 1000 keys
```

indicating that the sum of $(B_1(k) - 0.5)^2$ for 1000 Salsa20-4/20 keys $k$ was approximately $1000 \cdot (2^{-9.004712})^2$.

The following table shows the exponent of 2 in this estimate for each $(w, r)$—for example, the above exponent $-9.004712$ for $(w, r) = (4, 20)$:

| | 2 | 4 | 6 |
|---|---|---|---|
| 2 | −5.048867 | −9.041757 | −12.964445 |
| 4 | −4.975620 | −8.998856 | −12.971220 |
| 6 | −5.056969 | −8.971984 | −12.956509 |
| 10 | −5.024288 | −8.971151 | −12.939057 |
| 14 | −5.074450 | −8.988275 | −12.964276 |
| 20 | −5.043388 | −9.004712 | −12.960415 |

These experiments demonstrate conclusively, for each pair $(w, r)$ in the table, that the deviation of $B_1(k)$ from 0.5 is close to $0.5/2^{2w}$. The experiments are entirely consistent with the hypothesis that the deviation of $B_1(k)$ from 0.5 is extremely close to $0.5/2^{2w}$ for all pairs $(w, r)$.

Computing the $w = 4$ column of this table took about $2^{38}$ Athlon-64 cycles: about $2^{26}$ cycles for each $(r, k)$. Each addition of 2 to $w$ makes the computation of $B_1(k)$ take $2^8$ times as long. Computing the exact value of $B_1(k)$ for a single $k$ with $w = 32$ would take about $2^{138}$ cycles. For some purposes it is enough to approximate the value of $B_1(k)$: for example, one could confidently compute $B_1(k)$ to within $1/2^{34}$ in only about $2^{80}$ cycles.

I'm not going to delay this paper waiting for results for $w = 8$, and I probably won't bother completing the $w = 8$ computation, but $B_1(k) - 0.5$ is on the scale of $2^{-17} = 0.00000762939453125$ for all the $k$'s tested so far for $w = 8$:

$$
\begin{aligned}
B_1(\texttt{4c,d2,38,3e}) - 0.5 &= -0.0000071621034294367, \\
B_1(\texttt{a5,e0,ee,7}) - 0.5 &= -0.0000095204450190067, \\
B_1(\texttt{cb,7a,be,56}) - 0.5 &= \phantom{-}0.0000011851079761982.
\end{aligned}
$$

## What one would expect for a strong cipher

Instead of considering output blocks $z$ from Salsa20-$w/r$, consider successive blocks $z$ of a one-time pad. The bits $b_1(z)$ are then independent uniform random bits.

This modification replaces $B_1(k)$ with the average $\alpha$ of $2^{4w}$ independent uniform random bits. The distribution of this average $\alpha$ is close to a normal distribution around 0.5, with deviation $0.5/2^{2w}$. In particular, the average of $\alpha$ is exactly 0.5, and the deviation of $\alpha$ is exactly $0.5/2^{2w}$.

The same modification replaces 1000 values of $B_1(k)$ with 1000 independent values of $\alpha$, and thus replaces the average of 1000 squares $(B_1(k) - 0.5)^2$ with the average of 1000 squares $(\alpha - 0.5)^2$. The square root of the latter average has deviation roughly $(0.5/2^{2w})/\sqrt{1000}$ from $0.5/2^{2w}$, so it will usually be between $2^{-(2w+0.95)}$ and $2^{-(2w+1.05)}$, just like the numbers in the table above.

(Of course, a distribution is not completely characterized by its average and deviation. I've inspected the distribution of $B_1(k)$ in more detail and found no surprises.)

Summary: The distribution of $B_1(k)$ for Salsa20-$w/r$ is entirely consistent with what one would expect for a strong cipher.

## The claims of bias

An-Ping defines $d(v)$ for a "binary segment $v$" as "the XOR over all the bits of $v$." An-Ping defines

```
z' = z[0][0]^z[0][1]^z[0][2]
  ^z[1][0]^z[1][1]^z[1][2]
  ^z[2][0]^z[2][1]^z[2][2].
```

where $z[i][j]$ is "the output ... of Salsa20." In other words, $b_1$ for a Salsa20 output block is the same as An-Ping's $d(z')$.

An-Ping states claims of bias using the following notation: "If $v$ is a variable over the domain $K$ ... define $\delta(v) = (|K| - 2 * \sum d(v) \mid v \in K)/|K|$." In other words, as $v$ ranges uniformly over $K$, the bit $d(v)$ has average value $0.5(1-\delta(v))$.

An-Ping claims for Salsa20 that "When the secret key is fixed ... $\delta(z') = 1/2^{32}$." In other words, An-Ping claims for each key $k$ that the average value of $b_1 = d(z')$ for all of $k$'s output blocks is $0.5(1 - 1/2^{32})$. In other words, An-Ping claims for each key $k$ that $B_1(k) = 0.5(1 - 1/2^{32})$. Note that, if this were true, then the deviation of $B_1(k)$ from 0.5 would be exactly $0.5/2^{32}$.

In the same message, An-Ping claims that the average of $b_1$ for all outputs *for all keys* is $0.5(1 - 1/2^{96})$. This claim contradicts An-Ping's previous claim: if $B_1(k) = 0.5(1 - 1/2^{32})$ for every $k$ then the overall average must also be $0.5(1 - 1/2^{32})$.

In a subsequent message, An-Ping makes a considerably weaker claim, namely that "$|\delta(z')| \geq 1/2^{32}$." In other words, An-Ping claims, for each Salsa20 key $k$, that $|B_1(k) - 0.5| \geq 0.5/2^{32}$. Note that, if this were true, then the deviation of $B_1(k)$ from 0.5 would be at least $0.5/2^{32}$.

Apparently An-Ping is claiming, more generally, that $|B_1(k) - 0.5| \geq 0.5/2^w$ for Salsa20-$w/r$. The word size enters into An-Ping's "analysis" in only one way: namely, the xor of all bits of $x+y, x, y$ has variance $0.5/2^w$ when $x, y$ are uniform random words. An-Ping has not claimed (despite repeated prompting) that $w$ has any other effect on the claimed bias. As for the number of rounds, there is no use of the number of rounds in An-Ping's "analysis," and in fact An-Ping has explicitly written "It seems that increasing the number of round will not have much effection for our analysis."

Each of An-Ping's claims implies that the deviation of $B_1(k)$ from 0.5 is at least $0.5/2^w$. But the computer experiments show conclusively for small values of $w$, and strongly suggest for all values of $w$, that the deviation of $B_1(k)$ from 0.5 is very close to $0.5/2^{2w}$, contradicting An-Ping's claims.

An-Ping also presents a table titled "The Bias in the Linear Approximating $1, 2$" claiming that $|B_1(k) - 0.5|$ is on the scale of $2^{-16}$ for several test keys $k$. This table turns out to be incorrectly labelled, both in its title and in its column headings. What An-Ping actually computed was an *estimate* of $B_1(k)$ based on just $2^{30}$ output blocks. It is completely unsurprising to see deviations on the scale of $0.5/\sqrt{2^{30}}$ in these estimates. Analogous comments apply in Section 4.

## Security claims

After claiming for each key $k$ that $B_1(k) = 0.5(1 - 1/2^{32})$, An-Ping claims that this formula for $B_1(k)$ produces a "distinguishing attack ... for Salsa20 with at most $2^{64}$ blocks of keystream." I agree that, if An-Ping's formula for $B_1(k)$ were correct, then it would allow a $2^{64}$-block distinguishing attack with noticeable success chance; but An-Ping's formula for $B_1(k)$ is not even close to correct. There is no evidence of any flaw in Salsa20.

After claiming that the average of $b_1(z)$ for all output blocks $z$ *for all keys* is $0.5(1-1/2^{96})$, An-Ping claims a "distinguishing attack" using "about $2^{192}$ blocks of keystream." This claim is nonsense. A distinguisher, by definition, tries to tell the difference between *one* key and a one-time pad. A Salsa20 key produces only $2^{128}$ output blocks. Apparently An-Ping is confused about the definition of cipher security.

Analogous remarks apply in Section 4 below.

## Software for these calculations

The following program `bias1.c` computes $B_1(k)$ for 1000 keys $k$:

```
#include <stdio.h>
#include "salsa20_wr.h"

int streamb1(unsigned int k[4],
  unsigned int iv[2],unsigned int pos[2])
{
  unsigned int out[16];
  unsigned int x;
  unsigned int result;
  salsa20_wr_4(out,k,iv,pos);
  x = out[0] ^ out[1] ^ out[2];
  x ^= out[4] ^ out[5] ^ out[6];
  x ^= out[8] ^ out[9] ^ out[10];
  result = 0;
  while (x) {
    result ^= x & 1;
    x >>= 1;
  }
  return result;
}

double B1(unsigned int k[4])
{
  unsigned int iv[2];
  unsigned int pos[2];
  long long blocks = 0;
  long long b1total = 0;

  for (iv[0] = 0;iv[0] <= salsa20_wr_max;++iv[0])
    for (iv[1] = 0;iv[1] <= salsa20_wr_max;++iv[1])
      for (pos[0] = 0;pos[0] <= salsa20_wr_max;++pos[0])
        for (pos[1] = 0;pos[1] <= salsa20_wr_max;++pos[1]) {
          ++blocks;
          b1total += streamb1(k,iv,pos);
        }

  return b1total / (1.0 * blocks);
}

main(int argc,char **argv)
{
  unsigned int k[4];
  long long i;
  double t;
  double sumsq = 0;
  long long numkeys;

  numkeys = 100;
  if (argv[1]) numkeys = atoi(argv[1]);

  for (i = 0;i < numkeys;++i) {
    k[0] = random32() & salsa20_wr_max;
    k[1] = random32() & salsa20_wr_max;
    k[2] = random32() & salsa20_wr_max;
    k[3] = random32() & salsa20_wr_max;
    t = B1(k) - 0.5;
    printf("%d bits, %d rounds: "
      ,salsa20_wr_bits
      ,salsa20_wr_rounds
      );
    printf("B1(%x,%x,%x,%x) - 0.5 = %20.19lf\n"
      ,k[0],k[1],k[2],k[3]
      ,t
      );
    fflush(stdout);
    sumsq += t * t;
  }
  printf("%d bits, %d rounds: "
    ,salsa20_wr_bits
    ,salsa20_wr_rounds
    );
  printf("B1 deviation estimate 2^(%lf) using %lld keys\n"
    ,log(sqrt(sumsq / numkeys)) / log(2)
    ,numkeys
    );
  return 0;
}
```

The `random32()` function used to generate keys is declared in `random32.h`:

```
#ifndef RANDOM32_H
#define RANDOM32_H

extern unsigned int random32(void);

#endif
```

The implementation, `random32.c`, takes data from standard input:

```
#include "random32.h"

unsigned int random32(void)
{
  unsigned int result;
  result = 255 & getchar();
  result *= 256;
  result += 255 & getchar();
  result *= 256;
  result += 255 & getchar();
  result *= 256;
  result += 255 & getchar();
  return result;
}
```

I used the following shell script to run the computation:

```
exec 2>&1

CC='gcc -O3 -march=athlon64'

$CC -c random32.c
for bits in 2 4 6 8 32
do
  for r in 2 4 6 8 10 12 14 16 18 20
  do
    $CC -c salsa20_wr.c -Drounds=$r -Dbits=$bits
    mv salsa20_wr.o salsa20_$bits,$r.o
  done
done

$CC -c salsa20_wr_test.c
$CC -o salsa20_wr_test salsa20_wr_test.o salsa20_32,20.o
time ./salsa20_wr_test < /dev/urandom

$CC -c bias1.c
$CC -c bias2.c
for bits in 2 4 6 8
do
  for r in 20 2 4 10 6 14
  do
    $CC -o bias1 bias1.o salsa20_$bits,$r.o random32.o -lm
    time ./bias1 1000 < /dev/urandom
    $CC -o bias2 bias2.o salsa20_$bits,$r.o random32.o -lm
    time ./bias2 1000 < /dev/urandom
  done
done
```

The program `bias2.c` is shown in Section 4. The program `salsa20_wr_test.c`, not shown in this paper, checks for consistency between `salsa20_wr_4` and the standard Salsa20 code.

## 4 The second claimed bias

Given a 16-word block $z = (z[0], z[1], \ldots, z[15])$, define $b_2(z) \in \{0, 1\}$ as the xor of the following bits:

- bit $18 + 18 - 7$ of $z[0]$;
- bit $7 + 18 - 7$ of $z[1]$;
- bit $9 + 18 - 7$ of $z[2]$;
- bits $13 + 18 - 7$ and $18 - 7$ of $z[3]$;
- bit $7 + 18 - 7$ of $z[4]$;
- bit $7 + 7 - 7$ of $z[5]$;
- bit $7 + 9 - 7$ of $z[6]$;
- bits $7 + 13 - 7$ and $7 - 7$ of $z[7]$;
- bit $9 + 18 - 7$ of $z[8]$;
- bit $7 + 9 - 7$ of $z[9]$;
- bit $9 + 9 - 7$ of $z[10]$;
- bits $9 + 13 - 7$ and $9 - 7$ of $z[11]$;
- bits $13 + 18 - 7$ and $18 - 7$ of $z[12]$;
- bits $7 + 13 - 7$ and $7 - 7$ of $z[13]$;
- bits $9 + 13 - 7$ and $9 - 7$ of $z[14]$;
- bits $13 + 13 - 7$ and $0 - 7$ of $z[15]$.

The numbers $7, 9, 13, 18$ appearing in bit positions above are the Salsa20 rotation distances. As usual, bit positions are interpreted modulo $w$; for example, bit $0-7$ of $z[15]$ is bit 25 of $z[15]$ when $w = 32$.

For each Salsa20-$w/r$ key $k$, consider the $2^{4w}$ Salsa20-$w/r$ output blocks for that key, and define $B_2(k)$ as the average value of $b_2(z)$ for these blocks $z$.

This section reports computer calculations of $B_2(k)$ for many keys $k$. The computer calculations show that the deviation of $B_2(k)$ from 0.5 is very close to $0.5/2^{2w}$, as one would expect for a strong cipher. This section then discusses An-Ping's incorrect claims that $B_2(k)$ has a much larger deviation from 0.5.

## Computer calculations

I repeated the calculations described in Section 3 but with $b_2$ and $B_2$ in place of $b_1$ and $B_1$. Here is the resulting table of logs base 2 of estimated deviations of $B_2(k)$ from 0.5:

| | 2 | 4 | 6 |
|---|---|---|---|
| 2 | −5.013194 | −9.012149 | −12.988842 |
| 4 | −5.080234 | −8.984445 | −13.016905 |
| 6 | −5.117803 | −8.992286 | −12.975413 |
| 10 | −5.015275 | −8.962179 | −13.075581 |
| 14 | −5.041463 | −8.987249 | −13.038340 |
| 20 | −4.896893 | −9.002967 | −12.962563 |

These experiments demonstrate conclusively, for each pair $(w, r)$ in the table, that the deviation of $B_2(k)$ from 0.5 is close to $0.5/2^{2w}$. The experiments are entirely consistent with the hypothesis that the deviation of $B_2(k)$ from 0.5 is extremely close to $0.5/2^{2w}$ for all pairs $(w, r)$.

## What one would expect for a strong cipher

Instead of considering output blocks $z$ from Salsa20-$w/r$, consider successive blocks $z$ of a one-time pad. The bits $b_1(z)$ are then independent uniform random bits. In particular, as in Section 3, these bits produce table entries with deviation roughly $(0.5/2^{2w})/\sqrt{1000}$ from $0.5/2^{2w}$, just like the numbers in the table above.

Summary: The distribution of $B_2(k)$ for Salsa20-$w/r$, like the distribution of $B_1(k)$, is entirely consistent with what one would expect for a strong cipher.

## The claims of bias

An-Ping defines, for $w = 32$,

```
        z0 = z[0][0]{29}^z[0][1]{18}^z[0][2]{20}^z[0][3]{11,24}
            ^z[1][0]{18}^z[1][1]{7}^z[1][2]{9}^z[1][3]{0,13}
            ^z[2][0]{20}^z[2][1]{9}^z[2][2]{11}^z[2][3]{2,15}
      ^z[3][0]{11,24}^z[3][1]{0,13}^z[3][2]{2,15}^z[3][3]{25,19}.
```

Here $z[i][j]$ is "the output ... of Salsa20," as in Section 3; $z[a,b]\{i\}$ is "the $i$-th bit of $z[a,b]$"; and $z[a,b]\{i,j\}$ is "the XOR of the $i$-th bit, $j$-th bit of $z[a,b]$." In other words, $b_2$ for a Salsa20 output block is the same as An-Ping's $d(z_0)$; recall from Section 3 that An-Ping defines $d(v)$ for a "binary segment $v$" as "the XOR over all the bits of $v$."

Tracing the origin of the numbers $29, 18, 20, \ldots$ through An-Ping's paper reveals that they arise as $18+18-7, 18, 18+9-7, \ldots$, as shown in my definition of $b_2$. In particular, the trailing 25 arises as $-7$ modulo 32.

An-Ping, using the notation $\delta(v)$ discussed in Section 3, claims for Salsa20 that "$\delta(z_0) = 1/2^{22}$." In other words, An-Ping claims for each key $k$ that the average value of $b_2 = d(z_0)$ for all of $k$'s output blocks is $0.5(1-1/2^{22})$. In other words, An-Ping claims for each key $k$ that $B_2(k) = 0.5(1-1/2^{22})$. Note that, if this were true, then the deviation of $B_2(k)$ from 0.5 would be exactly $0.5/2^{22}$.

As in Section 3, An-Ping makes a contradictory claim regarding the average of $B_2(k)$, and then makes the weaker claim that $B_2(k)$ always has distance at least $0.5/2^{22}$ from 0.5. If the weaker claim were true then the deviation from $B_2(k)$ from 0.5 would be at least $0.5/2^{22}$.

Where does this 22 come from? Answer: An-Ping computes 22 as half of the sum of $9-7 \bmod 32 = 2$, $13-7 \bmod 32 = 6$, $18-7 \bmod 32 = 11$, and $0-7 \bmod 32 = 25$. The corresponding numbers for smaller $w$ are therefore as follows:

- 1 for $w = 2$: half the sum of $9-7 \bmod 2 = 0$, $13-7 \bmod 2 = 0$, $18-7 \bmod 2 = 1$, and $0-7 \bmod 2 = 1$.
- 4 for $w = 4$: half the sum of $9-7 \bmod 4 = 2$, $13-7 \bmod 4 = 2$, $18-7 \bmod 4 = 3$, and $0-7 \bmod 4 = 1$.
- 6 for $w = 6$: half the sum of $9-7 \bmod 6 = 2$, $13-7 \bmod 6 = 0$, $18-7 \bmod 6 = 5$, and $0-7 \bmod 6 = 5$.

An-Ping's "analysis" makes no other reference to the word size, and An-Ping specifically disclaims dependence on the number of rounds. Apparently An-Ping is claiming a deviation of at least $0.5/2^1$ for Salsa20-2/$r$; a deviation of at least $0.5/2^4$ for Salsa20-4/$r$; a deviation of at least $0.5/2^6$ for Salsa20-6/$r$; etc.

As in Section 3, the computer experiments show conclusively for small values of $w$, and strongly suggest for all values of $w$, that the deviation of $B_2(k)$ from 0.5 is very close to $0.5/2^{2w}$ for Salsa20-$w/r$, contradicting An-Ping's claims.

**Software for these calculations**

The following program `bias2.c` computes $B_2(k)$ for 1000 keys $k$:

```
#include <stdio.h>
#include "salsa20_wr.h"

int bit(unsigned int x,unsigned int b)
{
  b %= salsa20_wr_bits;
  b += salsa20_wr_bits;
  b %= salsa20_wr_bits;
  x >>= b;
  x &= 1;
  return x;
}

int streamb2(unsigned int k[4],
  unsigned int iv[2],unsigned int pos[2])
{
  unsigned int out[16];
  unsigned int result;

  salsa20_wr_4(out,k,iv,pos);

  result  = bit(out[ 0],18 + 18 - 7);
  result ^= bit(out[ 1], 7 + 18 - 7);
  result ^= bit(out[ 2], 9 + 18 - 7);
  result ^= bit(out[ 3],13 + 18 - 7);
  result ^= bit(out[ 3],      18 - 7);

  result ^= bit(out[ 4], 7 + 18 - 7);
  result ^= bit(out[ 5], 7 +  7 - 7);
  result ^= bit(out[ 6], 7 +  9 - 7);
  result ^= bit(out[ 7], 7 + 13 - 7);
  result ^= bit(out[ 7], 7      - 7);

  result ^= bit(out[ 8], 9 + 18 - 7);
  result ^= bit(out[ 9], 9 +  7 - 7);
  result ^= bit(out[10], 9 +  9 - 7);
  result ^= bit(out[11], 9 + 13 - 7);
  result ^= bit(out[11], 9      - 7);

  result ^= bit(out[12],18 + 13 - 7);
  result ^= bit(out[12],18      - 7);
  result ^= bit(out[13], 7 + 13 - 7);
  result ^= bit(out[13], 7      - 7);
  result ^= bit(out[14], 9 + 13 - 7);
  result ^= bit(out[14], 9      - 7);
  result ^= bit(out[15],13 + 13 - 7);
  result ^= bit(out[15], 0      - 7);

  return result;
}

double B2(unsigned int k[4])
{
  unsigned int iv[2];
  unsigned int pos[2];
  long long blocks = 0;
  long long b1total = 0;

  for (iv[0] = 0;iv[0] <= salsa20_wr_max;++iv[0])
    for (iv[1] = 0;iv[1] <= salsa20_wr_max;++iv[1])
      for (pos[0] = 0;pos[0] <= salsa20_wr_max;++pos[0])
        for (pos[1] = 0;pos[1] <= salsa20_wr_max;++pos[1]) {
          ++blocks;
          b1total += streamb2(k,iv,pos);
        }

  return b1total / (1.0 * blocks);
}

main(int argc,char **argv)
{
  unsigned int k[4];
  long long i;
  double t;
  double sumsq = 0;
  long long numkeys;

  numkeys = 100;
  if (argv[1]) numkeys = atoi(argv[1]);
  for (i = 0;i < numkeys;++i) {
    k[0] = random32() & salsa20_wr_max;
    k[1] = random32() & salsa20_wr_max;
    k[2] = random32() & salsa20_wr_max;
    k[3] = random32() & salsa20_wr_max;
    t = B2(k) - 0.5;
    printf("%d bits, %d rounds: "
      ,salsa20_wr_bits
      ,salsa20_wr_rounds
      );
    printf("B2(%x,%x,%x,%x) - 0.5 = %20.19lf\n"
      ,k[0],k[1],k[2],k[3]
      ,t
      );
    fflush(stdout);
    sumsq += t * t;
  }
  printf("%d bits, %d rounds: "
    ,salsa20_wr_bits
    ,salsa20_wr_rounds
    );
  printf("B2 deviation estimate 2^(%lf) using %lld keys\n"
    ,log(sqrt(sumsq / numkeys)) / log(2)
    ,numkeys
    );
  return 0;
}
```

### Further generalizations

One can generalize to any set of rotation constants. I've already explained how the rotation constants percolate through An-Ping's "analysis."

In particular, replace $7, 9, 13, 18$ with $0, 0, 0, 8$. An-Ping's generalized claim then states a whopping 0.015625 bias for the xor of the following bits: bit 16 of $z[0]$; bit 8 of $z[1], z[2], z[4], z[8]$; and bit 0 of $z[5], z[6], z[9], z[10]$.

I disproved this claim for $(w, r) = (32, 20)$ by trying 1048576 inputs for one key. The average xor was 0.49957275390625, a completely implausible 30 deviations away from the claimed below-0.484375-or-above-0.515625.

## 5 Bias calculation for dummies

Salsa20 combines a key, nonce, and block counter into a 16-word array $s_0$. It makes a simple modification to the array, producing a new 16-word array $s_1$;

makes another simple modification to the array, producing a new 16-word array $s_2$; and so on.

An-Ping specifies simple functions $f_0, f_1, f_2, \ldots, f_r$ and considers the bits $f_0(s_0), f_1(s_1), f_2(s_2), \ldots, f_r(s_r)$. An-Ping uses the following strategy to calculate the average value of $f_r(s_r)$:

- Use the simple relationship between $s_i$ and $s_{i+1}$ to compute the average value of $f_{i+1}(s_{i+1}) - f_i(s_i)$; also compute the average value of $f_0(s_0)$.
- Combine the average of $f_0(s_0)$, the average of $f_1(s_1) \oplus f_0(s_0)$, the average of $f_2(s_2) \oplus f_1(s_1)$, and so on through the average of $f_r(s_r) \oplus f_{r-1}(s_{r-1})$, to calculate the average value of $f_r(s_r) = f_0(s_0) \oplus (f_1(s_1) \oplus f_0(s_0)) \oplus (f_2(s_2) \oplus f_1(s_1)) \oplus \cdots \oplus (f_r(s_r) \oplus f_{r-1}(s_{r-1}))$.

The point of this section is that the above strategy is fundamentally flawed. To compute the average of $a_0 \oplus a_1 \oplus \cdots$, one needs much more information than merely the average of $a_0$, the average of $a_1$, etc.

Consider, for example, a random bit $a_0$ having average 0.6, and another random bit $a_1$ having average 0.6. (Note to novices: random variables are not necessarily uniform, and not necessarily independent.) Here are some possibilities for the average of $a_0 \oplus a_1$:

- 0. Consider a pair $(a_0, a_1)$ equalling $(0, 0)$ with probability 0.4 and $(1, 1)$ with probability 0.6.
- 0.48. Consider a pair $(a_0, a_1)$ equalling $(0, 0)$ with probability 0.16, $(0, 1)$ with probability 0.24, $(1, 0)$ with probability 0.24, and $(1, 1)$ with probability 0.36.
- 0.5. Consider a pair $(a_0, a_1)$ equalling $(0, 0)$ with probability 0.15, $(0, 1)$ with probability 0.25, $(1, 0)$ with probability 0.25, and $(1, 1)$ with probability 0.35.
- 0.8. Consider a pair $(a_0, a_1)$ equalling $(0, 1)$ with probability 0.4, $(1, 0)$ with probability 0.4, and $(1, 1)$ with probability 0.2.

Without studying the joint distribution of $a_0$ and $a_1$, one has no basis for guessing where the average lies between 0 and 0.8. Any particular guess is wrong for two of the above examples and for many other examples.

An-Ping never analyzes the joint distribution of the bits $f_0(s_0)$, $f_1(s_1) \oplus f_0(s_0)$, $f_2(s_2) \oplus f_1(s_1)$, etc. An-Ping simply leaps from the claimed averages of those bits to a claim regarding the average of $f_r(s_r)$, the xor of those bits. It doesn't matter how An-Ping calculates the claimed average of $f_r(s_r)$ from the claimed averages of $f_0(s_0)$, $f_1(s_1) \oplus f_0(s_0)$, $f_2(s_2) \oplus f_1(s_1)$, etc.; the entire method is indefensible, and the final number is nothing more than wild speculation. It is entirely unsurprising that An-Ping's claims turned out to be incorrect.

## 6 Salsa20 for dummies

An-Ping analyzes $b_1(z)$, for a Salsa20 output block $z$, by expressing it as a xor of various quantities $\ell(a, b)$ for various intermediate words $a, b$ in the Salsa20 state; and then averaging each $\ell(a, b)$ separately. This strategy is fundamentally flawed, as discussed in Section 6, but let's ignore that for the moment, and look at the details of the first step.

The point of this section is that An-Ping's formula for $b_1(z)$ is wrong. An-Ping obtains the formula by xor'ing simple round-by-round formulas; the problem is that most of those simple formulas are wrong. An-Ping obtains the simple formulas by applying various row symmetries and column symmetries to a correct formula; the problem is that Salsa20 does not have those symmetries. A computer experiment immediately finds many counterexamples to An-Ping's formula for $b_1(z)$.

An-Ping's formula for $b_2(z)$ is based on exactly the same chain of errors, and presumably half of all inputs are counterexamples, although I haven't bothered with computer experiments in this case.

## Understanding quarter rounds

The quarterround function in Salsa20 converts a 4-word array $(y_0, y_1, y_2, y_3)$ into quarterround$(y_0, y_1, y_2, y_3) = (z_0, z_1, z_2, z_3)$ where

$$\begin{aligned} z_1 &= y_1 \oplus ((y_0 + y_3) \lll 7), \\ z_2 &= y_2 \oplus ((z_1 + y_0) \lll 9), \\ z_3 &= y_3 \oplus ((z_2 + z_1) \lll 13), \\ z_0 &= y_0 \oplus ((z_3 + z_2) \lll 18). \end{aligned}$$

An-Ping defines $d$ as the xor-all-bits function and correctly observes that

$$\begin{aligned} d(z_1) &= d(y_1) \oplus d(y_0 + y_3), \\ d(z_2) &= d(y_2) \oplus d(z_1 + y_0), \\ d(z_3) &= d(y_3) \oplus d(z_2 + z_1), \\ d(z_0) &= d(y_0) \oplus d(z_3 + z_2), \end{aligned}$$

since $d$ is invariant under rotations. An-Ping correctly concludes that $d(y_0) \oplus d(y_1) \oplus d(y_2) \oplus d(z_0) \oplus d(z_1) \oplus d(z_2) = \ell(y_0, y_3) \oplus \ell(z_1, y_0) \oplus \ell(z_2, z_1) \oplus \ell(z_3, z_2)$ where $\ell(a, b) = d(a + b) \oplus d(a) \oplus d(b)$.

## Understanding row rounds

The rowround function in Salsa20 converts a 16-word array

$$y = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$$

into a 16-word array

$$z = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix}$$

by four applications of the quarterround function. For example, $(z_0, z_1, z_2, z_3) = \text{quarterround}(y_0, y_1, y_2, y_3)$, so $d(y_0) \oplus d(y_1) \oplus d(y_2) \oplus d(z_0) \oplus d(z_1) \oplus d(z_2) = \ell(y_0, y_3) \oplus \ell(z_1, y_0) \oplus \ell(z_2, z_1) \oplus \ell(z_3, z_2)$ as discussed above. Let's call this the "012 formula."

Define $\text{topleft}(y) = y_0 \oplus y_1 \oplus y_2 \oplus y_4 \oplus y_5 \oplus y_6 \oplus y_8 \oplus y_9 \oplus y_{10}$. An-Ping claims that $d(\text{topleft}(y)) \oplus d(\text{topleft}(z))$ is the xor of various $\ell$ values. It is reasonably clear (although An-Ping is too careless to state this explicitly) that An-Ping's claimed formula was obtained by xoring the 012 formula with a claimed "456 formula" and a claimed "8910 formula." The latter two formulas were obtained as follows:

- An-Ping assumes that the second row is handled in the same way as the first row: $(z_4, z_5, z_6, z_7) = \text{quarterround}(y_4, y_5, y_6, y_7)$. An-Ping deduces the "456 formula" $d(y_4) \oplus d(y_5) \oplus d(y_6) \oplus d(z_4) \oplus d(z_5) \oplus d(z_6) = \ell(y_4, y_7) \oplus \ell(z_5, y_4) \oplus \ell(z_6, z_5) \oplus \ell(z_7, z_6)$.
- An-Ping assumes that the third row is handled in the same way as the first row: $(z_8, z_9, z_{10}, z_{11}) = \text{quarterround}(y_8, y_9, y_{10}, y_{11})$. An-Ping deduces the "8910 formula" $d(y_8) \oplus d(y_9) \oplus d(y_{10}) \oplus d(z_8) \oplus d(z_9) \oplus d(z_{10}) = \ell(y_8, y_{11}) \oplus \ell(z_9, y_8) \oplus \ell(z_{10}, z_9) \oplus \ell(z_{11}, z_{10})$.

The problem is that An-Ping's assumptions are false. The second row is not handled in exactly the same way as the first row; a *rotation* of the second row is handled in exactly the same way as the first row. Specifically, the definition of Salsa20 states that $(z_5, z_6, z_7, z_4) = \text{quarterround}(y_5, y_6, y_7, y_4)$, implying a "567 formula" $d(y_5) \oplus d(y_6) \oplus d(y_7) \oplus d(z_5) \oplus d(z_6) \oplus d(z_7) = \ell(y_5, y_4) \oplus \ell(z_6, y_5) \oplus \ell(z_7, z_6) \oplus \ell(z_4, z_7)$. An-Ping's 456 formula is unjustified and, presumably, is incorrect half the time.

## Understanding column rounds

The rowround function in Salsa20 is the transpose of the columnround function: it converts a 16-word array

$$x = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix}$$

into a 16-word array

$$y = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$$

by four applications of the quarterround function to permutations of the *columns*.

An-Ping claims, as above, that $d(\text{topleft}(x)) \oplus d(\text{topleft}(y))$ is the xor of various $\ell$ values. It is reasonably clear, as above, that this claim was obtained from a "048 formula," a claimed "159 formula," and a claimed "2610 formula," Presumably each claim is incorrect half the time.

## Understanding all rounds

An-Ping claims that $b_1(z)$, for each Salsa20 output block $z$, is the xor of $\ell(a, b)$ for various intermediate words $a, b$ in the Salsa20 state. An-Ping obtains this claim by combining the above claims regarding $d(\text{topleft}(x)) \oplus d(\text{topleft}(y))$ and $d(\text{topleft}(y)) \oplus d(\text{topleft}(z))$, continuing through all rounds of Salsa20, and then considering the addition of the initial and final Salsa20 arrays. This claim involves $\ell(a, b)$ for 256 pairs $(a, b)$: 12 for each round (4 for each quarter-round times 3 relevant quarter-rounds inside each round) and an extra 16 for the final addition.

I tried Salsa20 for 1000 inputs and found, in 503 cases, that $b_1(z)$ failed to match this xor. In short, An-Ping's formula for $b_1(z)$ is wrong—not just occasionally but a whopping 50% of the time.

An-Ping appears to state at one point, without explanation, that there are 336 pairs $(a, b)$. This is more than the 256 pairs noted above; 336 pairs means *every* pair of words added during the Salsa20 computation, forcing An-Ping to consider, e.g., $y_{12} \oplus y_{13} \oplus y_{14}$. Presumably the discrepancy between 256 and 336 is simply the result of An-Ping's general sloppiness; but, for thoroughness, I checked $b_1(z)$ against the xor of all 336 values, and found that it failed to match in 476 out of 1000 cases.

I'm not trying to suggest that, as a general matter, cipher designers are obliged to perform calculations disproving unjustified cryptanalytic claims. It was Li An-Ping's responsibility to split the "analysis" into a series of testable claims, and to systematically check each claim with the help of a computer. The absence of computer verification would have been enough reason to reject the "analysis" even if nobody had pointed out any of An-Ping's errors.

## Can the formulas be repaired?

Suppose that one throws away An-Ping's incorrect 456 formula, 8910 formula, etc., and instead starts from the 012 formula, the 567 formula, etc. Can one combine the correct formulas into an expression for $b_1(z)$ as a xor of $\ell$ values?

I don't see any way to do it. Each row-round formula involves some $y_i$ that isn't cancelled by any column-round formula. One can, for example, write $d(y_0) \oplus d(y_1) \oplus d(y_2) \oplus d(z_0) \oplus d(z_1) \oplus d(z_2)$ as a xor of $\ell$ values, and one can write $d(x_5) \oplus d(x_9) \oplus d(x_{13}) \oplus d(y_5) \oplus d(y_9) \oplus d(y_{13})$ as a xor of $\ell$ values, but xor'ing these formulas leaves $d(y_1)$ and $d(y_9)$ dangling, and I see no other formulas that could be used to cancel $d(y_1)$ and $d(y_9)$.

Of course, one can consider formulas involving quantities other than $\ell$ values. An-Ping attempts to write $b_1(z)$ as a xor of $\ell$ values solely because "all or most" of the $\ell$ values are noticeably biased. One can trivially achieve "most" by simply including many constants in the xor; each constant is quite biased, having deviation 0.5 from 0.5. One can achieve "all" almost as trivially, by writing each quantity as the xor of two biased quantities. An-Ping's failure to do this correctly for Salsa20 does not mean that the task is even marginally difficult; one can easily write down such formulas for any cipher. Fortunately, as explained in Section 5, such formulas say nothing about cipher security.