Salsa20 speed

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

This document discusses a range of benchmarks relevant to cryptographic speed; estimates Salsa20's performance on those benchmarks; and explains, at a lower level, techniques to achieve this performance.

The bottom line is that Salsa20 provides *consistent* high speed in a wide variety of applications across a wide variety of platforms. Consistency means that, in *each* of these contexts, Salsa20 is not far from the fastest cryptographic function that I have seen at the same conjectured security level.

2 Review of the Salsa20 structure

Encryption of a 64-byte block is xor with the output of the Salsa20 hash function, where the input consists of the 32-byte Salsa20 key, the 8-byte nonce (unique message number), the 8-byte block counter, and 16 constant bytes. The reader is cautioned that encryption time is slightly longer than hashing time: in particular, a 64-byte xor is not free.

The Salsa20 hash function regards its 64-byte input x as an array of 16 words in little-endian form. It performs 320 invertible modifications, where each modification changes one word of the array. The resulting words are added to the original words, producing, in little-endian form, the 64-byte output Salsa20(x).

Each modification involves xoring into one word a rotated version of the sum of two other words. Thus the 320 modifications involve, overall, 320 additions, 320 rotations, and 320 xors. The rotations are all by constant distances.

The entire series of modifications is a series of 10 identical double-rounds. Each double-round is a series of 2 rounds. Each round is a set of 4 parallel quarter-rounds. Each quarter-round is a series of 4 word modifications.

3 Benchmarks

One often sees a stream cipher described as taking, e.g., "7 clock cycles per byte on a Pentium II." But this isolated figure tells only a small part of the story.

* The author was supported by the National Science Foundation under grant CCR–9983950, and by the Alfred P. Sloan Foundation. Date of this document: 2005.04.27.

Athlon; Pentium; PowerPC; etc.

The Pentium II is not the only popular general-purpose CPU. For example, in the last few months I've used an AMD Athlon, an AMD Athlon 64, an IBM PowerPC RS64 IV (Sstar), an Intel Pentium III, an Intel Pentium 4 f12 (Willamette), an Intel Pentium 4 f29 (Northwood), an Intel Pentium M, a Motorola PowerPC 7410 (G4), a Sun UltraSPARC II, and a Sun UltraSPARC III.

Some functions have cycle counts that stay fairly consistent from one CPU to another. Other functions don't. There are many examples of cryptographic functions that are specially designed to run well on a particular CPU (using the Pentium 4's complicated `pmuludq` instruction, for example) and that turn out to be much slower on other CPUs.

Salsa20 performs well on all of these CPUs, and I anticipate that it will also have good performance on tomorrow's popular general-purpose CPUs. Its basic operations—addition modulo 2^{32}, constant-distance 32-bit rotation, and 32-bit xor—are consistently fast. It has enough implementation flexibility—in particular, enough internal parallelism—that it is not crippled by the latency problems of some CPUs.

Specifically, the Salsa20 hash function takes between 8 and 17 cycles/byte on a wide range of CPUs. See Sections 4 through 11 for detailed comments on several specific CPUs, including timings of my published Salsa20 software.

General-purpose CPUs; embedded CPUs; FPGAs; ASICs

The large CPUs in laptop computers, desktop computers, and servers are not the only target platforms for cryptography. Cryptographic functions are sometimes implemented on tiny CPUs with 8-bit arithmetic operations and (e.g.) 128 bytes of memory; or on small portions of field-programmable gate arrays; or as small application-specific integrated circuits.

Some functions scale reasonably to all of these contexts: they maintain good price-performance ratios on tiny CPUs and inside FPGAs, and they achieve even better performance in application-specific circuits. Other functions don't. There are many examples of cryptographic functions that rely heavily on, e.g., random access to multi-kilobyte tables, and that slow down drastically on smaller circuits.

Salsa20 scales reasonably to all of these contexts. Its operations decompose easily into 8-bit operations on tiny CPUs, into 4-bit operations in typical gate arrays, and into 1-bit operations in dedicated circuits. It fits comfortably into 128 bytes of memory.

Long messages; short messages

A typical "cycles per byte" benchmark assumes that messages are long—e.g., 8192 bytes. But, in most networking protocols, such long messages never occur. The average Internet packet is only a few hundred bytes long. The average packet in some applications is even smaller.

Some functions perform well across a wide range of message lengths. Other functions don't. There are many examples of cryptographic functions that have very small "cycles per byte" figures but that slow down dramatically for small messages. For example, one function was advertised as taking 1 cycle per byte, with the fine print revealing a setup cost of 2000 cycles per message; this function is not competitive with a function that takes 3 cycles per byte with much lower per-message overhead.

The only Salsa20 per-message setup costs are creating a nonce and initializing a block counter to 0. In other words, Salsa20 offers excellent **message agility**.

This feature of Salsa20 is shared by any block cipher in counter mode, but it is certainly not shared by all stream ciphers.

Convenient message lengths; inconvenient message lengths

A typical benchmark is limited to convenient message lengths: e.g., multiples of 256 bytes. But messages of these convenient lengths are not the only types of messages encrypted. A thorough benchmark suite includes 0-byte packets, 1-byte packets, 2-byte packets, 3-byte packets, and so on through 8192-byte packets.

(Do I know any Internet applications that encrypt 1-byte packets? No. But it's better for a benchmark to provide too much information than too little. The reader can disregard packet sizes that he doesn't care about.)

For most functions, the convenient lengths are exactly the multiples of a function-specific "block size." Handling a message of an inconvenient length takes *roughly* the same amount of time as handling a message of the next convenient length.

The Salsa20 block size is 64 bytes.

Aligned messages; unaligned messages

Most benchmarks assume that messages begin at aligned addresses in memory: e.g., multiples of 16 bytes. But this assumption forces every application layer to either (1) take the time to align messages or (2) impose alignment constraints upon higher layers. There are free-form network protocols that allow encrypted messages to begin at arbitrary byte positions within a network packet; unless and until those protocols are eliminated, *someone* will have to deal with unaligned messages.

The penalty for handling unaligned messages can be quite noticeable. For example, in the SPARC instruction set, a single load instruction can load 8 bytes of plaintext, and a single store instruction can store 8 bytes of ciphertext—*if* the plaintext and ciphertext are required to be aligned. Code to handle arbitrary alignments needs 8 separate byte-load instructions and 8 separate byte-store instructions. The extra instructions take time even if the plaintext is actually aligned. Branching to separate code for aligned and unaligned plaintext costs code space, which creates other speed problems, as discussed below.

Consequently, one must distinguish three types of benchmarks: encryption of aligned data by code that cannot handle unaligned data; encryption of aligned

data by code that can handle unaligned data; and encryption of unaligned data. These distinctions have little to do with the choice of cryptographic function, but they make a sufficiently large difference in timings that they must be made explicit.

One key; many keys

Most cryptographic benchmarks assume that a single key is used for a long time. For example, on large CPUs, the benchmarks assume that any key-dependent tables have been precomputed and loaded into L1 cache, along with the key itself. But there are many applications that bounce rapidly between keys. A thorough benchmark suite includes bouncing among n keys for various values n: e.g., $n \in \{1, 2, 4, 8, \ldots, 1048576\}$.

The penalty for a table is not merely the "key setup time," i.e., the time to precompute the table, amortized over all uses of the key. It is also the time to load the table from L2 cache or from DRAM. This cache-miss time—which depends on the table size, the pattern of table-entry access, etc.—is incurred whenever many other keys have been active recently.

There is no precomputation in Salsa20. In particular, there are no key-dependent tables: 32-byte Salsa20 keys are stored in exactly 32 bytes. In other words, Salsa20 offers excellent **key agility**. There is no point in a "key-setup" function for Salsa20.

(In theory, small changes in (k, n) allow a small percentage of the $\text{Salsa20}_k(n)$ operations to be eliminated at the expense of storage. I do not expect any such tradeoffs to be used in practice. My cost estimates assume that these tradeoffs are not used.)

One message; many messages

Most encryption benchmarks assume that one message is encrypted at a time: an application provides a message as a single packet, which is encrypted in a single burst. But, at a lower level, bytes near the beginning of a message are typically created before bytes near the end. One can reduce latency on parallel platforms by encrypting each message block before subsequent blocks have arrived.

When *several* messages are simultaneously built from smaller components, each message has its own cipher state. A larger state means that the available hardware (e.g., L1 cache) can handle fewer simultaneous messages.

One must distinguish two different measurements of state size. In applications where many messages use the same key, what matters is the "fixed-key state size": the number of state bytes used for an additional message with the same key. In applications where each message uses a different key, what matters is the "variable-key state size": the number of state bytes used for an additional message with another key.

Salsa20 has minimal state size in both situations. Its variable-key state size is just 48 bytes: 32 bytes for the key, 8 bytes for a nonce, and 8 bytes for a block

counter. Its fixed-key state size is just 16 bytes: 8 bytes for the nonce and 8 bytes for the block counter. The state can usually be compressed further: for example, in many applications, messages are below 16384 bytes, so the block counter can be compressed to 1 byte.

This feature of Salsa20, like the minimal per-message setup costs, is shared by any block cipher in counter mode, but it is certainly not shared by all stream ciphers. Many stream ciphers have fixed-key state sizes of 64 bytes or more.

Sequential positions; parallel positions

Most encryption benchmarks focus on sequential platforms handling the blocks of a message serially: one encrypts the first block of a message, then the second block, etc. But parallel platforms can often drastically reduce latency by handling several blocks in parallel—*if* the cipher does not serialize blocks.

Salsa20 has minimal serialization of blocks. Each message block is encrypted independently, the only block-specific input being the block counter.

This is another feature shared by any block cipher in counter mode but certainly not shared by all stream ciphers. Many stream ciphers cannot take advantage of parallel platforms to reduce long-message latency.

Cryptography in isolation; cryptography in context

Most large-CPU encryption benchmarks assume that the CPU is doing nothing other than encryption. But encryption is merely one part of a network protocol. A CPU typically cycles between generating a packet, encrypting the packet, transmitting the packet, etc.; each of these stages involves quite a bit of code.

There can be dramatic slowdowns—tens of thousands of cycles for every packet—when this code does not fit into cache. Avoiding this slowdown means more than fitting the *encryption* code into cache; it means fitting the *generation and encryption and transmission* code, and all other active code, into cache. A thorough benchmark suite includes various levels of competition for cache space.

Salsa20 performs well even with tiny code sizes. One does not need to fully unroll the 20 rounds of the Salsa20 hash function to achieve good performance. The maximum unrolling in my published Salsa20 software is 4 rounds. Even a 20-iteration 1-round loop is reasonably fast. Code for one round typically takes about 200 bytes, depending on the CPU; overhead for the hash function is typically around 300 bytes; overhead for encryption depends on the complexity of the encryption API but is generally small.

Encryption; decryption

Encryption time is only half of the picture. Most applications need to encrypt *and decrypt* data.

Many "non-additive" stream ciphers, such as a block cipher in CBC mode, incur extra costs in these applications. The encryption-and-decryption code is

often twice as large as the encryption code. A key-dependent encryption-and-decryption table is often twice as large as a key-dependent encryption table. This extra space often turns into extra time, as discussed above.

For Salsa20, encryption and decryption are identical. This feature is shared by any block cipher in counter mode, and by many other stream ciphers.

4 Salsa20 on the AMD Athlon

My `salsa20_word_pm` software takes 29.25 Athlon cycles for a Salsa20 round, totalling 585 cycles (9.15 cycles/byte) for 20 rounds, totalling 645 cycles (10.08 cycles/byte) for the Salsa20 hash function, timed as 680 cycles with 35 cycles timing overhead. The timings are actually 655 or 656 cycles most of the time but 849 cycles on every eighth call, presumably because of branch mispredictions.

The compiled code occupies 1248 bytes. Its main loop occupies 937 bytes and handles 4 rounds.

Details

The Athlon has 7 usable integer registers, one of which is consumed by a round counter if the Salsa20 code is not completely unrolled. The Athlon is limited to 3 instructions per cycle and 2 memory operations per cycle.

The small number of registers means that each round requires many loads and stores. Loads can be absorbed into load-operate instructions, although they still count against the memory-operation bottleneck.

A quarter-round can be handled with 16 instructions (4 additions, 4 rotations, 4 xors, 4 stores), 12 memory operations (8 loads, 4 stores), and 1 register:

```
x0 = p                  store
p += x12                      load add
p <<<= 7                           rotate
p ^= x4                       load xor
x4 = p                  store
p += x0                       load add
p <<<= 9                           rotate
p ^= x8                       load xor
x8 = p                  store
p += x4                       load add
p <<<= 13                          rotate
p ^= x12                      load xor
x12 = p                 store
p += x8                       load add
p <<<= 18                          rotate
p ^= x0                       load xor
```

A quarter-round can also be handled with 17 instructions (the extra being a load), just 9 memory operations (5 loads, 4 stores), and 2 registers:

```
x5 = s                  store
r = x1                        load
r += s                             add
r <<<= 7                           rotate
r ^= x9                       load xor
x9 = r                  store
s += r                             add
s <<<= 9                           rotate
s ^= x13                      load xor
x13 = s                 store
r += s                             add
r <<<= 13                          rotate
r ^= x1                       load xor
x1 = r                  store
s += r                             add
s <<<= 18                          rotate
s ^= x5                       load xor
```

Performing four quarter-rounds in parallel, half in one way and half in the other, takes 6 registers, 66 instructions (at least 22 cycles), and 42 memory operations (at least 21 cycles). There are also 2 loop-control instructions for every 4 rounds.

The Athlon can't decode 3 instructions per cycle if any instructions cross 16-byte boundaries in memory. My `salsa20_word_pm` software makes no attempt to align instructions; this can easily explain the gap between $\approx$ 22 cycles/round and $\approx$ 29 cycles/round. In fact, my `salsa20_word_pm` software has no Athlon tuning; it was designed for the Pentium M.

I plan to try aligning instructions, to try shifting the balance slightly towards memory operations, and to experiment with XMM code as on the Pentium 4. There are other Athlon bottlenecks that are more difficult to analyze, so I won't speculate as to the achievable number of cycles/round.

As for the 60-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

Comparison to AES timings

Osvik reports that unpublished software—with no protection against timing leaks—takes 225 Athlon cycles (over 14 cycles/byte) to encrypt a 16-byte block with a 16-byte AES key, assuming that the key was pre-expanded into 176 bytes.

One can reasonably extrapolate that similar software would take over 300 Athlon cycles (over 18 cycles/byte) to encrypt a 16-byte block with a 32-byte AES key, assuming that the key was pre-expanded into 240 bytes.

5 Salsa20 on the IBM PowerPC RS64 IV (Sstar)

My `salsa20_word_aix` software takes 33 PowerPC RS64 IV cycles for each Salsa20 round, totalling 660 cycles (10.32 cycles/byte) for 20 rounds, totalling

756 cycles (11.82 cycles/byte) for the Salsa20 hash function, timed as 770 cycles with 14 cycles timing overhead.

The compiled code for the Salsa20 hash function occupies 768 bytes. Its main loop occupies 392 bytes and handles 2 rounds.

Details

The PowerPC RS64 IV has enough registers to avoid all loads and stores inside the hash-function rounds. The 16 words of hash-function input are loaded into separate registers; 4 quarter-rounds are performed in parallel, with 1 temporary register for each quarter-round; after 20 rounds, the input is loaded again, added to the round output, and stored.

The obvious bottleneck is that the PowerPC RS64 IV is limited to 2 integer operations per cycle, with a rotate instruction counting as 2 operations. Each round has 64 operations and therefore takes at least 32 cycles, totalling 640 cycles (10.00 cycles/byte) for 20 rounds, even with fully unrolled code. I'm satisfied with my current 33 cycles/round.

The above comment regarding the cost of rotation is a guess based on limited experiments. As far as I can tell, the only RS64 IV performance documentation is a 14-page introductory article that does not discuss this cost.

My software interleaves four parallel rounds in a reasonably straightforward way; rotations are presumed to have high latency and are scheduled early. Here is the first quarter of the code for four parallel quarter-rounds:

```
y4 = x0 + x12
y4 <<<= 7
                y9 = x5 + x1
                y9 <<<= 7
                                y14 = x10 + x6
                                y14 <<<= 7
                                                y3 = x15 + x11
                                                y3 <<<= 7
x4 ^= y4
                x9 ^= y9
                                x14 ^= y14
                                                x3 ^= y3
```

Simpler C code fed through `gcc -O3 -mcpu=power` takes 36.5 cycles per round.

As for the 96-cycles-per-hash overhead: One can easily eliminate most of this overhead by merging the Salsa20 hash function with a higher-level encryption function. The PowerPC function-call overhead is made particularly severe by the large number of PowerPC callee-save registers.

6 Salsa20 on the Intel Pentium III

My `salsa20_word_pii` software takes 37.5 Pentium III cycles for each Salsa20 round, totalling 750 cycles (11.72 cycles/byte) for 20 rounds, totalling 837 cycles

(13.08 cycles/byte) for the Salsa20 hash function, timed as 872 cycles with 35 cycles timing overhead. (The timings are actually 859 cycles most of the time but 908 cycles on every fourth call, presumably because of branch mispredictions.)

The compiled code for the Salsa20 hash function occupies 1280 bytes. Its main loop occupies 937 bytes and handles 4 rounds.

Details

The Pentium III has 7 usable integer registers, one of which is consumed by a round counter if the Salsa20 code is not completely unrolled. The small number of registers means that each round requires many loads and stores.

The Pentium III is limited to 3 operations per cycle. A store instruction counts as 2 operations. A load-operate instruction counts as 2 operations. The Pentium III is also limited to 2 integer operations per cycle.

A store to the stack, and a subsequent load from the stack, can be replaced with a store to "MMX registers," and a subsequent load from MMX registers. The MMX store counts for only 1 operation, unlike a stack store. On the other hand, the MMX load and the MMX store both count as integer operations, unlike a stack load and a stack store.

My software handles each quarter-round with 4 loads, 4 stores, 4 additions, 4 rotations, and 4 xors:

```
a = x12
b = x0
c = x4
e = a + b
e <<<= 7
c ^= e
x4 = c
d = x8
e = b + c
e <<<= 9
d ^= e
x8 = d
c += d
c <<<= 13
a ^= c
x12 = a
a += d
a <<<= 18
b ^= a
x0 = b
```

The number of registers used by this code ranges up to 5 but is often smaller, allowing some overlap between quarter-rounds. The Pentium III can rearrange quite a few instructions internally to improve parallelism. On the other hand,

Intel has failed to document the rules by which the Pentium III chooses which operations to perform next, so it is difficult to predict exact timings except by experiment.

With six MMX registers, operations are balanced with integer operations: each double-round has 120 integer operations, taking at least 30 cycles/round, and 180 operations, taking at least 30 cycles/round. After some experiments I ended up putting `x1`, `x2`, `x6`, `x7`, `x11`, and `x12` into MMX registers.

Beware that there are many other interesting Pentium III bottlenecks; code not tuned for the Pentium III is unlikely to achieve good performance. For example, `salsa20_word_pm` takes nearly 60 Pentium III cycles/round.

Perhaps 37.5 cycles/round can be improved. However, blind experiments are quite painful for the programmer, so I don't plan to put further effort into the Pentium III until I have a better Pentium III simulator.

As for the 87-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

Comparison to AES timings

Osvik reports that unpublished software—with no protection against timing leaks—takes 224 Pentium III cycles (14 cycles/byte) to encrypt a 16-byte block with a 16-byte AES key, assuming that the key was pre-expanded into 176 bytes.

One can reasonably extrapolate that similar software would take over 300 Pentium III cycles (over 18 cycles/byte) to encrypt a 16-byte block with a 32-byte AES key, assuming that the key was pre-expanded into 240 bytes.

7 Salsa20 on the Intel Pentium 4 f12 (Willamette)

My `salsa20_word_p4` software takes 48 Pentium 4 f12 (Willamette) cycles for each Salsa20 round, totalling 960 cycles (15 cycles/byte) for 20 rounds, totalling 1052 cycles (16.44 cycles/byte) for the Salsa20 hash function, timed as 1136 cycles with 84 cycles timing overhead.

The compiled code for the Salsa20 hash function occupies 1144 bytes. Its main loop occupies 629 bytes and handles 4 rounds.

Details

The Pentium 4 does badly with `salsa20_word_pii`: it has a high latency for moving data between the 32-bit integer registers and the 64-bit MMX registers. The Pentium 4 f12 does better with `salsa20_word_pm`, but other Pentium 4 CPUs have a high latency for reading data that was recently written to memory. So `salsa20_word_p4` takes a completely different approach.

The Pentium 4 has eight "XMM registers," each of which can hold four 32-bit integers. The Pentium 4 has several XMM instructions:

- Add an XMM register (r_0, r_1, r_2, r_3) into an XMM register (s_0, s_1, s_2, s_3): i.e., replace (s_0, s_1, s_2, s_3) with $(s_0 + r_0, s_1 + r_1, s_2 + r_2, s_3 + r_3)$. This has 2-cycle latency.
- Xor an XMM register into an XMM register: i.e., replace (s_0, s_1, s_2, s_3) with $(s_0 \oplus r_0, s_1 \oplus r_1, s_2 \oplus r_2, s_3 \oplus r_3)$. For example, set a register to 0. This has 2-cycle latency.
- Shift an XMM register (r_0, r_1, r_2, r_3) left or right by some number of bits: for example, replace (r_0, r_1, r_2, r_3) with $(r_0 \ll 7, r_1 \ll 7, r_2 \ll 7, r_3 \ll 7)$. This has 2-cycle latency.
- Shuffle an XMM register (r_0, r_1, r_2, r_3) into an XMM register (s_0, s_1, s_2, s_3). For example, a 32-bit left rotation replaces (s_0, s_1, s_2, s_3) with (r_3, r_0, r_1, r_2). (Recall that "left" really means "towards more significant bits," which is towards the right on a little-endian computer.) This has 4-cycle latency.

The Pentium 4 cannot perform two of these operations on the same cycle; it cannot perform two arithmetic operations (add, xor) on adjacent cycles; it cannot perform two shift operations (shift, shuffle) on adjacent cycles.

At the beginning of a column round, `salsa20_word_p4` stores the input $(x_0, x_1, \ldots, x_{15})$ in four XMM registers: `diag0` has $(x_0, x_5, x_{10}, x_{15})$, `diag1` has $(x_{12}, x_1, x_6, x_{11})$, `diag2` has (x_8, x_{13}, x_2, x_7), and `diag3` has (x_4, x_9, x_{14}, x_3). It performs a column round with the following instructions, which just barely fit into 8 registers:

```
# Two instructions inserted into gaps in previous round:
a0 = diag1 <<< 0
b0 = 0
# Main loop begins here, with more than two gaps:
a0 += diag0
                a1 = diag0 <<< 0
b0 += a0
a0 <<= 7
                b1 = 0
b0 >>= 25
diag3 ^= a0

diag3 ^= b0

                a1 += diag3
                               a2 = diag3 <<< 0
                b1 += a1
                a1 <<= 9
                               b2 = 0
                b1 >>= 23
                diag2 ^= a1
diag3 <<<= 32
                diag2 ^= b1
```

```
                        a2 += diag2
                                        a3 = diag2 <<< 0
                        b2 += a2
                        a2 <<= 13
                                        b3 = 0
                        b2 >>= 19
                        diag1 ^= a2
diag2 <<<= 64
                        diag1 ^= b2

                                        a3 += diag1

                                        b3 += a3
                                        a3 <<= 18

                                        b3 >>= 14
                                        diag0 ^= a3
                        diag1 <<<= 96
                                        diag0 ^= b3
```

At this point `diag0` has $(y_0, y_5, y_{10}, y_{15})$, `diag1` has $(y_1, y_6, y_{11}, y_{12})$, `diag2` has (y_2, y_7, y_8, y_{13}), and `diag3` has (y_3, y_4, y_9, y_{14}). Essentially the same code then performs a row round and shifts the indices back to the original positions.

Each round has 20 arithmetic instructions and 15 shift instructions, and therefore takes at least 40 cycles. It's possible to eliminate some instructions, but I don't see any way to decrease the theoretical latency below 40 cycles. I don't know why the actual performance is 48 cycles/round.

As for the 92-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

Comparison to AES timings

Osvik reports that unpublished software—with no protection against timing leaks—takes 260 Pentium 4 (f12?) cycles (16.25 cycles/byte) to encrypt a 16-byte block with a 16-byte AES key, assuming that the key was pre-expanded into 176 bytes. Matsui and Fukuda report that unpublished software—with no protection against timing leaks—takes 251 Pentium 4 (f29?) cycles (15.68 cycles/byte) and 284 Pentium 4 f33 cycles (17.75 cycles/byte).

One can reasonably extrapolate that similar software would take over 340 Pentium 4 f12 cycles (over 21 cycles/byte) to encrypt a 16-byte block with a 32-byte AES key, assuming that the key was pre-expanded into 240 bytes.

8 Salsa20 on the Intel Pentium M

My `salsa20_word_pm` software takes 33.75 Pentium M cycles for each Salsa20 round, totalling 675 cycles (10.55 cycles/byte) for 20 rounds, totalling 740 cycles

(11.57 cycles/byte) for the Salsa20 hash function, timed as 790 cycles with 50 cycles timing overhead. (The timings are actually 780 or 781 cycles most of the time but 856 cycles on every eighth call, presumably because of branch mispredictions.)

The compiled code for the Salsa20 hash function occupies 1248 bytes. Its main loop occupies 937 bytes and handles 4 rounds.

Details

The Pentium M has 7 usable integer registers, one of which is consumed by a round counter if the Salsa20 code is not completely unrolled. The small number of registers means that each round requires many loads and stores.

The Pentium M is limited to 3 operations per cycle. Like the Pentium III, the Pentium M counts a load-operate instruction as 2 operations. Unlike the Pentium III, the Pentium M counts a store instruction as 1 operation. This difference means that `salsa20_word_pii` is slightly faster on the Pentium M than on the Pentium III, taking only about 36 cycles/round; it also means that a quite different sequence of instructions produces better results.

See Section 4 for details of how `salsa20_word_pm` handles quarter-rounds. A round ends up using 90 operations (taking at least 30 cycles): 16 additions, 16 rotations, 16 xors, 16 stores, and 26 loads.

Perhaps 33.75 cycles/round can be improved. As in Section 6, I don't plan to carry out extensive blind experiments. I do plan to experiment with XMM code as on the Pentium 4.

As for the 65-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

Comparison to AES timings

The Pentium M might compute AES in marginally less time than the Pentium III, but both CPUs face the same basic AES bottleneck: encrypting a 16-byte block with a 16-byte AES key requires 200 S-box lookups, which cannot take fewer than 200 cycles (12.5 cycles/byte). Similarly, encrypting a 16-byte block with a 32-byte AES key requires 280 S-box lookups, which cannot take fewer than 280 cycles (17.5 cycles/byte). Even more S-box lookups are required if keys are not pre-expanded.

9 Salsa20 on the Motorola PowerPC 7410 (G4)

My `salsa20_word_macos` software takes 24.5 PowerPC 7410 cycles for each Salsa20 round, totalling 490 cycles (7.66 cycles/byte) for 20 rounds, totalling approximately 570 cycles (8.91 cycles/byte) for the Salsa20 hash function, timed as approximately 584 cycles with 14 cycles timing overhead. (Precise timings are difficult: the CPU's cycle counter has 16-cycle resolution.)

The compiled code for the Salsa20 hash function occupies 768 bytes. Its main loop occupies 392 bytes and handles 2 rounds.

Details

The PowerPC 7410, like the PowerPC RS64 IV, has enough registers to avoid all loads and stores inside the hash-function rounds.

The obvious bottleneck is that the PowerPC 7410 is limited to 2 integer operations per cycle. The PowerPC 7410, unlike the PowerPC RS64 IV, counts a rotate instruction as 1 operation.

Each round has 48 operations and therefore takes at least 24 cycles, totalling 480 cycles (7.50 cycles/byte) for 20 rounds, even with fully unrolled code. I'm satisfied with my current 24.5 cycles/round.

Here is the first quarter of the code for four parallel quarter-rounds:

```
y4 = x0 + x12
                y9 = x5 + x1
y4 <<<= 7
                                y14 = x10 + x6
                y9 <<<= 7
                                                y3 = x15 + x11
                                y14 <<<= 7
x4 ^= y4
                                                y3 <<<= 7
                x9 ^= y9
                                x14 ^= y14
                                                x3 ^= y3
```

As for the 80-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

Comparison to AES timings

Lipmaa reports that AES software by Ahrens—with, presumably, no protection against timing leaks—takes 401 PowerPC 7400 cycles (over 25 cycles/byte) to encrypt a 16-byte block with a 16-byte AES key, assuming that the key was pre-expanded into 176 bytes. I am not aware of any relevant differences between the PowerPC 7400 and the PowerPC 7410.

It should be possible to do somewhat better—my own public-domain AES software, *including* key expansion, takes about 490 cycles on the PowerPC 7410—but AES is clearly much slower than Salsa20 on this CPU.

10 Salsa20 on the Sun UltraSPARC II

My `salsa20_word_sparc` software takes 40.5 UltraSPARC II cycles for each Salsa20 round, totalling 810 cycles (12.66 cycles/byte) for 20 rounds, totalling

881 cycles (13.77 cycles/byte) for the Salsa20 hash function, timed as 892 cycles with 11 cycles timing overhead.

The compiled code for the Salsa20 hash function occupies 936 bytes. Its main loop occupies 652 bytes and handles 2 rounds.

Details

The UltraSPARC II handles each rotation with 3 integer operations: shift, shift, add. It is limited to 2 integer operations per cycle, and to 1 shift per cycle. Like the PowerPC, it has enough registers to avoid all loads and stores inside the hash-function rounds.

A round has 80 integer operations—32 adds, 32 shifts, 16 xors—and therefore takes at least 40 cycles. I'm satisfied with my current 40.5 cycles/round.

As for the 71-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

Comparison to AES timings

Lipmaa reports that unpublished software—with, presumably, no protection against timing leaks—takes 270 UltraSPARC II cycles (over 16 cycles/byte) to encrypt a 16-byte block with a 16-byte AES key, assuming that the key was pre-expanded into 176 bytes.

One can reasonably extrapolate that similar software would take over 370 UltraSPARC II cycles (over 23 cycles/byte) to encrypt a 16-byte block with a 32-byte AES key, assuming that the key was pre-expanded into 240 bytes.

11 Salsa20 on the Sun UltraSPARC III

My `salsa20_word_sparc` software takes 41 UltraSPARC III cycles for each Salsa20 round, totalling 820 cycles (12.82 cycles/byte) for 20 rounds, totalling 889 cycles (13.90 cycles/byte) for the Salsa20 hash function, timed as 905 cycles with 16 cycles timing overhead.

The compiled code for the Salsa20 hash function occupies 936 bytes. Its main loop occupies 652 bytes and handles 2 rounds.

Details

The UltraSPARC III is very similar to the UltraSPARC II. The UltraSPARC III documentation reports a few minor advantages that are not helpful for the Salsa20 computation: e.g., both integer operations in a cycle can be shifts. The disadvantages of the UltraSPARC III are not as well documented; I don't know why the UltraSPARC III is taking an extra 0.5 cycles per round. I don't plan to investigate.

Simple C code fed through `gcc -m64 -O3 -mcpu=ultrasparc` achieves 42.5 UltraSPARC III cycles per round. Beware that code using an array `x[16]` is almost twice as slow as code using separate variables `x0`, `x1`, etc.

Simple C code fed through Sun's compiler achieves 41.5 cycles per round, although at the expense of a completely unrolled main loop.

As for the 69-cycles-per-hash overhead: One can easily eliminate most of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

Comparison to AES timings

As far as I know, AES on an UltraSPARC III is at least as slow as AES on an UltraSPARC II.

12 Salsa20 on next-generation CPUs

One can safely expect Salsa20 to perform well on tomorrow's popular CPUs, for the same reason that Salsa20 achieves consistent high speed on a wide variety of existing CPUs.

The basic operations in Salsa20—addition modulo 2^{32}, constant-distance 32-bit rotation, and 32-bit xor—are so simple, and so widely used, that they can be safely expected to remain fast on future CPUs. Consider, as an extreme example, the Pentium 4 f12, widely criticized for its "slow" shifts and rotations (fixed by the Pentium 4 f33); this CPU can still perform two 32-bit shifts per cycle using its XMM instructions.

The accompanying communication in Salsa20—the addition, rotation, and xor modify 1 word out of 16 words, using 2 other words—is sufficiently small that it can also be expected to remain fast on future CPUs. Fast Salsa20 code is particularly easy to write if the 16 words, and a few temporary words, fit into registers; but, as illustrated by my Salsa20 implementation for the Pentium M, smaller register sets do not pose a serious problem.

Furthermore, Salsa20 can benefit from a CPU's ability to perform several operations in parallel. For example, the PowerPC 7450 (G4e) documentation indicates that the PowerPC 7450 can perform 3 operations per cycle instead of the 2 performed by the PowerPC 7410; I would not be surprised to see Salsa20 running at 6 cycles/byte on the PowerPC 7450. Latency does not become a bottleneck for Salsa20 unless the CPU's latency/throughput ratio exceeds 4.

One can imagine functions that use even simpler operations, and that have even less communication, and that support even more parallelism. But what really matters is that Salsa20 is simpler, smaller, and more parallel than the *average* computation. It is hard to imagine how a CPU could make Salsa20 perform badly without also hurting a huge number of common computations.

13 Salsa20 on smaller platforms

Building a Salsa20 circuit is straightforward. A 32-bit add-rotate-xor fits into a small amount of combinational logic. Salsa20's 4×4 row-column structure is a natural hardware layout; the regular pattern of operations means that each quarter-round will have similar propagation delays. Of course, the exact speed of a Salsa20 circuit will depend on the amount of hardware devoted to the circuit.

Similarly, the 32-bit operations in the Salsa20 computation can easily be decomposed into common 8-bit operations for a small CPU:

- A 32-bit xor can be decomposed into four 8-bit xors.
- A 32-bit addition can be decomposed into four 8-bit additions (with carry).
- A 32-bit rotation can be decomposed into, e.g., several 8-bit rotate-carry operations. The exact number of rotate-carry operations depends on how close the rotation distance is to a multiple of 8.

An average Salsa20 word modification ends up taking about 20 8-bit arithmetic operations—about 20 cycles on a typical 8-bit CPU. If loads and stores consume another 32 cycles then 20 rounds of the Salsa20 hash function will take about 16640 cycles (260 cycles/byte). Salsa20 has no trouble fitting into the 128 bytes of memory on a typical 8-bit CPU.

For comparison, AES with a 32-byte key reportedly takes 5221 cycles (326 cycles/byte) on the Intel 8051. Of course, a serious speed comparison would require writing Salsa20 software for that CPU; it's clear that Salsa20 won't be much slower than AES on the 8051, but at this point I don't know whether it will actually be faster.