sphlib Update for the SHA-3 Third-Round Candidates

Thomas Pornin, <thomas.pornin@cryptolog.com>

July 20, 2011

Abstract

sphlib[1] is an open-source library of hash function implementations, in both C and Java, including all candidates to the SHA-3 competition[2] which were still considered for the second round. The code aims at portability, not using any special feature which are not available in portable C. All code was rewritten from scratch, by the same developer, with similar efforts at optimization. sphlib was then used to benchmark the performance of the second-round SHA-3 candidates on a variety of platforms, with an emphasis on low-power embedded 32-bit platforms[3]. sphlib has then been updated with the "tweaks" published by the five candidates which have been selected for the third SHA-3 round; this document explores how much the tweaks alter the performance measures on our platforms.

1 Methodology

Each hash function is benchmarked when processing a very long message. The function code is first exercised a few dozen times, so that the various caches are appropriately filled and JIT compilation is triggered (for Java). Then, the function is invoked over a message consisting of a variable number of instances of a single 8 kB buffer; the number of instances is dynamically adjusted so that the total computation time exceeds 2 seconds. The clocking functions provided by the system (e.g. clock() in standard C) then yield an adequate measure. Achieved precision is about 3%.

We use seven target architectures. The hardware and software environments are identical to those which were used for the second-round benchmarks, except that the operating system and C compiler for the x86 platform (both 32-bit and 64-bit) have been updated. The seven targets are:

- A PC with an Intel Core 2 Quad Q6600 processor, clocked at 2.4 GHz ("Kentsfield" core). This is similar to the NIST reference platform[1]. RAM size and operating system are not relevant for our tests (everything important happens in level-1 cache anyway). The system runs in 64-bit mode (known as "x86-64", "AMD64", "x64", "Intel 64", "EM64T" and a few other names). The C compiler is GCC[4], version 4.5.2 (was 4.4.3 for second-round benchmarks).
- The very same PC, but this time used in 32-bit mode. Registers have size 32 bits instead of 64, and there are fewer of them. The C compiler is still GCC-4.5.2.
- An older PowerPC 750 (aka "G3"), clocked at 300 MHz. The processor runs in big-endian mode. The C compiler is GCC 4.1.3.
- A Linksys WRT54GS router, refurbished with the OpenWrt operating system[5] (a Linux clone for embedded systems). The core CPU is a Broadcom BCM3302, a MIPS-compatible 32-bit processor clocked at 200 MHz. The level-1 cache size is 8 kB for code, and another 8 kB for instructions. The C compiler is GCC, version 4.2.4.

[1] Actually, the NIST reference platform is described a bit vaguely, since "Core 2 Duo" covers a wide range of processors from Intel, with distinct timing characteristics.

- A Hewlett-Packard HP-50g scientific calculator. This system uses an ARM920T core (ARMv4 architecture), clocked at 12 MHz, but which can be programmatically boosted to 75 MHz (which we did for our tests). This system can be programmed in C (with a much reduced, non-standard C library), using HPGCC[6], a derivative from GCC 4.1.1.

- Our PC (Intel Core 2 Q6600, 2.4 GHz, 64-bit mode), running the Java VM edited by Sun Microsystems (now Oracle), version 1.6.0_20[7]. Note that newer VM versions are available (up to 1.6.0_26) but with different optimization heuristics, which, in the case of `sphlib`, appear to *decrease* performance, sometimes by 20% (depending on the hash function). In order to make the new measures coherent with those from second round, we use the very same version than previously.

- The same PC, in 32-bit mode, with the 32-bit version of the Java VM from Sun.

For each function, we measure the speed for a 256-bit and for a 512-bit output lengths.

The C code is compiled with options "`-O1 -fomit-frame-pointer`" or "`-O2 -fomit-frame-pointer`". Also, `sphlib` can optionally target architectures with a small L1 cache; the "`-DSPH_SMALL_FOOTPRINT`" command-line flag selects that variant. Moreover, on the ARM processor, we want to benchmark both "normal" code (32-bit opcodes) and "Thumb" code (16-bit simplified opcodes). Thus, we end up measuring speed for four variants (eight on the ARM); we systematically keep the best of all obtained figures.

For the Java code, we use the "`-server`" command-line flag which is supposed to trigger more aggressive optimizations (this is the default mode for the 64-bit JVM).

2 Measures

For each function, we list the speed expressed in megabytes per second, in cycles per processed byte, and the bandwidth relative to SHA-2 (e.g. a relative bandwidth of 2.0 would mean that the function is twice faster than SHA-2 with the same output size).

Function	256-bit output			512-bit output		
	Mbytes/s	**cycles/byte**	**rel. SHA-2**	**Mbytes/s**	**cycles/byte**	**rel. SHA-2**
SHA-2	138.37	17.30	1.000	178.36	13.42	1.000
BLAKE	173.74	13.78	1.256	295.80	8.09	1.658
Grøstl	82.09	29.16	0.593	51.23	46.73	0.287
JH	41.55	57.62	0.300	41.43	57.78	0.232
Keccak	140.91	16.99	1.018	75.62	31.66	0.424
Skein	340.87	7.02	2.463	344.15	6.96	1.930

Table 1: Performance of `sphlib` on x86-64 (Intel Q6600, 64-bit, 2.4 GHz).

Function	256-bit output			512-bit output		
	Mbytes/s	**cycles/byte**	**rel. SHA-2**	**Mbytes/s**	**cycles/byte**	**rel. SHA-2**
SHA-2	126.62	18.91	1.000	45.19	52.98	1.000
BLAKE	138.37	17.30	1.093	37.60	63.67	0.832
Grøstl	30.37	78.83	0.240	22.08	108.42	0.489
JH	17.66	135.56	0.139	17.80	134.49	0.394
Keccak	41.94	57.08	0.331	23.30	102.75	0.516
Skein	58.10	41.20	0.459	57.85	41.38	1.280

Table 2: Performance of `sphlib` on i386 (Intel Q6600, 32-bit, 2.4 GHz).

Function	256-bit output			512-bit output		
	Mbytes/s	**cycles/byte**	**rel. SHA-2**	**Mbytes/s**	**cycles/byte**	**rel. SHA-2**
SHA-2	14.04	21.37	1.000	4.58	65.50	1.000
BLAKE	16.29	18.42	1.160	4.37	68.65	0.954
Grøstl	2.39	125.52	0.170	1.31	229.01	0.286
JH	1.73	173.41	0.123	1.72	174.42	0.376
Keccak	3.24	92.59	0.231	1.75	171.43	0.382
Skein	5.69	52.72	0.405	5.69	52.72	1.242

Table 3: Performance of `sphlib` on G3 (PowerPC 750, 32-bit, 300 MHz).

Function	256-bit output			512-bit output		
	Mbytes/s	**cycles/byte**	**rel. SHA-2**	**Mbytes/s**	**cycles/byte**	**rel. SHA-2**
SHA-2	2.82	70.92	1.000	1.47	136.05	1.000
BLAKE	2.25	88.89	0.798	1.39	143.88	0.946
Grøstl	0.44	454.55	0.156	0.30	666.67	0.204
JH	0.54	370.37	0.191	0.54	370.37	0.367
Keccak	1.04	192.31	0.369	0.56	357.14	0.381
Skein	1.56	128.21	0.553	1.57	127.39	1.068

Table 4: Performance of `sphlib` on Broadcom BCM3302 (MIPS, 32-bit, 200 MHz).

Function	256-bit output			512-bit output		
	Mbytes/s	**cycles/byte**	**rel. SHA-2**	**Mbytes/s**	**cycles/byte**	**rel. SHA-2**
SHA-2	1.74	43.10	1.000	0.55	136.36	1.000
BLAKE	1.39	53.96	0.799	0.68	110.29	1.236
Grøstl	0.24	312.50	0.138	0.12	625.00	0.218
JH	0.19	394.74	0.109	0.19	394.74	0.345
Keccak	0.38	197.37	0.218	0.20	375.00	0.364
Skein	0.58	129.31	0.333	0.58	129.31	1.055

Table 5: Performance of `sphlib` on ARM920T (ARMv4, 32-bit, 75 MHz).

Function	256-bit output			512-bit output		
	Mbytes/s	**cycles/byte**	**rel. SHA-2**	**Mbytes/s**	**cycles/byte**	**rel. SHA-2**
SHA-2	82.88	28.89	1.000	77.31	30.97	1.000
BLAKE	52.12	45.93	0.629	81.34	29.43	1.052
Grøstl	34.33	69.73	0.414	23.54	101.70	0.304
JH	27.17	88.11	0.328	27.19	88.05	0.352
Keccak	56.09	42.68	0.677	30.52	78.44	0.395
Skein	101.03	23.70	1.219	103.32	23.17	1.336

Table 6: Performance of `sphlib` with Java (Intel x86 Q6600, 64-bit, 2.4 GHz).

Function	256-bit output			512-bit output		
	Mbytes/s	**cycles/byte**	**rel. SHA-2**	**Mbytes/s**	**cycles/byte**	**rel. SHA-2**
SHA-2	68.23	35.09	1.000	26.40	90.68	1.000
BLAKE	46.78	51.18	0.686	38.26	62.57	1.449
Grøstl	17.57	136.25	0.258	11.84	202.20	0.448
JH	9.02	265.41	0.132	9.00	266.00	0.341
Keccak	18.40	130.11	0.270	10.03	238.68	0.380
Skein	38.28	62.54	0.561	38.99	61.40	1.477

Table 7: Performance of `sphlib` with Java (Intel x86 Q6600, 32-bit, 2.4 GHz).

3 Comments

3.1 On functions

The third-round tweak on BLAKE consists in additional rounds, mechanically implying a corresponding slowdown; it also enlarges code for implementations which unroll the main loop, but this did not cause the code size to cross any cache size threshold on our target systems. The expected extra cost for BLAKE is about +40% for the "small" version (for 224-bit and 256-bit outputs), and +14% for the "big" version (for 384-bit and 512-bit outputs).

The same applies to JH, which increased its round number from 35.5 to 42, hence a +18% cost.

The Grøstl tweaks include new round constants, which impact more internal state bytes than the previous constants. This slightly increases the cost of adding those constants. The other tweak is the change in the shift values in P and Q which happens to increase code size, because now Q no longer uses the same shift values than P. This prevents code sharing between P and Q; this does not impact unrolled versions of Grøstl, but it does slow down a bit more the Java version.

The Keccak tweak alters only the padding rule, which does not register at all in a "long message" benchmark like the one we perform. The `sphlib-3.0` implementation for Keccak is actually faster than the previous code, because of an optimization that I had missed, and which the C compiler did not optimize either. Namely, the Grøstl specification describes the θ transform as the sum (bitwise XOR) of five rotated 64-bit values; the sum and rotation commute, and thus we should do the rotation on the sum only, not on the source values. The speedup, compared to second-round benchmarks, depends on the architecture but is about +50% on the ARM920T (the slow performance of `sphlib-2.1` for Keccak on some ARM systems had been reported by the Keccak authors; this should now be corrected).

Skein is generically defined as three functions, depending on the internal state size (Skein-256, Skein-512 and Skein-1024); the submission for the second round used Skein-256 for the 224-bit and 256-bit hash output sizes, and Skein-512 for the larger output sizes. `sphlib-2.1` implemented these choices, in order to be compatible with the published test vectors. During the second SHA-3 conference, it was revealed that the real Skein submission called for Skein-512 for all output sizes, and this is confirmed by the test vectors included in the third round submission package. `sphlib-3.0` aligns with those test vectors. The net consequence is that the performance of Skein with a 256-bit output size is now equal to the performance with a 512-bit output size. The other Skein tweak (a new 64-bit constant) has no influence whatsoever on measured performance.

3.2 On performance relatively to SHA-2

The salient fact shown by our measures is that on the "small" 32-bit platforms, the SHA-3 finalists are not substantially faster than SHA-2, and often much slower. In particular, on the Broadcom MIPS and the ARM920T, SHA-256 is faster than all SHA-3 finalists, whether they use a 256-bit or a 512-bit output, so even selecting a 512-bit output and truncating it will not make the future SHA-3 as fast as SHA-256 on those platforms, when a 256-bit hash value is needed. The situation is most extreme on the ARM920T, where BLAKE-256 achieves 80% of the speed of SHA-256, while all other functions are at least three times slower than SHA-256.

For a 512-bit output size, the SHA-3 candidates look a bit better, relatively speaking, but only because SHA-512 is quite slow on small 32-bit systems. Still, the fastest SHA-3 finalist with a 512-bit output is no more than 24% faster than SHA-512 on the ARM920T, and only 7% on the MIPS.

Situation is only slightly better on the PowerPC, even though its larger L1 cache (32 kB) favours "complex" functions (through unrolling). Only BLAKE is faster than SHA-256, and by only 16%; the second best function being Skein which does not even achieve half the throughput of SHA-256. For a 512-bit output, only BLAKE and Skein are competitive with SHA-512, and not much faster.

A similar situation is encountered with the 32-bit Java platform. This is *the* most common combination for applets, since even the 64-bit Windows 7 operating system ships with a 32-bit Web browser, which is used by default; the JVM for applets running in the browser address space, it is also in 32-bit mode. There again, SHA-256 outperforms all SHA-3 finalists; only for a 512-bit output do some finalists (BLAKE and Skein again) offer a speed improvement over the slow SHA-512, and, then again, that improvement is less than +50%.

Even with a 64-bit Java platform, SHA-2 is not overly outclassed; at most, Skein-512 beats SHA-512 by a meager +34%.

Only on the x86 architecture, with C code, do some of the finalists really beat SHA-2; this is also a platform which offers SIMD instructions which most functions can use one way or another. Such SIMD instructions are inexistent on our PowerPC, ARM and MIPS targets, and unavailable in the Java platforms; thus, the portable code of `sphlib` should offer near-optimal performance on these. The conclusion is that the SHA-3 finalists are *intrinsically* not substantially faster than SHA-2 on 32-bit architectures which do not have SIMD instructions.

3.3 On the use of SHA-3

The future role of SHA-3 has not been defined unambiguously; NIST has announced that they do not plan to deprecate SHA-2 any time soon.

If SHA-3 is meant to be a spare function, to be used as a replacement of SHA-2 should SHA-2 be broken, then the best candidate for that role is BLAKE: it is the closest to SHA-2, in terms of performance, on almost all architectures. An application using SHA-2 is likely to be equally (un)happy with BLAKE, on the same hardware.

If raw performance on systems with 64-bit integer types is the most desired property, then Skein is the way to go. However, performance suffers on 32-bit systems.

These comments are only about performance; they assume that all finalists are equally secure.

References

[1] *sphlib 3.0*, `http://www.saphir2.com/sphlib/`

[2] *Cryptographic Algorithm Hash Competition*, `http://csrc.nist.gov/groups/ST/hash/sha-3/`

[3] T. Pornin, *Comparative Performance Review of the SHA-3 Second-Round Candidates*, presented at the Second SHA-3 Candidate Conference, August 23-24, 2010.

[4] *GCC, the GNU Compiler Collection*, `http://gcc.gnu.org/`

[5] *OpenWrt*, `http://openwrt.org/`

[6] *HPGCC*, `http://sourceforge.net/projects/hpgcc/`

[7] *Java SE Downloads*, `http://www.oracle.com/technetwork/java/javase/downloads/index.html`