Comparison of seven SHA-3 candidates software implementations on smart cards.

Mourad Gouicem*
Oberthur Technologies
Contact : {g.piret, e.prouff}@oberthur.com

October 2010

Abstract

In this work, we present and compare seven SHA-3 second-round candidates implementations on two different architectures used on smart cards: the Intel 8051 and the ARM7TDMI. After presenting the performances of our implementations, we explain for each candidate the main differences between our 8-bit and 32-bit implementations. Then, we compare our results to those of two benchmarks published at the second SHA-3 candidates conference this summer, and deduce a ranking according to performance and what we call *8-bit tolerance*.

1 Introduction

In this study, we implemented seven SHA-3 second-round candidates: BLAKE [3], Blue Midnight Wish [10], Grøstl [9], Keccak [4], Luffa [6], Shabal [5] and Skein [8].

We choose two platforms currently used in smart cards: the Intel 8051, an 8-bit microcontroller, and the ARM7TDMI, a 32-bit processor. We list hereafter the features of both architectures that are particularly relevant in our context.

The Intel 8051 is a very constrained architecture. The main constraints are:

- a low number of 8 general purpose registers,
- a low amount of fast RAM (less than 10 bytes of DATA memory),
- and a limited rotation (shift/rotate is possible by only one position).

It must be noted that we didn't use "pure" 8051, but an extended set of instructions, allowing fast loading of contiguous memory blocks.

The ARM7TDMI is a less constrained architecture. It is a 32-bit core based on the ARMv4T ARM architecture version, with a three-stage pipeline. Its main constraints are:

- 13 general purpose registers
- and no division instruction. Fortunately, it is not a bottleneck for any of the candidates we implemented.

*This work is the result of an internship in Oberthur Technologies' crypto team, supervised by Gilles Piret and Emmanuel Prouff, from 06/04/2010 to 30/09/2010.

We added three more constraints on both architectures.

- Each program should manage all versions of the same candidate. More precisely, each program implements the SHA-3 API [11] for all supported hash sizes (224, 256, 684 and 512 bits).
- Code size should not exceed 10 Kb. This limit only prevents us to unroll all the loops.
- We fixed a development period of two weeks per candidate. This duration corresponds to a classical hash function development period in an industrial context. Thus, the same effort and period have been dedicated to each candidate. So, if a candidate is more optimized than another, it only means the first one was easier to implement and to optimize than the other.

Under the above constraints our implementation strategy was to reach the best tradeoff between timing efficiency and compactness. Thus, our implementations are neither the most efficient nor the most compact ones.

2 Implementations

First of all, we have to precise the metric we use. We only considered the speed of the compression function and ignored the initialization and finalization phases. As a consequence, the speeds we mention only refer to the behavior of the candidates when hashing very long messages.

As far as we were only interested in the compression functions speed, we implemented them in assembly to optimize them as much as possible. As writing mode of operation in assembly does not bring any significant improvement of the overall hash function speed, we implemented them in C (in C51 for 8051 implementations). We list our implementations results in Figure 1. The unit we use is the number of cycles per byte of message hashed. To measure the number of cycles, we ran each function in a simulator and measured the number of cycles necessary for the compression function to hash a message block. We then divided this number of cycles by the size in bytes of the message blocks.

By looking at the results, it can be noted that the efficiency ranking of the seven candidates is independent of the hash size produced by the algorithm for both 8051 (cf Figure 2, only Skein and Keccak switch their respective positions) and ARM (cf Figure 3).

In Table 1, we also give the *ratio* between the 8051 and ARM versions of each function. This *ratio* is the result of the division of the 8051 version speed by the ARM7TDMI version speed. We mention this value to measure if the gap between a function performances widens or narrows (cf Figure 4) with the passage from a 32-bit architecture to an 8-bit architecture. A *ratio* close to 1 means the function can be ported to 8-bit architecture without great loss of performance. A high *ratio* means a loss of performance when ported to 8-bit architecture.

In our study, we will not interpret these values in absolute terms, but only to compare the behavior of the implemented candidates. We remark that all the functions do not support equally the transposition, as we have a mean *ratio* of 6.5 and a standard deviation of 4.1.

A high *ratio* may be the result of one or a combination of the following three elements :

1. **We tweaked the ARM7TDMI version more than the 8051 one.** This argument falls down as, for each function, the same tweaks have been applied to both versions by the same developer.

Algorithm	8051		ARM7TDMI		ratio
	cycles	cycles/byte	cycles	cycles/byte	
Blake 224 Blake 256	41135	642.7	8215	128.4	5
Blake 384 Blake 512	81390	635.9	19157	149.7	4.2
BMW 224 BMW 256	36234	566.2	2227	34.8	16.3
BMW 384 BMW 512	75621	590.8	7316	57.2	10.3
Grøstl 224 Grøstl 256	84915	1326.8	56618	884.7	1.5
Grøstl 384 Grøstl 512	228781	1787.4	155542	1215.2	1.5
Keccak 224	237352	1648.3	50834	353	4.7
Keccak 256		1745.2		373.8	
Keccak 384		2282.2		488.8	
Keccak 512		3296.6		706	
Luffa 224 Luffa 256	28030	875.9	2361	73.8	11.9
Luffa 384	39573	1236.7	3476	108.6	11.4
Luffa 512	52166	1630.1	4758	148.7	11
Shabal 224 Shabal 256 Shabal 384 Shabal 512	25079	391.9	4284	66.9	5.9
Skein 224 Skein 256	62222	1944.4	10079	315	6.2
Skein 384 Skein 512	131001	2046.9	18260	285.3	7.2

Figure 1: Speeds of the implemented candidates.

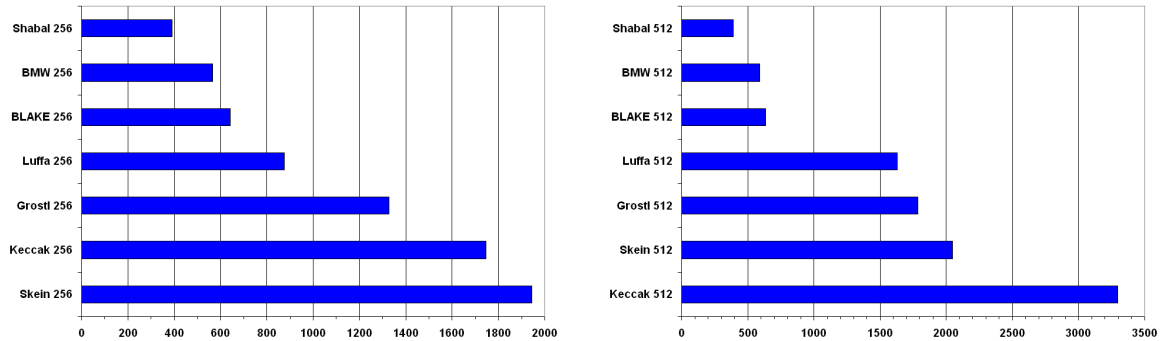

Figure 2: Rankings for 8051.

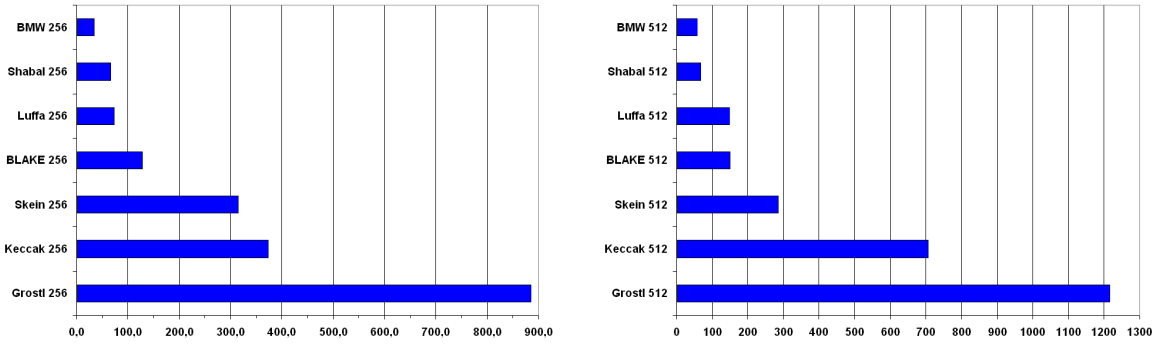

Figure 3: Rankings for ARM.

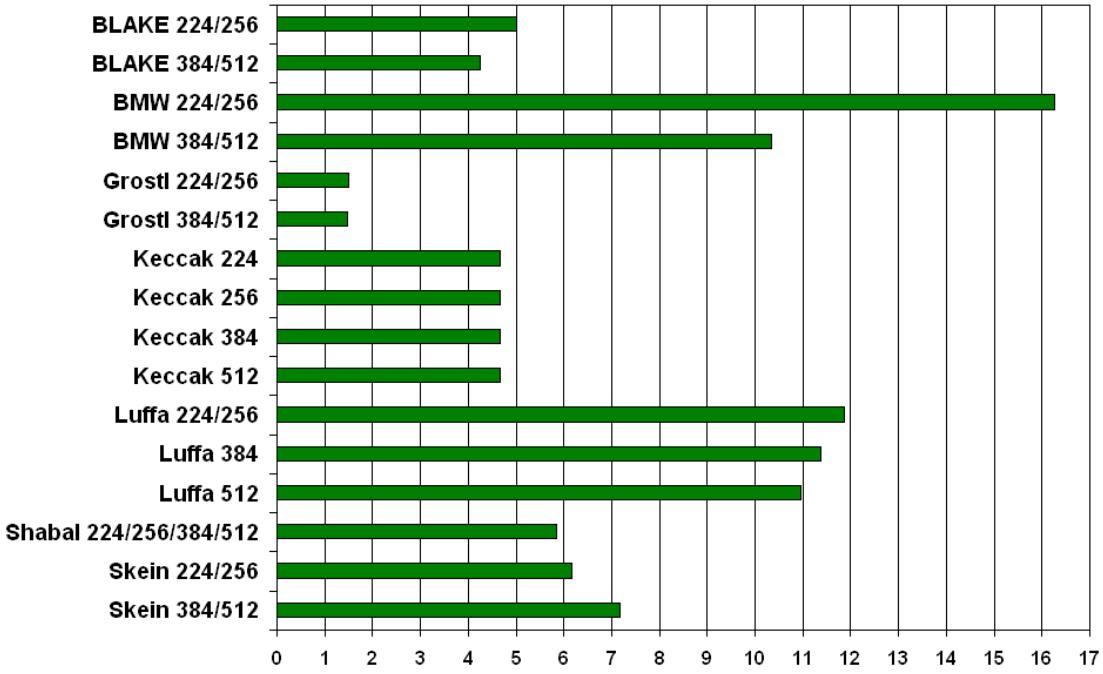

Figure 4: *Ratio* between 8051 and ARM7TDMI implementations.

2. **The candidate takes advantage of the 32-bit architecture.** Two design choices make the function more efficient:

   - the use of 32-bit words and interleaved 32-bit logical or arithmetic instructions and rotations (or shifts). As the ARM is a 32-bit architecture equipped with a barrel shifter, both operations can be carried out in a single cycle. The candidates taking advantage of this free rotation are Blake, BMW, Luffa and Shabal.
   - the computation on intermediate results whose size is smaller than the global register size. Here, we mean a huge number of operations on data of size less than 416 bits ($= 13 \cdot 32$ bits, the capacity of the 13 ARM registers) and more than 64 bits ($= 8 \cdot 8$ bits, the size of eight 8051 registers) as the Q_is in Luffa.

3. **The candidate is affected by a constraint of the 8-bit architecture.** Two design choices can affect the candidates performances:

   - an intensive use of word rotation. As rotation or shift on 8-bit architecture can only be made bit by bit through the carry, it badly impacts the performances. The most affected candidates are BMW, Luffa and Skein.
   - the use of 64-bit words. As we can put in registers only one 64-bit word, each byte of the second operand of any binary operation must be loaded one by one via the accumulator, which is less efficient than using instructions loading multiple registers at once. All the candidates are affected except Grøstl and Shabal (and 256-bit output versions of BMW, BLAKE and Luffa).

Grøstl is the only function dealing with 8-bit words. Thus, it doesn't suffer from any limitation of the 8051, and doesn't take advantage of the 32-bit words. This explains why its *ratio* is close to 1.

3 Comparison with state of the art implementations

The main goal of this section is to compare our results with two performances evaluations presented at the second SHA-3 candidates conference. Since our analysis essentially corroborates the previous ones, our final purpose is to reveal a tendency on these seven candidates performances and their abilities to be ported on embedded architectures.

The first work is sphlib [12]. We choose it because the author followed an approach very similar to ours : each function is implemented by the same developer in two weeks. However, he doesn't target embedded architectures in his study.

The second one is the XBX benchmark [7], an extension of the SUPERCOP-eBASH framework that allows benchmarking on small devices. According to the authors, "The main sources of implementations were SUPERCOP[2] (supercop-20100712.tar.bz2), the avr-crypto-lib [1], it's derivate arm-crypto-lib and sphlib [12]" except for Grøstl and Skein for which assembly implementations have been adapted. We choose this one to compare our implementations to embedded speed reference implementations.

3.1 Ranking comparison

For 8-bit implementations, only XBX benchmark is eligible for comparison. We choose to compare our 8051 results with XBX's ATmega1281 results (cf Figure 5), which is an Atmel AVR microcontroller. The 8051 and the AVR are quite similar except that AVR has 32 registers versus only 8 registers for 8051. We notice that the two rankings differ only by the ranks of Grøstl and Skein. They both have better ranks on XBX implementation because they are the only functions implemented in assembly in XBX benchmark.

For 32-bit implementations, we are able to compare to sphlib and XBX results. We choose to compare to sphlib's ARM920T results and to XBX's Atmel AT91RM9200 results (cf Figure 6). The Atmel AT91RM9200 is a processor based on the ARM920T core. The reason why we choose the ARM920T is that it is based on the same version of the ARM architecture as ARM7TDMI: the ARMv4T. So we can expect the two platforms to behave the same way.

A general tendency emerges. Blue Midnight Wish and Shabal are always on top, followed by BLAKE, Luffa and Skein. Finally, Grøstl and Keccak are behind.

256 bit output		
Rank	8051	ATmega1281 [7]
1	Shabal	Shabal
2	BMW	BMW
3	BLAKE	BLAKE
4	Luffa	Skein
5	Grøstl	Grøstl
6	Keccak	Luffa
7	Skein	Keccak
512 bit output		
Rank	8051	ATmega1281 [7]
1	Shabal	Shabal
2	BMW	Skein
3	BLAKE	BMW
4	Luffa	Grøstl
5	Grøstl	BLAKE
6	Skein	Luffa
7	Keccak	Keccak

Figure 5: Rankings comparison on 8-bit architectures.

256 bit output			
Rank	ARM7TDMI	ARM920T [12]	AT91RM9200 [7]
1	BMW	Shabal	Shabal/BMW
2	Shabal	BMW	
3	Luffa	BLAKE	BLAKE
4	BLAKE	Luffa	Luffa
5	Skein	Skein	Skein
6	Keccak	Grøstl	Keccak
7	Grøstl	Keccak	Grøstl
512 bit output			
Rank	ARM7TDMI	ARM920T [12]	AT91RM9200 [7]
1	BMW	Shabal	Shabal
2	Shabal	BMW	BMW
3	Luffa	BLAKE	BLAKE
4	BLAKE	Skein	Skein
5	Skein	Luffa	Luffa
6	Keccak	Keccak	Grøstl
7	Grøstl	Grøstl	Keccak

Figure 6: Rankings comparison on 32-bit architectures.

Now, let's compare more deeply our results to those of the two other studies. On both Figures 7 and 8, the *coefficient* is the division of our implementation speed by the minimum of sphlib's [12] and XBX's [7]. The study of this coefficient allows us to see if we equally optimized each candidate or not.

256 bit output			
Algorithm	8051	ATmega1281 [7]	coefficient
BLAKE	642.7	884	0.7
BMW	566.2	485	1.2
Grøstl	1326.8	1309	1.0
Keccak	1745.2	4572	0.4
Luffa	875.9	2722	0.3
Shabal	391.9	210	1.9
Skein	1944.4	1204	1.6
512 bit output			
Algorithm	8051	ATmega1281 [7]	coefficient
BLAKE	635.9	3795	0.2
BMW	590.8	2969	0.2
Grøstl	1787.4	3365	0.5
Keccak	3296.6	7945	0.4
Luffa	1630.2	4688	0.3
Shabal	391.9	210	1.9
Skein	2046	1443	1.4

Figure 7: Results comparison on 8-bit architectures.

On 8-bit, the performance coefficients are around 0.5 for each candidate except for Shabal and Skein. XBX's Shabal implementation is the avrcryptolib-asm one, and Skein implementation is the Fhreefish one. The only explanation we can give is that Shabal and Skein benefit a lot of the AVR's 32 registers. For Skein, the Threefish implementation makes use of macro to unroll eight Threefish rounds, and manipulate the state in the registers in order to avoid permutations. In 8051, we can't avoid these permutations, as the state (in both versions) does not fit in registers.

On 32-bit, the performance coefficients are around 2 except for Grøstl, Shabal and Skein implementations, which look less optimized compared to the other functions. The implementations benched by XBX are sphlibs'. Thus, they share a big code size. This big code size is due to an intensive use of macros. As our time/space tradeoff limits us to a code size of 10Kb, we were not able to do such optimizations. This element explains the performance drop between sphlib implementations and ours. For Skein, we think the drop comes especially from our choice to not unroll the eight rounds of Threefish as the 512-bits version state does not fit in registers.

A surprising fact is that we have the same coefficients for Skein and Grøstl. This means the time/space tradeoff we made impacts the two functions equally.

We can also notice that, for each candidate, passing from 256- to 512-bits output decreases the coefficient on both architectures, except for Luffa on 32-bit architecture. It may be the result of the code pooling between all the versions of each function. Contrary to sphlib and XBX implementation, we make an intensive use of functions and not macros. As we have less function calls on 512 bit version per byte, the coefficient decreases.

256 bit output				
Algorithm	ARM7TDMI	ARM920T [12]	AT91RM9200 [7]	coefficient
BLAKE	128.4	42.6	47	3.0
BMW	34.8	30.5	25	1.4
Grøstl	884.7	288.5	234	3.8
Keccak	373.8	300	150	2.5
Luffa	73.8	73.5	86	1.0
Shabal	66.9	23.3	25	2.9
Skein	315.0	83.3	122	3.8
512 bit output				
Algorithm	ARM7TDMI	ARM920T [12]	AT91RM9200 [7]	coefficient
BLAKE	149.7	98.7	119	1.5
BMW	57.2	69.4	81	0.8
Grøstl	1215.2	681.8	562	2.2
Keccak	706.0	576.9	571	1.2
Luffa	148.7	133.9	178	1.1
Shabal	66.9	23.3	25	2.9
Skein	285.3	129.3	156	2.2

Figure 8: Results comparison on 32-bit architectures.

3.2 32-bit to 8-bit tolerance comparison

Another point we want to corroborate is the tolerance to the passage from a 32-bit to an 8-bit architecture for each candidate.

For this purpose, we observed the *ratios* relative to the passage from AT91RM9200 to ATmega1281 in XBX benchmark [7]. These results are recorded in Figure 9.

Algorithm	ATmega1281 [7]	AT91RM9200 [7]	ratio
BLAKE 224/256	884	47	18.8
BLAKE 384/512	3795	119	31.9
BMW 224/256	485	25	19.4
BMW 384/512	2969	81	36.7
Grøstl 224/256	1309	234	5.6
Grøstl 384/512	3365	562	6.0
Keccak 256	4572	150	30.5
Keccak 512	7945	571	13.9
Luffa 224/256	2722	86	31.7
Luffa 512	4688	178	26.3
Shabal	210	25	8.4
Skein 224/256	1204	122	9.9
Skein 384/512	1443	156	9.3

Figure 9: Ratios between ATmega1281 and AT91RM9200 XBX implementations[7].

If we compare these *ratios* with those of our implementations (cf Figure 10), we have the same tendency. BMW is the most impacted function, followed by Luffa, Keccak and BLAKE, Skein, Shabal, and finally, Grøstl.

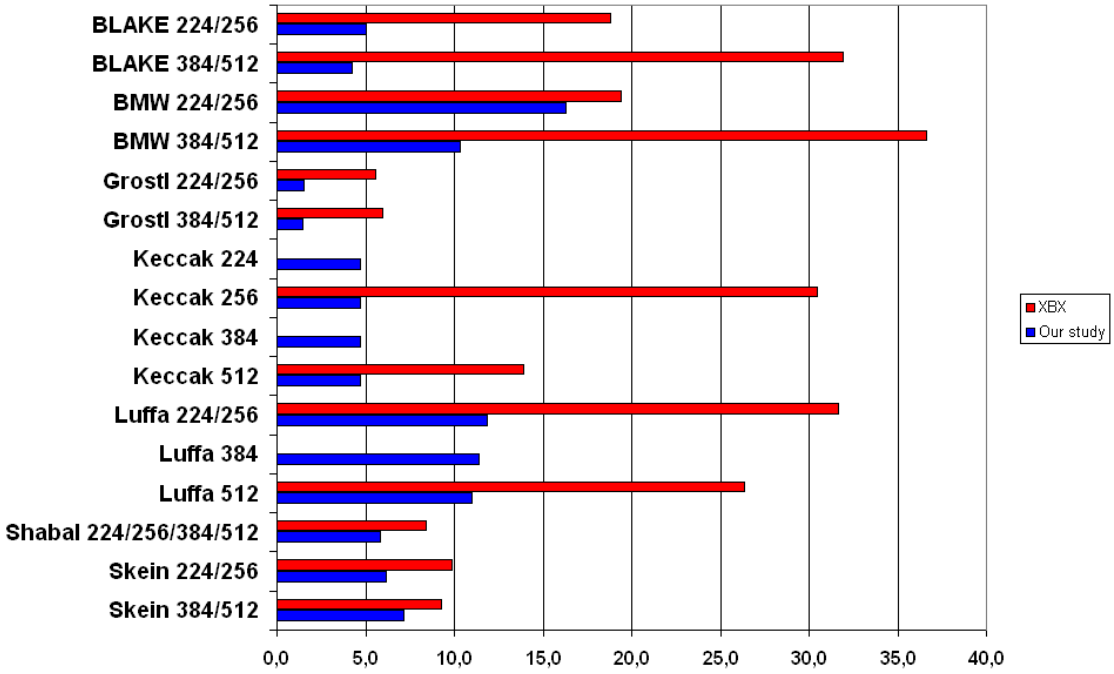

Figure 10: *Ratios* comparison.

We noted only two major differences:

- In our implementations, for each function, the *ratio* of the 224/256 version is always higher than the one of 512 version (except for Skein). It is the opposite on XBX implementations (even for Skein !).
- There is a huge difference between Keccak 256 and Keccak 512 XBX's *ratios*. The only explanation is that the implementation benched by XBX on ATmega1281 (sphlib) and on AT91RM9200 (opt32u6) are different.

4 Conclusion

In this paper, we implemented seven candidates of the SHA-3 second-round competition, according to common industrial production methods. From these implementations, we extracted a trend on candidates performances on embedded systems (particularly on smart cards). Thus, Blue Midnight Wish and Shabal are always on top of our rankings, followed by BLAKE, Luffa and Skein. Finally, Grøstl and Keccak are behind.

We also studied the tolerance to the passage from 32-bit to 8-bit of each candidates. Performance-wise, BMW is the most impacted function, followed by Luffa, Keccak and BLAKE, Skein, Shabal, and finally, Grøstl.

An interesting continuation to this work would be to extend it to other candidates, to identify more elements influencing the 8-bit port tolerance *ratio* and to quantify the usage of these elements within each function.

References

[1] avr-crypto-lib. http://avrcryptolib.das-labor.org/trac.

[2] ebash: Ecrypt benchmarking of all submitted hashes. http://bench.cr.yp.to/ebash.html.

[3] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. Sha-3 proposal blake. Submission to NIST, 2008.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. Submission to NIST (Round 2), 2009.

[5] Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-François Misarsky, Marìa Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, and Marion Videau. Shabal, a submission to nist's cryptographic hash algorithm competition. Submission to NIST, 2008.

[6] Christophe De Canniere, Hisayoshi Sato, and Dai Watanabe. Hash function luffa: Specification. Submission to NIST (Round 2), 2009.

[7] Marcus Himmel Christian Wenzel-Benner, Jens Greaf. Xbx benchmarking results august 2010. https://xbx.das-labor.org/trac/wiki.

[8] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. Submission to NIST (Round 2), 2009.

[9] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a sha-3 candidate. Submission to NIST, 2008.

[10] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jorn Amundsen, and Stig Frode Mjolsnes. Cryptographic hash function blue midnight wish. Submission to NIST (Round 2), 2009.

[11] NIST. Ansi C cryptographic API profile for SHA-3 candidate algorithm submissions.

[12] Thomas Pornin. Comparative performance review of the sha-3 second round candidates. http://www.saphir2.com/sphlib.