

Java Implementation and Performance Analysis of the Second Round SHA-3 Candidates on Constrained Devices

Kim Andre Martinsen, Mats Knutsen, Danilo Gligoroski

Faculty of Information Technology, Mathematics and Electrical Engineering, Institute of Telematics,
Norwegian University of Science and Technology, Trondheim, Norway,
e-mail: {kimandm, matskn}@stud.ntnu.no , Danilo.Gligoroski@item.ntnu.no

Abstract

In this paper, we show the results of our implementation of the second round SHA-3 candidates in Java and perform a cost and performance analysis of them on a low-cost 32-bit ARM9 CPU by measuring cycles/byte and ROM requirements. The analysis is conducted on the Sun SPOT platform, by Sun Microsystems, with a Squawk Virtual Machine.

1 Introduction

The National Institute of Standards and Technology (NIST) is in the process of selecting a new cryptographic hash algorithm through a public competition [1]. At the First SHA-3 Candidate Conference several individuals stressed the need for supporting constrained platforms such as mobile phones and smart cards [1].

To our knowledge, no work has been demonstrated for comparing Java implementations of the SHA-3 candidates on constrained platforms. In this paper, we compare the implementation costs of the remaining SHA-3 candidates on a low-cost 32-bit ARM9 CPU by measuring cycles/byte and ROM costs.

In the remaining part of this section, we attempt to explain the importance of considering Java implementations on constrained devices for hash implementations. Firstly, the number of constrained devices, such as smart cards, surrounding us daily is rapidly increasing. In 2008 5.085 billion smart cards shipped globally, while the estimates for 2010 are 5.455 billion smart cards [2]. Further, smart cards have a wide range of applications spanning from cryptography and health care to identification and authentication.

Java Card is the Java initiative for running Java applications on smart cards. With Java being one of the most popular programming languages, Java Card simplifies the development of applications for smart cards. Java Card 3.0, released late 2009, is tailored for next generation of smart cards, which have 32-bit processors [3].

For Java to run on the ARM, a Virtual Machine(VM) resides on top of the CPU. This VM, Squawk VM, is designed for the next generation of smart cards [4]. Squawk is a small VM running without any Operating System (OS), this allows for a simpler and more compact VM. Java VMs are typically written in native languages, such as C and C++. Squawk however, has a different approach as it is written almost entirely in Java. This provides ease of portability and a seamless integration of the VM and the application resources such as threads and objects [4].

At the time of writing, no commercial version of the new Java Card running version 3.0 is available. To compensate for this, the performance tests are conducted on a Sun SPOT¹. The Sun SPOT is a small embedded device developed at Sun Labs. As with Java Card 3.0, Sun SPOT is running the Squawk VM on a low-cost 32-bit processor [4].

This paper is structured as follows. Section 2 gives a brief description of the challenges faced, when writing C code to Java code. Section 3 gives an overview of our performance and cost measurements. In Section 4, the results from our measurements are presented, while Section 5 concludes the findings from this paper.

2 Java Implementation of the Candidates

In contrast to low-level languages, like C, the memory management in Java is not handled by the programmer. For instance, in C the *union* is a value that may have several representations in various formats. The primary usefulness of a union is to conserve space, since it provides a way of letting many different types be stored in the same space. The Java language guarantees type safety, so each member of a *union* has to be implemented as single variables which have to be synchronized when one of them are alternated.

The Java implementations of the candidates are based on the reference implementation representing the candidate in the second round. To our knowledge, no optimized Java implementation of all the candidates are publicly presented. With the challenges presented above, and also considering time constraints, none of the candidates have been optimized in any way. The goal of this work is to give a balanced view of the candidates implemented on a new platform.

3 Overview of Our Performance and Cost Measurements

This section presents an overview of our comparison of the remaining SHA-3 candidates, in terms of performance and cost. In constrained devices, power consumption and the area required to implement the algorithm are limiting factors [1]. Power consumption usually reduces to computational efficiency, which will be measured in

¹SPOT stands for Small Programmable Object Technology

cycles/byte. The size, in kilobytes (kB), of the compiled file loaded on the Squawk-capable constrained device is used as the measurement for ROM requirements of the algorithms.

In our comparison, we consider Java translated implementations of the submitted round 2 reference implementations. Further, our main target for estimation and comparison are those variants of the hash functions which produce a 256-bit message digest. External functionality like salting or keyed hashing modes has not been implemented.

3.1 Measuring Performance

To measure the performance of the candidates, a Java MIDlet² were deployed on the Sun SPOT. The MIDlet executed each of the algorithms 4500 times for the follow input lengths: 8, 64, 576, 1536 and 4096 bytes. Each Sun SPOT processor board has two AT91 Timer Counters that are part of the ARM920T system-on-a-chip (SOC) [5]. A AT91 Timer Counter were used to measure the time consumed by each algorithm, while processing the input messages. The output of these AT91 Timer Counters is microseconds.

Our results are presented as a lower-quartile, a median and a upper-quartile of the 4500 measurements taken of each candidate, for each of the five input lengths.

3.2 Measuring Cost

Applications deployed onto the Squawk VM are first compiled to regular Java *class* files and then further to a *suite* file. Squawk VM includes a mechanism for serializing a graph of objects, these objects becomes a collection of internal classes encapsulated in the *suite*. On average, a suite file is 35% the size of the corresponding class file [4].

As each candidate can not be deployed to the device without extra application code, we have only measured the size of the class file corresponding to each algorithm. This is the compiled code before it is packed into a *suite* with the rest of the files needed.

4 Results

This section presents the results generated from our work. Worth mentioning is that Hamsi was not able to compile for Squawk VM, and hence no results for Hamsi are presented in this section. This is due to Hamsi exceeding a Java constant size restriction in Squawk.

²A MIDlet is an application that runs on implementations of the Mobile Information Device Profile, one of the Java ME specifications.

4.1 Performance Results

In addition to measure the performance of the Java implementations on the Sun SPOT, a performance test was conducted on a Intel Pentium 4 3GHz desktop computer as a frame of reference. The results from the Intel test are depicted in Tab. 1.

Tab. 2 depicts the results from the performance test on the Sun SPOT ARM920T 180MHz running Squawk VM.

4.2 Cost Results

Tab. 3 depicts our cost measurements of the SHA3 candidates.

Cycles/byte for 4096 bytes				Cycles/byte for 1536 bytes			
Quartile	Median	Quartile	Hash	Quartile	Median	Quartile	Hash
245,38	248,12	317,46	SHAvite3256	268,14	271,34	385,74	SHAvite3256
272,80	274,93	281,46	BMW256	318,13	328,15	338,72	BLAKE256
315,68	319,04	329,70	BLAKE256	359,04	364,91	375,01	BMW256
367,64	369,17	387,42	Shabal256	417,15	418,37	477,38	Shabal256
495,9	508,51	514,13	Skein256	512,62	523,30	566,04	Skein256
602,62	608,01	695,07	Luffa256	618,55	624,88	735,63	Luffa256
2332,46	2343,57	2530,97	Hamsi256	2369,46	2377,24	2583,17	Hamsi256
6013,94	6063,81	6248,78	Keccak256	6221,30	6246,18	6549,47	Keccak256
6322,39	6424,96	6537,82	Fugue256	8550,46	8726,34	8937,34	CubeHash256
7442,60	7576,06	7728,87	CubeHash256	10266,70	10357,05	10866,79	Fugue256
12383,79	12464,45	12643,56	JH256	12948,53	12993,60	13444,31	JH256
16607,95	16736,06	16919,29	ECHO256	18017,60	18192,51	18688,41	ECHO256
18218,33	18397,44	18528,83	SIMD256	18678,86	18732,52	19221,21	SIMD256
29628,20	29691,23	29883,10	Grøstl256	30785,39	30843,26	31155,77	Grøstl256

Cycles/byte for 576 bytes				Cycles/byte for 64 bytes			
Quartile	Median	Quartile	Hash	Quartile	Median	Quartile	Hash
330,51	335,05	604,98	SHAvite3256	680,57	689,72	748,01	Blake256
351,84	354,66	367,78	Blake256	1343,27	1812,52	1889,51	Skein256
550,70	552,48	825,94	Shabal256	1124,42	2229,52	2455,31	SHAvite3256
552,34	559,61	581,36	BMW256	2258,28	2261,63	3812,37	Shabal256
552,84	564,96	801,61	Skein256	1266,67	2486,16	2526,57	Luffa256
664,36	671,89	928,79	Luffa256	2802,23	2871,14	2995,61	BMW256
2450,28	2468,35	2908,26	Hamsi256	3528,40	3592,92	6070,05	Hamsi256
6954,92	6979,11	7305,88	Keccak256	12815,51	12879,98	14057,14	Keccak256
11530,05	11792,84	12139,01	CubeHash256	34333,25	34485,61	42122,44	JH256
14025,29	14988,33	15552,61	JH256	36073,10	36169,83	37182,34	SIMD256
19949,92	19966,03	20633,97	SIMD256	49545,17	49675,36	51605,04	ECHO256
20594,54	20953,36	21975,75	Fugue256	62471,59	72355,69	74907,43	CubeHash256
21447,48	21498,72	22522,79	ECHO256	73213,21	73382,25	74007,53	Grøstl256
33865,05	33931,38	34276,03	Grøstl256	153739,06	157110,19	162306,47	Fugue256

Cycles/byte for 8 bytes			
Quartile	Median	Quartile	Hash
2951,25	2951,63	3088,56	BLAKE256
5260,04	5389,50	10670,87	Luffa256
7023,61	7186,13	7579,10	SHAvite3256
6300,14	10137,00	10456,70	Skein256
11785,20	12190,50	12981,87	Hamsi256
15362,36	15398,25	15470,33	Shabal256
24660,25	25022,25	25829,00	BMW256
103339,02	103809,75	105316,47	Keccak256
274813,00	275885,25	320095,42	JH256
288105,91	288845,25	291901,48	SIMD256
354701,32	355699,50	357306,72	Grøstl256
396175,13	397146,38	405545,12	ECHO256
494397,41	506217,00	528178,64	CubeHash256
1181568,87	1226854,50	1250182,79	Fugue256

Table 1: Performance Measurements on Intel Pentim 4 3GHz

Cycles/byte for 4096 bytes				Cycles/byte for 1536 bytes			
Quartile	Median	Quartile	Hash	Quartile	Median	Quartile	Hash
7225,57	7244,65	8323,44	BMW256	8801,49	8824,82	11141,11	BMW256
9265,59	9295,20	9305,80	Shabal256	10052,41	10232,85	10238,78	Shabal256
14247,87	14703,88	15155,43	SHAvite3256	15489,48	15521,06	17663,36	SHAvite3256
15829,13	16049,01	17146,23	BLAKE256	15899,37	16479,50	17219,64	BLAKE256
16261,50	16400,38	16416,46	Skein256	16755,94	16937,27	17001,17	Skein256
45044,45	47045,31	47964,37	Luffa256	48255,04	48991,48	49225,08	Luffa256
73869,55	74632,52	75418,86	Keccak256	78237,43	79339,80	80538,97	Keccak256
152272,21	153401,62	154566,34	SIMD256	156868,67	158017,99	159768,41	SIMD256
162051,47	166768,41	170644,45	CubeHash256	192055,46	194056,17	197254,42	CubeHash256
211721,88	213019,05	214382,31	Fugue256	274624,17	276213,91	278397,26	Fugue256
259742,44	261185,53	262626,81	ECHO256	285548,90	287631,19	289857,88	ECHO256
341576,27	346572,31	351576,22	Grøstl256	358732,56	370554,43	381650,82	Grøstl256
670755,25	676688,53	682705,99	JH256	775918,61	779317,67	782712,08	JH256
N/A	N/A	N/A	Hamsi256	N/A	N/A	N/A	Hamsi256

Cycles/byte for 576 bytes				Cycles/byte for 64 bytes			
Quartile	Median	Quartile	Hash	Quartile	Median	Quartile	Hash
12771,16	13305,59	13318,75	Shabal256	31304,13	32370,02	33474,77	BLAKE256
13321,63	13584,04	19580,24	BMW256	34681,42	35358,36	35690,64	Skein256
16950,20	17351,48	18905,47	BLAKE256	48044,74	50366,21	52670,80	Shabal256
17782,80	17945,78	17970,88	Skein256	59303,28	59664,80	67866,37	SHAvite3256
18376,43	18419,88	21094,73	SHAvite3256	70075,04	70392,20	108408,90	BMW256
52356,80	52413,98	52737,47	Luffa256	93142,73	93634,91	93810,89	Luffa256
87915,88	88491,50	91843,89	Keccak256	160815,70	161055,29	169345,14	Keccak256
166456,86	170636,30	173316,46	SIMD256	294708,55	295054,69	313714,86	SIMD256
271857,87	274563,03	279125,11	CubeHash256	804118,77	805950,74	832831,25	ECHO256
344922,19	350461,26	356261,96	ECHO256	925242,90	928087,90	932236,31	Grøstl256
405246,67	419416,73	421033,69	Grøstl256	1293450,33	1305829,03	1338317,15	CubeHash256
438264,92	442195,26	446376,91	Fugue256	2056855,65	2104913,31	2167669,40	JH256
819573,33	828671,60	837601,74	JH256	2499328,19	2503167,99	2567218,87	Fugue256
N/A	N/A	N/A	Hamsi256	N/A	N/A	N/A	Hamsi256

Cycles/byte for 8 bytes			
Quartile	Median	Quartile	Hash
135025,91	137743,47	150609,67	BLAKE256
203277,62	206386,83	211322,02	Skein256
325306,92	351510,28	354179,03	Shabal256
366775,27	369936,17	415334,07	SHAvite3256
374296,11	379263,30	384047,28	Luffa256
563348,94	566022,29	879709,11	BMW256
1295028,32	1296952,13	1363139,68	Keccak256
2340421,70	2342937,10	2491367,39	SIMD256
4563850,93	4568942,48	4704900,83	Grøstl256
6422366,05	6438679,54	6652598,15	ECHO256
9239108,19	9293066,96	9536356,32	CubeHash256
16458116,05	16838963,18	17339729,84	JH256
18689930,39	18710416,85	19212191,51	Fugue256
N/A	N/A	N/A	Hamsi256

Table 2: Performance Measurements on Sun SPOT (ARM920T 180MHz)

Hash	Size in kB
Shabal	3,65
CubeHash	4,19
JH	4,93
Keccak	5,94
SIMD	6,13
Luffa	6,26
Grøstl	6,36
BMW	6,86
Skein	7,10
BLAKE	7,31
Fugue	10,06
ECHO	11,6
SHAVite3	17,6
Hamsi	N/A

Table 3: Cost Measurements

5 Conclusions

In this paper, we compared the performance and cost of the remaining SHA-3 candidates on a low-cost ARM920T running the Java based Squawk VM. The results indicate that the computational efficiency of our Java implementations is weak compared to optimized C.

We confirm that software implementations on constrained devices are relatively slow, if not designed properly. Candidates requiring a lot of memory management mechanisms, and with poor reference implementations, can have poor performance on constraint platforms running Java. Some of these are mentioned in the introduction, and was the motivation behind this work.

References

- [1] National Institute of Standards and Technology: “Status Report on the First Round of the SHA-3 Cryptographic Hash Algorithm Competition“. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf (2010/04/10), September 2009.
- [2] Eurosmart: “Market Overview“. Available: <http://www.eurosmart.com/index.php/publications/market-overview.html> (2010/04/27).
- [3] Oracle: “Java Card 3.0.1 Platform Specification“. Available: <http://java.sun.com/javacard/3.0.1/specs.jsp> (2010/04/27).

- [4] N. Shaylor, D. N. Simon and W. R. Bush: “A Java virtual machine architecture for very small devices”, 2003. Available: <http://www.grundu.net/papers/squawk.pdf>
- [5] R. Goldman: “Using the AT91 Timer/Counter”, 2007. Available: <http://www.sunspotworld.com/docs/AppNotes/TimerCounterAppNote.pdf>