Towards Algorithm Agility for Wireless Sensor Networks
Comparison of the Portability of Selected Hash Functions

Manuel Koschuch, Matthias Hudler and Zsolt Saffer
Competence Centre for IT-Security, FH Campus Wien - University of Applied Science, Favoritenstrasse 226, 1100, Vienna, Austria

Keywords: Iris, MicaZ, Sensor Motes, Hash Functions, Performance, ATmega128.

Abstract: Cryptographic hash functions are an important building block used in many cryptosystems. The flexibility and ability of a system designer to choose the most fitting function for a given system enables fast, efficient and secure designs. In this position paper we give preliminary results of porting three selected hash algorithms to Iris and MicaZ Sensor nodes in terms of achieved performance, memory requirements and the influence of different compiler optimizations on these measurements. Our main goal is to provide a sort of baseline approximation of how much effort is needed to port reference code of these algorithms to a new platform without trying to optimize it, leaving all this work to the compiler; enabling designers to not having to stick to already ported algorithms, when they might be suboptimally suited for a given environment.

1 INTRODUCTION

With the advent of wireless sensor networks (WSNs), the need for efficient cryptography in these environments rises too. A WSN typically consists of a potentially huge number of small, independent sensor nodes (motes), dispatched over a wide area, communicating their sensor readings in a hop-to-hop fashion over the air to a central basestation.

Since the messages are transfered over the air interface and can be intercepted, manipulated, or even injected by potential malicious eavesdroppers, certain precautions have to be taken to secure the communication between the individual nodes. This gets complicated by the severe restrictions in terms of available energy, memory and processing power of the single motes. Ordinary cryptographic systems, as they are used in wired PC-based surroundings, usually cannot be easily adapted to sensor networks. Every single primitive of the cryptosystem has to be either carefully selected and evaluated for the use on the constrained, usually 8-bit, platforms, or even specifically tailored to fit this environment.

This makes deploying a new, secure network, or even the switch from one mote-processor architecture to a different one, often cumbersome and expensive. In addition it entices to only use already existing versions of algorithms, thereby creating possible dangerous dependencies on certain primitives.

In this position paper, we focus on one such primitive, used as a basic building block for secure networks: hash functions. We try porting three selected hash algorithms, based on very different concepts (SHA-1 for its still widespread use, Grøstl as a SHA-3 final candidate, and Tiger, as an example for a 64-bit based hash), to Iris and MicaZ motes with an 8-bit ATmega microcontroller and 8, respectively 4, kilobytes of RAM. The main questions we are trying to answer are: how much effort is required to fit the algorithms to these constrained environments? Do all of these algorithms work efficiently on the sensor motes? How much ROM and RAM are required to perform the hashing? How much, if any, influence do standard compiler optimizations in this environment have on the performance of the implementation?

The availability of grounded results for the feasibility of using different hash algorithms in sensor networks should allow for a more diverse approach when selecting appropriate primitives and giving system designers more flexibility in their decisions, as well as the possibility to choose the kind of algorithm best suited for the environment at hand.

The remainder of this paper is now structured as follows:

Section 2 gives an overview of cryptographic hash functions in general and details the internal structures of the three functions examined in this work. Section 3 describes our implementation and discusses the

Koschuch M., Hudler M. and Saffer Z..
Towards Algorithm Agility for Wireless Sensor Networks - Comparison of the Portability of Selected Hash Functions.
DOI: 10.5220/0004612500640068
In *Proceedings of the 4th International Conference on Data Communication Networking, 10th International Conference on e-Business and 4th International Conference on Optical Communication Systems* (DCNET-2013), pages 64-68
ISBN: 978-989-8565-72-3

preliminary results, while finally Section 4 provides some closing remarks and shortly discusses our next steps.

2 CRYPTOGRAPHIC HASH FUNCTIONS

A general hash function has two important properties:

Compression. an input of arbitrary length is mapped to an output of fixed length, e.g. 160 or 192 bits

Efficiency. calculation of the hash value for a given input is fast and easy

In addition to the properties mentioned above, a *cryptographic* hash function also has to have three additional characteristics ((Menezes et al., 2001)):

Pre-image Resistance. given a hash function H, an input x, and $y = H(x)$, it should be practically infeasible to determine x from the knowledge of H and y.

Second Pre-image Resistance. given a hash function H and an input x, it should be practically infeasible to find an $x' \neq x$, such that $H(x) = H(x')$.

Collision Resistance. given a hash function H, it should be practically infeasible to find two distinct inputs x and x' (with $x \neq x'$), such that $H(x) = H(x')$.

The last property (collision resistance) is the hardest one to fulfill and, due to the birthday paradox, the output of an up-to-date hash function should be at the very least 160 bits long, so that the effort for an attacker to brute-force a collision stays above 2^{80} $(= \sqrt{2^{160}})$.

All the hash functions that were observed in this work qualify as secure cryptographic hash functions with, at least currently, no known weaknesses that would prevent their use in practice, although in the case of SHA-1 a possible complete break is already long overdue. The following Subsections give a short overview of the three hash functions that were selected for this work.

2.1 SHA-1

The Secure Hash Algorithm (Eastlake and Jones, 2001) is still the de-facto standard for secure hash functions. Although in October 2012 a new SHA-3 standard (based on KECCAK, see (Bertoni et al., 2011)) has been selected, and several members of the SHA-2 family with different digest sizes are available since 2001 (National Institute of Standards and Technology, 2012), SHA-1 is still very often used in practical applications, also due to the huge number of readily available implementations for a variety of devices and environments.

SHA-1 produces a 160-bit digest and processes a message in blocks of 64 bytes. Internally it uses 32-bit words and 80 rounds. Collision attacks with a complexity of 2^{51} have been reported (Manuel, 2011) and new designs should avoid the algorithm whenever possible, but due to its current prevalence we included it in this work.

The main body of the SHA-1 implementation consists of a sequence of circular shifts, xors and additions, together with a nonlinear combination of five intermediate words that change every twenty rounds. For this work we used the reference code from the SDCC port of the SHA-1 implementation by Vinicius Kursancew[1]. The main loops are given in Listing 1.

Listing 1: SHA-1 Kernel.

```
for(t=0;t<20;t++){
  temp =  A <<<5 + ((B & C)
       | ((~B) & D)) + E + W[t] + K[0];
  E = D; D = C; C = B <<<30; B = A; A = temp;
}

for(t=20;t<40;t++){
  temp = A <<<5 + (B ^ C ^ D) + E + W[t] + K[1];
  E = D; D = C; C = B <<<30; B = A; A = temp;
}

for(t=40;t<60;t++){
  temp = A <<<5 + ((B & C)
      | (B & D) | (C & D)) + E + W[t] + K[2];
  E = D; D = C; C = B <<<30; B = A; A = temp;
}

for(t=60;t<80;t++){
  temp = A <<<5 + (B ^ C ^ D) + E + W[t] + K[3];
  E = D; D = C; C = B <<<30; B = A; A = temp;
}
```

2.2 Grøstl

Grøstl (Gauravaram et al., 2011) was one of the five finalists in the SHA-3 competition, producing digest sizes of either 256 and 512 bits. The internal structure of Grøstl is very similar to the one of the Advanced Encryption Standard (AES, (National Institute of Standards and Technology, 2001)), even sharing its 256-byte S-Box. After 10 rounds (for the 256-bit variant), operating on 64-byte message blocks, the internal 512-bit state is truncated to the final output. The

[1]www.vkcorp.org

operations performed in the individual functions during a round are either table lookups, shifts or xors. The main loops are given in Listing 2. The code was taken from the reference implementation by Soeren S. Thomsen and Krystian Matusiewicz[2].

Listing 2: Grøstl Kernel.

```
call GroestlProcess.P(ctx, temp1); /* P(h+m) */
call GroestlProcess.Q(ctx, temp2); /* Q(m) */

/* apply P-permutation to x */
void P(hashState *ctx, u8 x[ROWS][COLS1024]) {
 u8 i;
 Variant v = P512;
 for (i = 0; i < ctx->rounds; i++) {
  AddRoundConstant(x, ctx->columns, i, v); //xors
      and shifts
  SubBytes(x, ctx->columns); //table lookups
  ShiftBytes(x, ctx->columns, v); //shifts
  MixBytes(x, ctx->columns); //xors and shifts
 }
}

/* apply Q-permutation to x */
void Q(hashState *ctx, u8 x[ROWS][COLS1024]) {
 u8 i;
 Variant v = Q512;
 for (i = 0; i < ctx->rounds; i++) {
  AddRoundConstant(x, ctx->columns, i, v); //xors
      and shifts
  SubBytes(x, ctx->columns); //table lookups
  ShiftBytes(x, ctx->columns, v); //shifts
  MixBytes(x, ctx->columns); //xors and shifts
 }
}
```

2.3 Tiger

The Tiger hash function (Anderson and Biham, 1996) is, while already quite old, an interesting case for our study, since it heavily relies on arithmetic on 64-bit words. Several weaknesses in the algorithm have already been discovered (Mendel and Rijmen, 2007; Wang and Sasaki, 2010), yet currently there is no feasible attack against a full round version of Tiger known. Tiger also operates on 64-byte message blocks, using three passes with eight rounds each, and producing a 192-bit digest. At its core it makes heavy use of lookups into four tables, each consisting of 256 64-bit words, thereby requiring $(4*256*64)/8 = 8,192$ bytes only for storing the tables. In this work we were particularly interested if it was possible to port an algorithm like this (using 64-bit arithmetic and quite a lot of memory) to our constrained environments, how much effort would be required to perform the port, and how fast the result would be. Listing 3 gives the main kernel of the hash function, taken from the reference code[3].

Listing 3: Tiger Kernel.

```
save_abc(); //only assignments
pass(a,b,c,5); //table lookups and xors
key_schedule(); //xors and subtractions
pass(c,a,b,7); //table lookups and xors
key_schedule(); //xors and subtractions
pass(b,c,a,9); //table lookups and xors
feedforward(); //xors and subtractions

void pass(a,b,c,mul){
 round(a,b,c,x0,mul);
 round(b,c,a,x1,mul);
 round(c,a,b,x2,mul);
 round(a,b,c,x3,mul);
 round(b,c,a,x4,mul);
 round(c,a,b,x5,mul);
 round(a,b,c,x6,mul);
 round(b,c,a,x7,mul);
}

void round(a,b,c,x,mul){
 c ^= x ;
 a -= t1[c_0] ^ t2[c_2] ^ t3[c_4] ^ t4[c_6] ;
 b += t4[c_1] ^ t3[c_3] ^ t2[c_5] ^ t1[c_7] ;
 b *= mul;
}
```

3 IMPLEMENTATION AND PRELIMINARY RESULTS

We implemented the three algorithms detailed in Section 2 on Memsic Iris and MicaZ motes[4]. Both are equipped with an 8-bit Atmel AVR ATmega128 microprocessor (the 1281 version in the case of Iris, the 128L variant for the MicaZ), clocked at 8 MHz, with the Iris motes able to utilize up to 8KB of RAM, whereas the MicaZ motes are limited to only 4KB. Both motes are provided with 128KB of Flash ROM, in addition to 512KB of measurement ROM to hold sampled values (which was not used for our evaluation).

In practice, there was next to no runtime difference between the two nodes (which was to be expected, since they share a common processor architecture), the only implementation difference emerged in terms of available RAM; in the Tiger implementation for the MicaZ node we had to move six of the lookup tables from RAM to ROM in order to get be-

[2]http://www.groestl.info/implementations.html

[3]http://www.cs.technion.ac.il/∼biham/Reports/Tiger/

[4]http://www.memsic.com/wireless-sensor-networks/

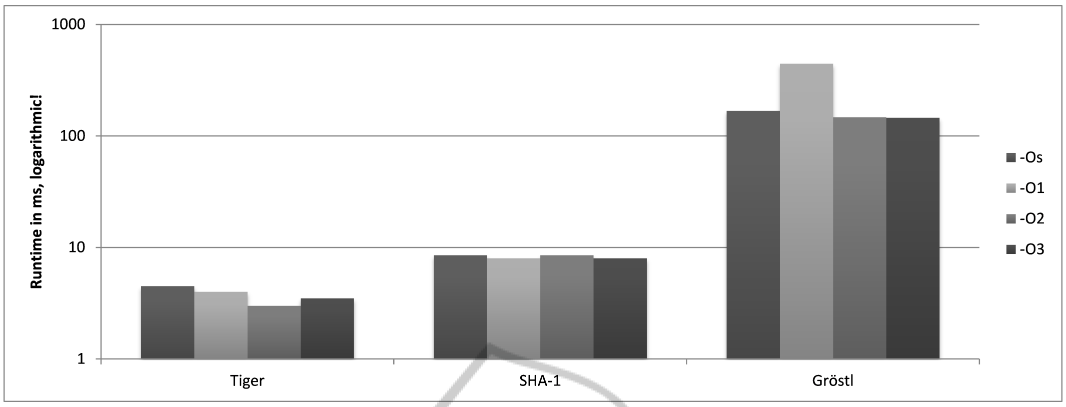

Figure 1: Runtime in milliseconds per 64-Byte Input block on the Iris Mote *(note the logarithmic scale of the ordinate)*.

low the 4KB limit. On the Iris platform, moving two tables to ROM sufficed.

The motes ran TinyOS[5] and were programmed using NesC, with the ncc compiler version 1.2.4 and the underlying avr-gcc 4.1.2. Our main goal, as already mentioned in Section 1, was not to find the perfect optimization of the algorithms for a given architecture, but rather to (a) find out how much effort has to be put in modifying reference implementations to run on embedded 8-bit microcontrollers, (b) how much performance can be expected by these implementations, and, finally, (c) how different algorithms with different internal structures react to compiler optimizations.

Both the SHA-1 and Grøstl reference code could be ported without any large modifications, since they use at most 32-bit datatypes that are supported on our target platforms, through the use of `unsigned long long`. Tiger, on the other hand, had to be largely rewritten; all the 64-bit operations had to be separated into a high and a low part, the tables had to be split into eight tables of 256 32-bit words each, and carry/borrow propagation had to be taken care of for each operation individually. Listing 4 gives a short excerpt of how the round function looks like after this treatment (see also Listing 3 for the original 64-bit-based implementation).

Table 1 and Figure 1 give the time needed to process a single 64-byte block for each hash algorithm on the Iris mote, with four different ncc compile switches, from -Os ("optimize for size") up to -O3 ("optimize yet more"). This time was obtained by performing multiple runs with inputs of varying length, resulting in a different number of blocks to be processed. By subtracting two runtime values with a consecutive number of blocks, adding two such differences, and dividing by two, we tried to eliminate any fixed setup or preprocessing time needed by the

[5]http://www.tinyos.net/

Listing 4: Modified Tiger Kernel.

```
//c ^= x;
cz_low ^= x_low; atomic cz_high ^= x_high;

//a -= t1[c_0] ^ t2[c_2] ^ t3[c_4] ^ t4[c_6];
az_low -=
  pgm_read_dword(&(t1_low[(tbyte)(cz_high)])) ^
  pgm_read_dword(&(t2_low[((tbyte*)(&cz_high))
       [2]])) ^
  t3_low[(tbyte)(cz_low)] ^
  t4_low[((tbyte*)(&cz_low))[2]];

az_high -=
  pgm_read_dword(&(t1_high[(tbyte)(cz_high)])) ^
  pgm_read_dword(&(t2_high[((tbyte*)(&cz_high))
       [2]])) ^
  t3_high[(tbyte)(cz_low)] ^
  t4_high[((tbyte*)(&cz_low))[2]] ;

//fix the borrow
if(az_high > tempr) (az_low)--;
```

algorithms.
Note that the ordinate of Figure 1 is given in logarithmic scale to account for the big time difference between the algorithms. In this first preliminary run we were not so much concerned with the individual switches triggered by these options, but more with their overall impact on performance and code size. Which algorithms are affected by these switches? And if they are affected, how? The results we obtained were consistent with our earlier implementations on different hardware platforms (Koschuch et al., 2012); especially noteworthy are two observations: although the most complex algorithm from a code point-of-view, Tiger outperformed even SHA-1 by a factor of 2. And Grøstl's runtime is very sensitive to different compiler switches, whereby the Tiger and SHA-1 implementations are almost unaffected by them in terms of performance.

Table 2 gives an overview of the RAM needed by

Table 1: Time in Milliseconds per 64-byte Input Block on the Iris Mote.

	Tiger	SHA-1	Grøstl
-Os	4.5	8.5	167.5
-O1	4	8	444.5
-O2	3	8.5	147.5
-O3	3.5	8	145.5

Table 2: RAM Usage in Bytes on the Iris Mote.

Tiger	SHA-1	Grøstl
5,332	1,512	1,667

Table 3: ROM Usage in Bytes on the Iris Mote.

	Tiger	SHA-1	Grøstl
-Os	53,388	9,808	18,852
-O1	16,970	5,110	13,620
-O2	16,424	4,952	13,840
-O3	55,180	16,640	21,268

the different implementations on the Iris mote. Since the RAM usage is completely unaffected by the various compiler switches, only one row is given there. Tiger was the only algorithm that, due to the 8KB lookup tables, did not fit into the RAM without further modifications. So four of the tables (six for the MicaZ mote) were moved to the ROM using compiler directives.

Finally, Table 3 details the ROM usage (in bytes) of the algorithms on the Iris mote. It is evident that Tiger is by far the largest of the three algorithms examined, followed by Grøstl and SHA-1. In contrast to the performance figures, the compiler switches affect all three algorithms when it comes to the amount of allocated ROM, with the -O2 option generally being the best choice (apart from a minor deviation in the Grøstl case) for the hash functions tested.

4 CONCLUSIONS AND OUTLOOK

In this work we performed a preliminary performance analysis of three selected hash functions on Iris and MicaZ sensor nodes. Our main goal was to give an estimate of the effort that is to be expected when porting a reference implementation of a hash algorithm to an 8-bit microcontroller, and how much gain in terms of performance and memory requirements can be achieved by utilizing simple compiler switches.

Our first results look promising, with the additional surprise of the ported 64-bit Tiger hash to be the best performing, albeit most memory and porting effort demanding, algorithm of the tested set.

Our next steps will be to perform the same analysis with the other SHA-3 final candidates as well as the new SHA-3 (KECCAK) itself, and also compare the results obtained here with results from our previous works, in order to give potential implementers a foundation for an informed decision when having to select a particular hash function for the use in a new (or old, or extended) sensor network.

REFERENCES

Anderson, R. and Biham, E. (1996). Tiger: A fast new hash function. In *Fast Software Encryption, Third International Workshop Proceedings*, pages 89–97. Springer-Verlag.

Bertoni, G., Daemen, J., Peeters, M., and van Assche, G. (2011). The KECCAK reference version 3.0. Technical report, STMicroelectronics and NXP Semiconductors.

Eastlake, D. and Jones, P. (2001). RFC3174 - US Secure Hash Algorithm 1 (SHA1). Technical report.

Gauravaram, P., Knudsen, L. R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., , and Thomsen, S. S. (2011). Grøstl - a SHA-3 candidate. Technical report.

Koschuch, M., Hudler, M., and Krüger, M. (2012). Hashalgorithms for 8051-based sensornodes. In *DCNET 2012 - International Conference on Data Communication Networking Proceedings of DCNET and OPTICS 2012*, pages 65–68. SciTePress - Science and Technology Publications.

Manuel, S. (2011). Classification and generation of disturbance vectors for collision attacks against SHA-1. *Des. Codes Cryptography*, 59(1-3):247–263.

Mendel, F. and Rijmen, V. (2007). Cryptanalysis of the tiger hash function. In *Proceedings of the Advances in Crypotology 13th international conference on Theory and application of cryptology and information security*, ASIACRYPT'07, pages 536–550, Berlin, Heidelberg. Springer-Verlag.

Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (2001). *Handbook of Applied Cryptography*. CRC Press, 5th printing edition.

National Institute of Standards and Technology (2001). FIPS 197, Advanced Encryption Standard (AES), Federal Information Processing Standard (FIPS), Publication 197. Technical report, DEPARTMENT OF COMMERCE.

National Institute of Standards and Technology (2012). FIPS 180-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-4. Technical report, DEPARTMENT OF COMMERCE.

Wang, L. and Sasaki, Y. (2010). Finding preimages of tiger up to 23 steps. In *Proceedings of the 17th international conference on Fast software encryption*, FSE'10, pages 116–133, Berlin, Heidelberg. Springer-Verlag.