SHA-3 on ARM11 processors

Peter Schwabe[1], Bo-Yin Yang[1], and Shang-Yi Yang[2] ⋆

[1] Institute of Information Science and Research Center for Information Technology Innovation
Academia Sinica,
128 Section 2 Academia Road, Taipei 115-29, Taiwan
`peter@cryptojedi.org, by@crypto.tw`

[2] Department of Electrical Engineering
National Taiwan University
1, Section 4, Roosevelt Road, Taipei 10617, Taiwan
`ilway25@crypto.tw`

Abstract. This paper presents high-speed assembly implementations of the 256-bit-output versions of all five SHA-3 finalists and of SHA-256 for the ARM11 family of processors. We report new speed records for all of the six implemented functions. For example our implementation of the round-3 version of JH-256 is 35% faster than the fastest implementation of the round-2 version of JH-256 in eBASH. Scaled with the number of rounds this is more than a 45% improvement. We also improve upon previous assembly implementations for 32-bit ARM processors. For example the implementation of Grøstl-256 described in this paper is about 20% faster than the `arm32` implementation in eBASH.

Keywords: SHA-3, ARM processors, software implementation

1 Introduction

This paper describes assembly implementations of all SHA-3 finalists for the ARM11 family of processors. These implementations set new speed records for all of the SHA-3 finalists; for comparison we implemented SHA-256 for the ARM11. ARM11 processors can be found in many smartphones by Apple, Samsung, HTC, Nokia and others. Some high-end smartphones now feature the newer Cortex-A8 and Cortex-A9 CPUs, but still many smartphones with ARM11 processors are shipped today.

One would expect that compilers are able to transfrom well-written C implementations of the SHA-3 finalists into high-performance code for this relatively simple 32-bit RISC architecture. However, in some cases we were able to improve upon C implementations by more than a factor of 2.

We submitted the software described in this paper to eBASH [7] for public benchmarking All benchmarks reported in this paper are from the SUPERCOP benchmarking suite. We put all software described in this paper into the public domain to maximize reusability.

Notation. We use $\oplus$ for a bitwise xor, $\vee$ for bitwise or and $\wedge$ for bitwise and.

Acknowledgements. We thank the CyanogenMod and GAOSP teams for their work on alternative firmware for Android smartphones. Their work made it possible to install Debian GNU/Linux in a chroot environment on a Samsung GT i7500 Galaxy phone and use it as development and benchmarking platform. Furthermore we thank all the authors of SHA-3-candidate implementations who published their software and included it in eBASH [7].

2 The ARM11 processor family

The ARM11 family of microprocessors has been introduced by ARM in 2002 and is the only implementation of the ARMv6 architecture. The most widely used processor of this family is the ARM1136, others are the

⋆ This work was supported by the National Science Council, National Taiwan University and Intel Corporation under Grant NSC99-2911-I-002-001 and 99-2218-E-001-007, and by the Academia Sinica Career Award. Part of the work was done while the first author was employed by National Taiwan University. Permanent ID of this document: `fd40a1bbff5e17e661dfed3102dbf1fa`. Date: November 25, 2011

ARM1156 and the ARM1176. We developed and benchmarked the software described in this paper on an ARM1136 processor, more specifically on a Samsung GT i7500 Galaxy smartphone containing a Qualcomm MSM7200A chip released in 2007. The characteristics of ARM11 processors are described in detail in the ARM11 technical reference manuals [3,1,2,4,5]. In the following we give a summary of the features that are most relevant to the implementations described in this paper.

ARM11 processors have a 32-bit instruction set and 16 architectural 32-bit integer registers. One register is used as stack pointer, one as program counter, so 14 registers are freely usable. Instructions are issued in order, one instruction per cycle. The arithmetic instructions relevant to the implementations described in this paper have a latency of 1 cycle, the result of an instruction can thus be used as input to the next instruction without latency penalty. Access to memory is cached with cache sizes between 4 KB and 64 KB. Loads from cache have a latency of 3 cycles.

The instruction set is a standard RISC load-store instruction set except for two features: free shifts and rotates and loads and stores of more than 32 bits.

Free shifts and rotates. All arithmetic instructions have three operands, the output does not necessarily overwrite one of the inputs. Additionally, the second input operand can be shifted or rotated by arbitrary distances provided as immidiate value or through a register. These shifts or rotates as part of arithmetic instructions do not decrease throughput or increase latency of the instruction, they are essentially for free. However, the shifted or rotated input value is required one stage earlier in the pipeline than a non-shifted input. Therefore, using the output of one instruction as shifted or rotated input to the next instruction imposes a penalty of one cycle.

Load and store double and multiple. The ARMv6 instruction set contains load and store instructions that move more than 32 bits between memory and registers. More specifically, the `strd` instruction stores 64 bits from two consecutive registers (e.g., `r0` and `r1`) to a 64-bit memory location, the `ldrd` loads 64 bits from memory into two consecutive registers. Some additional restrictions apply for these 64-bit load and store instructions:

– The first register argument has to be an even register (`r0`, `r2`, `r4`, ...),
– the instructions do not support all addressing modes that their 32-bit counterparts support, in particular they do not support shifted register offsets (documentation is very misleading here, see, e.g. [3, Section 16.11] says that performance of `ldrd` and `strd` depends on the shift distance of the register offset), and
– they take one memory cycle only if the memory location is 8-byte aligned, otherwise they take 2 memory cycles.

For details also see [3, Section 16.11]. The ARMv6 architecture also supports loads and stores of more than 64 bits in one instruction (`ldm` and `stm` instructions). Addressing modes are even more limited than for `strd` and `ldrd`. They need as many memory cycles as a corresponding sequence of 64-bit loads or stores and thus yield better performance only in very special cases that we were not able to exploit in our implementations.

Accessing the cycle counter. Access to the 32-bit cycle counter is only possible from kernel mode, for example using the following code:

```
unsigned int c;
asm volatile("mrc p15, 0, %0, c15, c12, 1" : "=r"(c));
```

In a posting to the eBATS mailing list `ebats@list.cr.yp.to` from August 12, 2010, Bernstein publicized code for a kernel module that gives access to the cycle counter on ARM11 devices through the Linux device file `/dev/cpucycles4ns`. The SUPERCOP benchmarking suite [7] supports cycle counts through this device file; we use SUPERCOP for all benchmarks.

3 Blake

The full specification of Blake is given in [6]. We only briefly recall the structure of the computationally most expensive part, the compression function. The inputs to the Blake-256 compression function are a chaining value of 8 32-bit words $h_0, \dots, h_7$, a message block of 16 32-bit words $m_0, \dots, m_{15}$, a salt of 4 32-bit words $s_0, \dots, s_3$. and counter consisting of 2 32-bit words t_0, t_1 The output is a new chaining value consisting of 8 32-bit words. The compression consists of 3 main steps:

- An initialization expands the 14 words of chaining value, salt and counter to a 16-word state $(v_0, \dots, v_{15})$.
- The 16-word state is transformed through 14 rounds. Each round consists of 8 evaluations of a function G, which modifies 4 words of the state in place and takes as additional inputs 2 words of the message block and two out of a set of 16 constants $c_0, \dots, c_{15}$ The total of $14 \cdot 8 = 112$ evaluations of G are the main computation of the Blake-256 hash function. Each evaluation of G requires 6 32-bit word additions, 6 32-bit xor operations, and 4 rotations of 32-bit words by 16,12,8, and 7 bits, respectively. The 8 evaluations of G can be seen as 2 blocks of 4 evaluations each; evalutions of G in each of the two blocks are independent and can be swapped or interleaved.
- The finalization uses 24 32-bit xor operations to map $h_0, \dots, h_7, s_0, \dots, s_3$, $v_0, \dots, v_{15}$ to a new chaining value $h'_0, \dots, h'_7$.

3.1 Implementation details

The 6 additions and 6 xors in each evaluation of G add up to a total of $112 \cdot 12 = 1344$ arithmetic instructions throughout the 14-round main loop. This corresponds to a $1344/64 = 21$ cycles lower bound for Blake-256. This lower bound is ignoring costs for loads of message words, costs for loads of constants, spills of state words, as the 16 state words do not fit into the usable 14 registers, and overhead from the initalization and finalization phase. Furthemore it assumes that all rotations can be carried out for free in the second argument of additions or xors.

To obtain the speed of 33.93 cycles per byte for long messages we applied two optimization techniques. First we manage to merge (almost) all rotations with arithmetic instructions and second we carefully reschedule code to reduce the number of spills.

Removing rotations. As explained in Section 2, the second argument of arithmetic instructions can be shifted or rotated arbitrarily. This shift or rotation does not cost any additional cycles, if the shifted value is not the output of the directly preceding instruction. In other words, the combination of an arithmetic operation $\odot$ with a rotation by n of the form

$$a \leftarrow b \odot (c \ggg n)$$

costs the same as $\odot$ without the rotation (but imposes an additional scheduling constraint). The 4 rotations used inside the function G are not of this form, they rotate the result of an arithmetic instruction instead of one of the inputs:

$$a \leftarrow (b \odot c) \ggg n).$$

This can easily be decomposed into two instructions

$$a \leftarrow (b \odot c), \text{ and}$$
$$a \leftarrow a \ggg n.$$

Our implementation instead only computes the first of the two instructions and rotates a by n the next time a is used as input. We consistently apply this technique to all variables. This means that we keep track of the implicit rotation distances for each variable and apply this rotation whenever the variable is used as an input. Very soon this will lead to the case that both inputs to an arithmetic instruction need to be rotated, i.e.,

$$a \leftarrow (b \ggg n_1) \odot (c \ggg n_2).$$

In this case we compute

$$a \leftarrow b \odot (c \ggg (n_2 - n_1))$$

and set the implicit rotation distance of a to n_1. Note at this point that all rotation distances are constants; the value $n_2 - n_1$ is computed at compile time.

In principle we can merge all rotations with arithmetic instructions in this way; the only restriction is that implicit rotation distances of variables must be invariant across different iterations of loops. We fully unrolled the 14 rounds of the Blake-256 compression function, so we only need to make the implicit rotations distances explicit by actual rotations at the very end of the loop. Out of the 16 state words, 12 end up with a implicit rotation distance of 0, so we only need 4 dedicated rotation instructions for the whole compression function.

Reducing spills. The 16 32-bit words $v_0, \ldots, v_{15}$, the 16 constants $c_0, \ldots, c_{15}$, and the 16 32-bit message words clearly do not fit into the 14 usable 32-bit registers. Even worse, not even the 16 state words can be kept in registers, the compression function thus requires loads and stores. To keep the number of loads and stores low we do the following:

– Before entering the compression loop we put the 16 constants on the stack so that they can be accessed through the stack pointer. We could also access them through offsets to the program counter, but our code is too long to access constants from any position in the code through the allowed 8-bit offset.
– In each iteration of the compression loop we place the 16 message words on the stack. This saves one register containing the pointer to the message block, furthermore we can easily convert from big-endian to little-endian encoding by loading message blocks in big-endian mode once and storing them in little endian mode on the stack.
– We partition the set of state words into low words $v_0, \ldots, v_7$ and high words $v_8, \ldots, v_{15}$, we keep the low state words in registers throughout the whole computation. Each evaluation of G transforms 2 low state words and 2 high state words, so we can compute G with 12 arithmetic instructions, 2 loads of high words, 2 stores of high words, 2 loads of message words, and 2 loads of constants – a total of 20 instructions.
– We replace the two loads of high 32-bit words by one 64-bit load, this reduces the number of instructions per evaluation of G to 19. For this to work we need to make sure that previous stores of these two words store them to consecutive memory locations.
– We reorder evaluations of G in a way that allows us to reuse the output high state words as input to the next evaluation where possible.

In total the 112 evaluations of G take 2044 instructions, 18.25 instructions per evaluation of G. These instructions are carefully scheduled to hide all latencies and thus contribute $2044/64 = 31.94$ cycles/byte to the total cost.

4 Grøstl

We only recall the computationally intensive part of Grøstl-256, the compression function. The full specification of Grøstl is given in [13]. The compression function maps a 512-bit state h_{i-1} and a 512-bit message block m_i to a 512-bit state h_i. This compression uses two 512-bit permutations P and Q and computes

$$h_i = P(h_{i-1} \oplus m_i) \oplus Q(m_i) \oplus h_{i-1}.$$

The design of the permutations P and Q is inspired by AES, the main change is the size of the state which for Grøstl is an 8×8 byte matrix instead of the 4×4 matrix for AES. This change allows to make more efficient use of 64-bit architectures. The permutations P and Q are very similar, both transform the state in 10 rounds each round consists of the operations AddRoundConstant, SubBytes, ShiftBytes, and MixBytes. The SubBytes and the MixBytes operations are the same for P and Q; SubBytes is the byte substitution also used in AES. The AddRoundConstants and MixBytes operations are slightly different in P and Q; this requires separate implementations of P and Q but has no effect on the implementation techniques. Everything explained in the following is valid for both P and Q.

4.1 Implementation details

The designers of Grøstl recommend in [13, Section 8.1.3] to use a lookup-table based implementation for 32-bit processors that do not support 128-bit vector instructions. The idea is to compute the SubBytes, ShiftBytes, and MixBytes operations columnwise, where the computation of each column consists of

– 8 table lookups of 64-bit values from 8 tables $T_0, \ldots, T_7$ of size 2 KB each, each lookup indexed by one byte of the state, and
– 7 64-bit xors of these 8 values to obtain the new the 64-bit column.

This is the same idea as the lookup-table-based approach for AES described in [11, Section 5.2].

For a 32-bit implementation all operations on 64-bit values need to be split into 2 operations on 32-bit values; a small benefit is that the total size of the tables can be halved because entries in tables $T_1, \dots, T_7$ are simply rotations of values in table T_0. ARM addressing modes do not allow to load from a base address (pointer to the tables) plus a shifted offset register value. Each of these column computations thus requires 8 byte extractions (for example single-byte loads), 8 additions of the table pointer to a (shifted) byte offset, 16 32-bit loads, 14 xors, and 2 32-bit stores of the computed new state column. These final stores are required because the complete state does not fit into registers and allows to perform byte extraction through single-byte loads. This total of 48 instructions, performed 8 times per round, over 10 rounds in both P and Q yields a lower bound of $(48 \cdot 8 \cdot 10 \cdot 2)/64 = 120$ cycles per byte, ignoring the cost of AddRoundConstants.

This approach is what the `arm32` assembly implementation for 32-bit ARM processors by Wieser (included in SUPERCOP since version 20110914) does. With about 140 cycles/byte it comes remarkably close to the lower bound if we consider cost for AddRoundConstants and loop overhead.

At a speed of about 110 cycles per byte for long messages, our implementation improves upon this implementation by more than 20%. The main reason is that it makes use of 64-bit table lookups.

64-bit lookups. Instead of performing the table lookups with 32-bit loads we use the `ldrd` instruction to perform 64-bit lookups. As the result of such a lookup is returned in two 32-bit registers, we do not need larger tables, "rotation by 32" is free, the tables only need 8 KB of storage. With this improvement the computation of one column only requires 8 byte lookups, 8 additions of the table base pointer to the shifted byte offsets, 8 64-bit lookups, 14 xors, and 1 64-bit store. These 39 instructions yield a lower bound of $(39 \cdot 8 \cdot 10 \cdot 2)/64 = 97.5$ cycles per byte, again ignoring the cost of the AddRoundConstants operation and loop overhead.

Interleaved tables. If the 4 lookup tables of size 2 KB each were layed out in memory one after the other, we would need to either keep 4 table addresses in memory or add constant offsets of 2048, 4096 or 6192 to 75% of the lookup addresses. The ARM addressing modes for the `ldrd` instruction support adding constant offsets to a base address, but these offsets must not be larger than 8 bytes (signed). We circumvent this problem by interleaving 64-bit entries of the 4 tables in memory, two consecutive 64-bit entries of the same table thus start at addresses that are 32 bytes apart. The bytes extracted from the state thus need to be shifted by 5 instead of 2 to serve as lookup offsets.

5 JH

The JH construction may be considered a modified Sponge. The full specification of JH-256 is given in [14]. The central part of the JH hash functions is the compression function. This compression function transforms a 1024-bit state and a 512-bit input block into a 1024-bit state as follows:

- Xor the input block into the first half of the state,
- apply a block cipher E_8 with a fixed key (expanded to hash-function round constants) to the state, and
- xor the input block into the second half of the state.

The speed-critical part of this compression is the application of the block cipher E_8 which consists of 42 rounds of a substitution-permutation network designed for efficient bitsliced implementations using 128-bit or 256-bit vector registers. In bitsliced implementation using 128-bit vector registers the state is decomposed into 8 128-bit vectors. Each round operates on these 128-bit vectors of the state and 2 128-bit round constant. One round consists of the following operations:

- 2 applications of the `Sbox` operation, an in-place transformation of 4 state vectors involving 1 round constant (see Listing 1),
- 1 application of the `L` operation, an in-place transformation of 8 state vectors (see Listing 1), and
- swapping of adjacent bit blocks in 4 of the state vectors. The size of these bit blocks is $2^{(i \bmod 7)}$ in round i, i.e., in rounds 0, 7, 14, 21, 28, and 35 swap adjacent bits; in rounds 1, 8, 15, 22, 29, and 36 swap adjacent blocks of 2 bits; in rounds 2, 9, 16, 23, 30, and 37 swap adjacent blocks of 4 bits; and so on; and in rounds 6, 13, 20, 27, 34, and 41 swap adjacent blocks of 64 bits.

Listing 1 The `Sbox` and the `L` operations of the JH compression function

```
#define Sbox(v0,v1,v2,v3,rcst)   \
  v3  = ~(v3);                   \
  v0 ^= ((~(v2)) & (rcst));      \
  tmp0 = (rcst) ^ ((v0) & (v1)); \
  v0 ^= ((v2) & (v3));           \
  v3 ^= ((~(v1)) & (v2));        \
  v1 ^= ((v0) & (v2));           \
  v2 ^= ((v0) & (~(v3)));        \
  v0 ^= ((v1) | (v3));           \
  v3 ^= ((v1) & (v2));           \
  v1 ^= (tmp0 & (v0));           \
  v2 ^= tmp0;
```

```
/*The MDS transform*/
#define L(v0,v1,v2,v3,v4,v5,v6,v7) \
  (v4) ^= (v1);                    \
  (v5) ^= (v2);                    \
  (v6) ^= (v0) ^ (v3);             \
  (v7) ^= (v0);                    \
  (v0) ^= (v5);                    \
  (v1) ^= (v6);                    \
  (v2) ^= (v4) ^ (v7);             \
  (v3) ^= (v4);
```

5.1 Implementation details

For the ARM11, each operation on 128-bit vectors needs to be decomposed into 4 operations on 32-bit values. The 19 bit-logical operations in the `Sbox` operation (ignoring negations) and the 10 bit-logical operations in the `L` operation thus yield a lower bound on JH performance of $42 \cdot 4 \cdot (2 \cdot 19 + 10) = 8064$ cycles per block or 126 cycles per byte. This ignores costs for loads and stores, loop overhead and for the bit-block swapping. We now describe the optimization techniques that we applied to obtain the performance of 156.43 cycles per byte for long messages.

Partial unrolling. Fully unrolling the compression function would result in code larger than 32 KB, more than the instruction cache on by far most ARM11 processors. Instead we unroll 7 rounds, a choice that comes from the block sizes in the swapping step. This way we keep the size of the compression function comfortably below 8 KB. Instead of using a round counter in a register we place a sentinel value at the end of the round-constant table and exit the loop when this sentinel value is read.

Loop reordering. As we decomposed 128-bit vectors into 4 32-bit values, the compression function has two loops, one loop over the 42 rounds, another one over the 4 vector chunks. If we ignored the bit-block swapping step at the end of each round these loops could permute, but in the last two out of 7 rounds (bit block sizes of 32 and 64) registers at different positions in the vector communicate. The obvious way of ordering the loops is thus an outside (partially unrolled) loop of the rounds and then a loop of length 4 (unrolled) inside each round. Each of these iterations of the inside loop operates on different 8 32-bit state values, so with the 14 available registers this solution requires 32 load and 32 store operations per round.

However, by swapping the order of the two loops in the first 5 rounds of a 7-round block, these 32 load-save pairs can avoided. Only the last two rounds of a 7-round block require frequent loads and stores. We then reverse the processing order in the last round of a 7-round block, saving a further 4 stores and 4 loads between this round and the previous round, as well as this round and the next block.

For the state we use a memory layout that allows us to perform all loads and stores of the state in 64-bit, halving the number of operations. This layout also allows us to use 64-bit loads and stores in the xor sequences at the beginning and the end of the compression function.

Optimized `Sbox` operation. The ARMv6 instruction set allows to combine a logical and with a negation of one of the arguments (`bic` instruction). Most negations inside the `Sbox` operation are exactly of this type but the negation in the first line (cmp. Listing 1) can not easily be eliminated. However, in 2/7 of all uses of `Sbox` the negated value is the output of a swap operation. In these cases the negation can be combined with a logical and in the swap operation (see below).

Efficient bit-block swapping. The swapping of 16-bit blocks is a rotation that is free if we merge it with subsequent uses of the value in arithmetic instructions. Swapping 8-bit blocks can be done in just one instruction using the `rev16` instruction. Swapping of adjacent bit blocks of size 1, 2, and 4 is not that straightforward but can still make use of free shifts. For example swapping adjacent bits in a register x uses a mask $m =$ `0xaaaaaaaa` in another register and three instructions as follows:

$$\begin{aligned} t &\leftarrow m \wedge (x \ll 1) \\ x &\leftarrow m \wedge x \\ x &\leftarrow t \oplus (x \gg 1) \end{aligned}$$

Note that in one out of four swaps, the `bic` instruction (which negates the shifted second operand) is substituted for the logical and to save the negation at the beginning of the `Sbox` operation (see above).

6 Keccak

The Keccak hash function uses a sponge construction. The message is absorbed into a 1600-bit state in r-bit blocks, for Keccak with 256-bit output the SHA-3 submission specifies $r = 1088$ [9, Section 2]. Each absorbtion of an r-bit block consists of two steps:

– Xor the message block with the first r bits of the state, and
– transform the state through the 1600-bit permutation Keccak-f[1600].

After all blocks of the (padded) message have been absorbed, the hash value is extracted from the state in a squeeze operation. The speed-critical part of hashing long messages is the absorbtion and in particular the Keccak-f[1600] transformation. A full specification of Keccak is given in [8] and in [9].

The Keccak-f[1600] transformation considers the state a $5 \times 5 \times 8$ byte cuboid or a 5×5 matrix of 64-bit lanes. The transformation is performed in 24 rounds, each round consists of the following steps:

– xor the 5 lanes of each column to obtain values $b_0, \dots, b_4$,
– compute 5 values $c_0, \dots, c_4$, each as the xor of one of the b_i with another of the b_i rotated by 1,
– compute the updated state columnwise, for each column
  • pick up 5 state lanes diagonally,
  • xor each of these state lanes with a different c_i (one lane of the whole state is additionally xored with a round constant),
  • rotate each of these lanes by a different fixed distance,
  • compute each lane of the updated column by negating one of the lanes, computing the logical and with another lane and then xoring the result to a third lane.

6.1 Implementation details

For 32-bit architectures without vector registers the designers of Keccak suggest the technique of bit interleaving [10, Sections 1.4 and 2.2]. The idea is to collect all bits on even positions of a 64-bit lane in one 32-bit register and all bits at odd positions in another 32-bit register. This requires interleaving every 64-bit chunk of the message, but allows to perform all rotations as rotations of 32-bit words which is particularly efficient for the ARM11 as such rotations are essentially free.

All logical operations on 64-bit lanes need to be carried out as two 32-bit operations. Computation of $b_0, \dots, b_4$ thus takes 40 xors, computation of $c_0, \dots, c_4$ takes another 10 xors. The computation of each column requires 10 xors with a constant, and 10 xors and 10 ands (with negation) of 32-bit half-lanes. A lower bound on the number of cycles for all 24 rounds (ignoring rotations and negations) can thus be derived as $24 \cdot (40 + 10 + 5 \cdot (10 + 10 + 10)) = 4800$, this corresponds to $4800/136 = 35.294$ cycles per byte, an additional cost of about 4 cycles per byte is required for the interleaving of the 64-bit message chunks.

The Keccak designers recommend the `simple32bi` implementation as starting point for implementations targeting 32-bit architectures. This implementation is included in SUPERCOP, it uses the interleaving technique and provides best performance on an ARM11 of all Keccak implementations benchmarked in eBACS; the `armasm` implementation by Ronny van Keer fails the tests of the SUPERCOP benchmarking suite.

Our assembly implementation requires 71.73 cycles per byte for long messages, 17.5% faster than the simple32bi implementation but quite a bit slower than the lower bound derived above. The main reason that neither the `simple32bi` nor our implementation get closer to the lower bound is a the overhead from frequent loads and stores of parts of the large state. There are mainly two reasons why our implementation outperforms previous implementations: we manage to reduce the number of load and store instructions and we merge more rotations with arithmetic instructions.

Reducing the number of loads and stores. Updating the state columnwise as described above uses 5 blocks of operations on 64-bit lanes or 10 blocks of computations on 32-bit half-lanes. A straight-forward implementation loads 5 half-lanes per block, then loads 5 32-bit c_i values, performs arithmetic instructions and then uses 5 store instructions to update the column half-lanes. With all rotations merged into arithmetic instructions each block only uses 15 arithmetic instructions, loads and stores thuscontribute a 50% overhead.

With an appropriate memory layout of the state we reduce the number of store instructions to 3 by using 2 64-bit stores. Furthermore we reorganize the 10 blocks of computations such that c_i in registers can be reused across blocks as much as possible. The initial computation of $b_0, \ldots, b_4$ (in 10 32-bit words) uses 64-bit loads of the state.

Removing rotations. We use the same techniques as for Blake to merge almost all rotations with arithmetic instructions. Unrolling all 24 rounds would result in excessively large code, instead we perform the first round, then a loop of length 11 around 2 rounds and then the final round. All rounds but the last produce shifted state half-lanes, the shift distances are the same for all rounds. All rounds but the first perform shifts of state half-lanes whenever they are used as input to an arithmetic instruction. In order to keep shift distances invariant over iterations of the loop we need to perform 10 dedicated shifts in the computation of $b_0, \ldots, b_4$.

7 Skein

The idea of Skein is to build a hash function out of the tweakable block cipher Threefish. The full specification of Skein is given in [12]. We only briefly review the Threefish block cipher, the most important component of Skein. Threefish as used in Skein uses a 512-bit state that is transformed in 72 rounds. Each round consists of 4 so-called MIX operations. After every 4 rounds the 64-bit words of the state are permuted and a key (round constant) is injected. An additional round constant is injected before the first round. The key injection loads 8 out of 9 64-bit extended-key words, it loads 2 out of 3 64-words of extended tweak value, adds each of the 8 extended-key words to one of the state words, adds each of the 2 tweak values to one of the state words, and in i-th key injection adds i to one of the state words.

7.1 Implementation details

Each MIX operation takes 2 64-bit integers x and y as inputs. It computes 2 64-bit output integers u and v as

$$u = (x + y) \bmod 2^{64}, \text{ and } v = u \oplus (y \ggg R),$$

where R is a round-dependent constant. On the 32-bit ARMv6 architecture we need to split each 64-bit word into two 32-bit chunks. With $x = (x_0, x_1)$ and $y = (y_0, y_1)$ we can compute $u = (u_0, u_1)$ with one addition and one addition with carry. Xoring u with the shifted y can be done in 4 instructions as follows:

$$\begin{aligned} t &\leftarrow u_1 \oplus (y_0 \gg (32 - R)) \\ v_0 &\leftarrow u_0 \oplus (y_0 \ll R) \\ v_0 &\leftarrow v_0 \oplus (y_1 \gg (32 - R)) \\ v_1 &\leftarrow t \oplus (y_1 \ll R). \end{aligned}$$

From 72 rounds with 4 mix operations each we get 1728 instructions per block. From 19 key injections we would expect another $19 \cdot 11$ 64-bit additions, i.e. $19 \cdot 22 = 418$ instructions; this would together yield a lower bound from pure arithmetic instructions of $(1728 + 418)/64 = 33.53$ cycles per byte.

Since the 14 general-purpose registers cannot hold all the state words, the most important optimization required to come close to this bound is minimizing the number of loads and stores of the state words. We do this by rearranging the executation sequence to keep as many state words in memory as long as possible. This task is aided by a self-written program that traces the use of state words. Furthermore we show how key injection can be done with less than 11 additions and less than 10 64-bit loads.

Rearranging the execution sequence. Before entering the key injection, there are four rounds, each of which consists of four MIX operations followed by a permutation. As the common practice in unrolling loops, we mix the "permuted" state instead of doing mix and permutation in order. In this way, we can interleave the MIX functions of all these four rounds to achieve a much small number of loads and stores.

Consider for example the first 4 rounds. Let s_0 to s_7 be the state words. The orignial sequence of MIX operations is the following:

$$\begin{aligned}&\text{Round 1: MIX}(s_0, s_1),\ \text{MIX}(s_2, s_3),\ \text{MIX}(s_4, s_5),\ \text{MIX}(s_6, s_7)\\&\text{Round 2: MIX}(s_2, s_1),\ \text{MIX}(s_4, s_7),\ \text{MIX}(s_6, s_5),\ \text{MIX}(s_0, s_3)\\&\text{Round 3: MIX}(s_4, s_1),\ \text{MIX}(s_6, s_3),\ \text{MIX}(s_0, s_5),\ \text{MIX}(s_2, s_7)\\&\text{Round 4: MIX}(s_6, s_1),\ \text{MIX}(s_0, s_7),\ \text{MIX}(s_2, s_5),\ \text{MIX}(s_4, s_3).\end{aligned}$$

Performing the mix operations in this order would cause a lot of overhead from loading and storing state words. For example s_0 is used in the first MIX operation, then all other state words are used before s_0 is used again, this means that s_0 needs to be spilled. Similar statements hold for the other state values.

To keep as many state words in registers as long as possible, we reorder the MIX sequence as follows:

$$\begin{aligned}&\text{MIX}(s_0, s_1),\ \text{MIX}(s_2, s_3),\ \text{MIX}(s_2, s_1),\ \text{MIX}(s_0, s_3)\\&\text{MIX}(s_4, s_5),\ \text{MIX}(s_6, s_7),\ \text{MIX}(s_6, s_5),\ \text{MIX}(s_4, s_7)\\&\text{MIX}(s_4, s_1),\ \text{MIX}(s_6, s_3),\ \text{MIX}(s_6, s_1),\ \text{MIX}(s_4, s_3)\\&\text{MIX}(s_0, s_5),\ \text{MIX}(s_2, s_7),\ \text{MIX}(s_2, s_5),\ \text{MIX}(s_0, s_7).\end{aligned}$$

This order of MIX operations allows us load and store state words only every four MIX functions. Furthermore we interleave each round-constant injection with the preceeding MIX operations to eliminate some loads and stores.

Precomputing parts of the key injection. In the key injection, two of the 64-bit state words, s_5 and s_6, are modified by adding a 64-bit word of the extended key k and a 64-bit word of the extended tweak value t. In the i-th key injection this is done as

$$\begin{aligned}s_5 &\leftarrow s_5 + k_{(i+5) \bmod 9} + t_{i \bmod 3},\\ s_6 &\leftarrow s_6 + k_{(i+6) \bmod 9} + t_{(i+1) \bmod 3},\end{aligned}$$

where the additions are all modulo 2^{64}. In the intuitive implementation this takes 4 loads of 64-bit words and 4 64-bit additions, a total of $4 + 2 \cdot 4 = 12$ instructions.

In our implementation, we see that regarding to all possible values of i, there are only nine possibilities for $k_{(i+5) \bmod 9}+t_{i \bmod 3}$ and $k_{(i+6) \bmod 9}+t_{(i+1) \bmod 3}$, namely $\{k_0, k_3, k_6\}+t_1$, $\{k_1, k_4, k_7\}+t_2$ and $\{k_2, k_5, k_8\}+t_0$. We precompute these 9 values as $kt_{0,1}, kt_{3,1}, kt_{6,1}, kt_{1,2}, kt_{4,2}, kt_{7,2}$, etc. and perform only two 64-bit loads and two 64-bit additions, which saves 6 instructions per key schedule or about $(19 \cdot 6)/64 = 1.78$ cycles per byte.

8 SHA-256

For reference we also optimized SHA-256 in assembly. Unlike all SHA-3 candidates, the 256-bit state of SHA-256 fits into the available registers. Furthermore the design favors 32-bit architectures such as the ARMv6 and can make efficient use of the free rotations. For example a transformation like

$$(x \ggg a) \oplus (x \ggg b) \oplus (x \ggg c)$$

with constants a, b, and c can turn into

$$(x \oplus (x \ggg (b-a)) \oplus (x \ggg (c-a))) \ggg a$$

and thus make use of the same techniques we used for Blake to eliminate dedicated rotations. It is thus not surprising to see that SHA-256 outperforms all SHA-3 candidates on this platform; although in the case of Blake-256 only slightly.

9 Results and Comparison

This section presents performance results of our implementations and a comparison with the previously fastest implementation in eBASH (SUPERCOP version 20110914). All numbers in Tables 1–6 are cycles per byte as reported by the SUPERCOP benchmarking suite. For the benchmarks we removed several compiler options from SUPERCOP that are irrelevant for ARM11 (such as `-m64` or `-mcpu=ultrasparc`), with these compiler options the benchmarking results would not have been ready for the submission of this paper. We also added compiler options, specifically we added the flag `-no-schedule-insns` to various previously contained combinations of compiler flags. This flag is crucial for best performance of many C implementations; we also informed the eBACS editors about this observation, the changes are included in the most recent version of SUPERCOP (version 20111120). We will update the paper with benchmarks from SUPERCOP version 20111120. SUPERCOP in version 20110914 did not contain any implementation of the round-3 version of JH-256. We compare our results with the benchmarks of the round-2 version, note that this version is only using 35.5 rounds instead of the 42 of our implementation.

For the display of benchmarks we follow eBACS [7]. Specifically, for each message length we report the median of 45 measurements and the 25% and 75% quartiles. The value for "long" messages is extrapolated from measurements of 4096-byte messages and 2048-byte messages.

All measurements were performed on a Samsung GT I7500 Galaxy smartphone with a 528-MHz ARM1136 processor inside a Qualcomm MSM7200A chip. All code was compiled with gcc 4.4.5.

Other related work. The tables only compare with the fastest implementation in eBASH as of SUPERCOP version 20110914; there are many more slower implementations for all candidates submitted to eBASH that can be used on ARM11 CPUs. An assembly implementation of Skein supporting different output and state sizes targeting the ARM Cortex-A8 family of CPUs by Hitland can be found on https://github.com/unbounded/skein-arm. We ran the benchmarking tool shipped with this implementation on the same platform we used for eBASH benchmarks. For Skein with a 512-bit state size and 256-bit output size the tool reports a median of 48.78 cycles per byte, only slightly slower than our implementation. Parts of this implementation (using the NEON vector instruction set of the Cortex-A8 CPU) are included in the latest SUPERCOP suite, we encourage the author to also submit the non-NEON part of his implementation to eBASH for more extensive comparative benchmarking.

		long	**4096 bytes**	**1536 bytes**	**576 bytes**	**64 bytes**	**8 bytes**
this paper[a]	25%-quartile	33.96	34.95	36.62	40.95	94.88	509.00
	median	33.93	34.97	36.68	41.04	95.00	521.50
	75%-quartile	33.46	35.11	36.94	41.48	98.19	546.00
sphlib (ver. 3.0)[b]	25%-quartile	46.29	47.49	49.48	54.82	123.06	614.00
	median	46.29	47.49	49.48	55.09	123.06	614.50
	75%-quartile	46.44	47.74	49.52	55.97	123.13	719.16

[a] Compiled with gcc -funroll-loops -fno-schedule-insns -O2 -fomit-frame-pointer

[b] Compiled with gcc -mcpu=arm1136j-s -Os -fomit-frame-pointer -fno-schedule-insns

Table 1. Benchmark results of Blake-256

		long	**4096 bytes**	**1536 bytes**	**576 bytes**	**64 bytes**	**8 bytes**
this paper[a]	25%-quartile	110.11	113.17	118.26	131.87	301.88	1551.0
	median	110.16	113.24	118.44	131.95	301.94	1559.5
	75%-quartile	111.14	114.28	120.38	132.47	304.63	1604.13
arm32[b]	25%-quartile	139.89	143.48	149.52	165.86	374.63	1882.00
	median	140.17	143.62	149.65	165.86	374.63	1882.00
	75%-quartile	141.51	144.34	152.00	168.01	377.80	1998.38

[a] Compiled with gcc -funroll-loops -fno-schedule-insns -O2 -fomit-frame-pointer

[b] Compiled with gcc -mcpu=arm1020t -O -fomit-frame-pointer

Table 2. Benchmark results of Grøstl-256

		long	**4096 bytes**	**1536 bytes**	**576 bytes**	**64 bytes**	**8 bytes**
this paper[a]	25%-quartile	156.31	159.09	163.73	176.26	337.43	2733.50
	median	156.43	159.17	163.76	176.43	339.00	2746.00
	75%-quartile	157.33	159.78	163.95	179.02	341.59	2773.50
bitslice_opt32[b,c]	25%quartile	246.33	250.37	257.26	276.41	519.83	4155.88
	median	247.16	250.89	257.60	276.74	523.06	4195.00
	75%quartile	258.13	256.98	259.02	277.17	529.19	4232.50

[a] Compiled with gcc -funroll-loops -O3 -fomit-frame-pointer

[b] Compiled with gcc -funroll-loops -fno-schedule-insns -O2 -fomit-frame-pointer

[c] Round-2 version with only 35.5 instead of 42 rounds

Table 3. Benchmark results of JH-256

		long	**4096 bytes**	**1536 bytes**	**576 bytes**	**64 bytes**	**8 bytes**
this paper[a]	25%-quartile	71.50	74.31	77.46	87.80	180.59	1438.00
	median	71.73	74.45	77.50	87.91	182.22	1447.00
	75%-quartile	72.87	75.18	77.78	88.79	185.20	1486.38
simple32bi[b]	25%-quartile	86.86	90.22	93.89	106.33	216.06	1723.00
	median	86.95	90.28	93.92	106.54	217.75	1731.00
	75%-quartile	88.27	90.99	94.51	106.80	218.81	1752.50

[a] Compiled with gcc -funroll-loops -O3 -fomit-frame-pointer

[b] Compiled with gcc -mcpu=arm1136jf-s -O3 -fomit-frame-pointer -fno-schedule-insns

Table 4. Benchmark results of Keccak

		long	**4096 bytes**	**1536 bytes**	**576 bytes**	**64 bytes**	**8 bytes**
this paper[a]	25%-quartile	42.07	43.13	44.91	49.64	108.48	867.88
	median	42.10	43.16	45.04	49.67	108.73	874.13
	75%-quartile	43.53	43.93	45.83	50.05	113.11	893.50
sphlib-small (ver. 3.0)[b]	25%-quartile	94.48	96.32	99.39	107.58	209.11	1688.00
	median	94.57	96.40	99.41	107.65	210.64	1688.88
	75%-quartile	97.18	97.83	99.63	108.79	211.30	1698.50

[a] Compiled with gcc -funroll-loops -march=iwmmxt -O2 -fomit-frame-pointer

[b] Compiled with gcc -mcpu=arm1136jf-s -O3 -fomit-frame-pointer -fno-schedule-insns

Table 5. Benchmark results of Skein-256

		long	**4096 bytes**	**1536 bytes**	**576 bytes**	**64 bytes**	**8 bytes**
this paper[a]	25%-quartile	32.15	33.02	34.43	38.27	85.11	453.88
	median	31.95	33.05	34.45	38.29	85.11	462.25
	75%-quartile	31.14	33.39	34.61	38.46	89.34	489.38
sphlib (ver. 3.0)[b]	25%-quartile	39.19	40.19	41.86	46.33	103.31	522.50
	median	39.19	40.19	41.86	46.33	103.31	522.50
	75%-quartile	39.26	40.25	41.86	46.34	103.38	535.50

[a] Compiled with gcc -mcpu=arm1136j-s -O3 -fomit-frame-pointer
[b] Compiled with gcc -mcpu=arm1136jf-s -O2 -fomit-frame-pointer -fno-schedule-insns

Table 6. Benchmark results of SHA-256

References

1. ARM Limited. *ARM1156T2-S Technical Reference Manual, Revision: r0p4*, 2007. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0338g/DDI0338G_arm1156t2s_r0p4_trm.pdf.
2. ARM Limited. *ARM1156T2F-S TechnicalReference Manual, Revision: r0p4*, 2007. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0290g/DDI0290G_arm1156t2fs_r0p4_trm.pdf.
3. ARM Limited. *ARM1136JF-S and ARM1136J-S Technical Reference Manual, Revision: r1p5*, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0211k/DDI0211K_arm1136_r1p5_trm.pdf.
4. ARM Limited. *ARM1176JZ-S Technical Reference Manual, Revision: r0p7*, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0333h/DDI0333H_arm1176jzs_r0p7_trm.pdf.
5. ARM Limited. *ARM1176JZF-S Technical Reference Manual, Revision: r0p7*, 2009. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf.
6. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE (version 1.3). Revised submission to NIST, 2010. http://131002.net/blake/blake.pdf.
7. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to.
8. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference (version 3.0), 2011. http://keccak.noekeon.org/Keccak-reference-3.0.pdf.
9. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak sha-3 submission (version 3). Revised submission to NIST, 2011. http://keccak.noekeon.org/Keccak-submission-3.pdf.
10. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview (version 3.0), 2011. http://keccak.noekeon.org/Keccak-implementation-3.0.pdf.
11. Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.
12. Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family, version 1.3. Revised submission to NIST, 2008. http://www.skein-hash.info/sites/default/files/skein1.3.pdf.
13. Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, , Martin Schl affer, , and Søren S. Thomsen. Grøstl a SHA-3 candidate (version 2.0). Revised submission to NIST, 2011. http://www.groestl.info/Groestl.pdf.
14. Hongjun Wu. The hash function JH (updated version from jan. 16, 2011). Revised submission to NIST, 2011. http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf.