Structuring and optimizing KECCAK software (extended abstract)

Gilles Van Assche[1] and Ronny Van Keer[1]

STMicroelectronics

Abstract. Originally submitted as a hash function to the SHA-3 contest, KECCAK has shown quite some flexibility and was used to build several variants. We propose the KECCAK code package for reusing the common parts of these algorithms and for narrowing the scope of the optimizations. In this presentation, we discuss some aspects of the software implementation of KECCAK-based algorithms, including techniques to optimize the KECCAK-p family of permutations and the structure of the KECCAK code package.

Originally submitted as a hash function to the SHA-3 contest, KECCAK has shown quite some flexibility, e.g., as extendable output-function, pseudo-random bit generator or authenticated encryption scheme, and was used to build several variants such as KETJE, KEYAK or KANGAROOTWELVE [3,7,8,9,14]. All these algorithms share the use of the sponge construction or a construction close to it (i.e., duplex, monkey duplex, full-state keyed duplex) and of the KECCAK-p round function [1,2,4,12,13].

A software implementation can naturally benefit from reusing the common parts of these algorithms. However, one has to deal with diversity too. For instance, KECCAK itself relies on permutations of different sizes, from KECCAK-$f[200]$ to KECCAK-$f[1600]$ for what concerns software implementation. Some variants use a different number of rounds, e.g., KEYAK and KANGAROOTWELVE use KECCAK-$p[1600, n_r = 12]$ instead of KECCAK-$f[1600]$ = KECCAK-$p[1600, n_r = 24]$. Others rely on the parallel evaluation of these permutations and expect fast implementations using SIMD instructions. Combined with the different constructions and modes, and the different platforms one wishes to optimize for, the number of implementations can grow quickly.

As a solution, we propose the *KECCAK code package* [10]. It is structured in two levels. The high-level cryptographic services implement the modes and constructions in plain C, without any specific optimizations, while the low-level services implement the permutations and the state input/output functions, which can be optimized for a given platform. The idea is to have a single, portable, code base for the high level and the possibility to dedicate the low level to certain platforms for best performance.

In this presentation, we will detail some aspects of the software implementation of KECCAK-based algorithms, namely:

- techniques to optimize the KECCAK-p family of permutations, from memory savings on small devices to specifics of the AVX-512™ instruction set;
- the structure of the KECCAK code package, with KANGAROOTWELVE as a case study.

1 Implementation techniques

We summarize here some implementation techniques that are relevant in software [6].

1.1 Minimizing the time

How to cut a lane: bit interleaving The state of Keccak-$f[1600]$ can be expressed as 25 lanes of 64 bits each. In software, this calls for an implementation using 64-bit words. While this is an optimal choice on software platforms actually offering 64-bit operations, the bit interleaving technique allows efficient implementations on systems with smaller word sizes and can also be used to target compact hardware circuits.

In its simplest form, namely factor-2 interleaving, it splits each lane in two words: one containing the bits with even indices and one with odd indices. The state of Keccak-$f[1600]$ is then represented as 50 words of 32 bits. The rotations in θ and ρ are performed as cyclic shifts on 32-bit words, making them efficient on a 32-bit processor. There is a cost associated to the conversion of the input message into this representation, but this cost remains small compared to the evaluation of the permutation itself.

In general an interleaving factor of s maps each lane to s words of $\frac{64}{s}$ bits. For instance, factor-8 interleaving expresses the round function of Keccak-$f[1600]$ in terms of operations on bytes. Further details and examples can be found in [6, Section 2.1].

Processing planes A plane is a set of 5 lanes that can be combined in χ. So doing plane-per-plane processing nicely fits in χ. The dispersion step π just before χ can be implemented implicitly by fetching the lanes from appropriate locations, and the rotations ρ can be done individually on each lane together with π. The step θ can be done on the fly (see Section 1.1). Detailed scheduling of the operations can be found in [6, Section 2.4].

Bit interleaving can also be used to process fractions of planes. For an interleaving factor of s, 5 words of $64/s$ bits are processed together. Currently, the fastest software implementations are organized to process each plane at a time. This includes both implementations optimized for 64-bit platforms ($s = 1$, no interleaving) and those for 32-bit ones ($s = 2$).

Lane complementing The mapping χ of Keccak-$f[1600]$ consists in 5 XOR, 5 AND and 5 NOT operations. Some platforms support instructions that combine a AND and a NOT, but not all do. In the latter case, the lane complementing technique aims at removing 4 out of 5 NOT operations by representing some of the lanes by their complement. This makes simple use of the De Morgan laws, replacing a fraction of the logical ANDs by ORs. We explain how this can be done in [6, Section 2.2].

Extending the state to compute θ on the fly The θ operation consists in XORing a pattern in the entire state that depends only on the parity of the columns before θ. The pattern to XOR is called the θ-effect and is constant over each column. If the implementation can afford a bit of extra memory, one can use 5 lanes:

– to accumulate the parity of the columns as the output of χ in the previous round is being computed, and/or

– to store the θ-effect to be able to XOR it as the current round is being processed.

Further details and examples can be found in [6, Section 2.3] and in [6, Section 2.4.1].

1.2 Minimizing the memory usage

In terms of memory usage, the sponge and duplex constructions have no feedforward loop and can do in-place absorbing, without the need for additional memory dedicated specifically to that purpose. Hence, the memory footprint of KECCAK is determined solely by that of the underlying permutation. We here describe a technique to minimize the memory needed by KECCAK-$f[1600]$ without sacrificing speed.

Efficient in-place processing A typical speed-optimized software implementation reserves two memory areas, each with the size of the state (200 bytes). The computation of a round takes the state in one area and stores the result in the other, alternatively. In [6, Section 2.5], we propose a way to store processed data back into the same memory location it was taken from. Hence a single instance of the state must be reserved, plus some extra memory to store the parity and/or the θ-effect (see Section 1.1). As π moves lanes to different coordinates, this requires to define a mapping between the lane coordinates (x, y) and the memory location that depends on the round number. The mapping has a cycle of 4 rounds, so after the 12 or 24 nominal rounds the memory area returns to its original configuration.

This technique can be combined with bit interleaving. In that case, the mapping between the lane coordinates and memory location must be adapted. E.g., with factor-2 interleaving the mapping still has a cycle of 4 rounds. For instance, a fast implementation on the 32-bit processor ARM Cortex-M3 makes use of the in-place processing with 4 rounds unrolled and requires only 272 bytes on the stack [6, Section 3.2.1].

2 Structure of the KECCAK code package

The KECCAK code package gathers different free and open-source implementations of KECCAK and variants. It is organized as illustrated in Figure 1. At the top, the high-level cryptographic services are implemented in plain C, without any specific optimizations. At the bottom, the low-level services implement the permutations and the state input/output functions, which can be optimized for a given platform. The interface between the two layers is called *SnP*, abbreviated from "state and permutation".

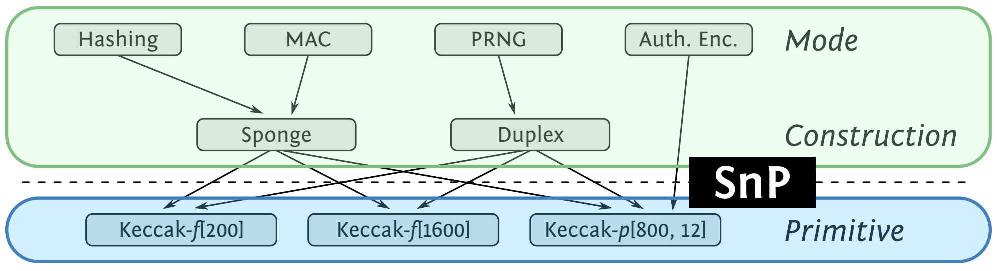

Fig. 1. The structure of the KECCAK code package.

The situation is similar for parallelized services, as illustrated in Figure 2. The interface is adapated to the parallelism and is called *PlSnP* ("parallel states and permutations").

This structure simplifies the set of optimized implementations on different platforms. Nearly all the processing takes place in the evaluation of the KECCAK-f permutation as well as in adding (using bitwise addition of vectors in GF(2)) input data into the state and extracting output data from it. The (Pl)SnP interfaces help isolate the parts that need to be most optimized, while the rest of the code can remain generic. Optimized implementations can be interchanged and a developer can select the best one for a given platform.

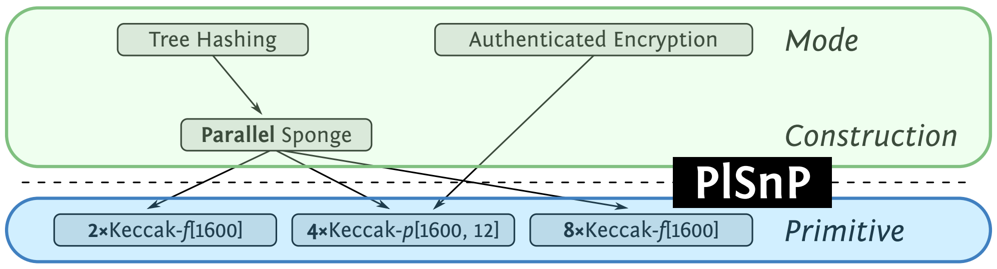

Fig. 2. The structure of the Keccak code package for modes exploiting parallelism.

2.1 SnP

The low-level services implement the different permutations Keccak-$f[200]$ to Keccak-$f[1600]$ and Keccak-$p[200, n_r]$ to Keccak-$p[1600, n_r]$. Note that these two permutation families are closely related. In Keccak-p the number of rounds is a parameter while in Keccak-f it is fixed. As Keccak-f are just instances of Keccak-p, we focus on the latter here.

Both the sponge and duplex constructions operate on a state. They apply a given permutation to it, add data into it or extract data from it. Therefore we define a layer that supports these three operations and their combination: the permutation and state management. We are aware that this slightly deviates from the layering depicted in Figure 1, as we support in fact operations that sponge and duplex need to perform on the state, including applying the permutation.

The low-level services provide an opaque representation of the state (i.e., of which the user does not have to know the details), together with functions to add data into and extract data from the state. This allows using an optimized implementation of Keccak-p that relies on a specific representation of the state, such as lane complementing or bit interleaving. Together with the permutations themselves, the low-level services implement the SnP interface, including functions:

- to initialize the state;
- to add (in GF(2)) or to overwrite bytes to the state;
- to extract bytes from the state, and optionally to add them to a buffer;
- to apply the permutation.

In stream and authenticated encryption, part of the output is used as a key stream, which is added to the plaintext or to the ciphertext. This addition is done by the low-level functions. The idea is that the higher level should be relieved from processing data, as this may need to be optimized in a platform-specific way.

2.2 PlSnP

On several platforms, it is advantageous to execute the Keccak-p permutation on several state values in parallel as it can result in faster processing per input/output data unit than when using sequential executions. This is typically useful for tree hashing modes, key stream generation and authenticated encryption.

We define an opaque structure that gathers n states, currently for $n \in \{2, 4, 8\}$. The PlSnP interface offers a function performing the Keccak-p permutation on the n state instances in parallel. On a platform that does not benefit from parallelism, this multi-instance function can call the single-permutation implementation n times as a fall-back. Or more generally, a n-times function can call a $\frac{n}{2^i}$-times function as a fall-back.

The reason for making the structure opaque is to allow an optimized implementation organizing the n states in a favorable way. For instance, an implementation using 128-bit SIMD instructions could store the 64-bit lane (x, y) of state #0 immediately followed by the 64-bit lane (x, y) of state #1 so as to be able to load the two lanes in one shot, as proposed in [6, Section 3.1.3].

The SnP interface includes functions:

- to initialize the n states;
- to add (in GF(2)) or to overwrite bytes to one of the states;
- to add or to overwrite bytes in all the states at once (with the granularity of a lane);
- to extract bytes from one of the states, and optionally to add them to a buffer;
- to extract bytes (and optionally add them) from all the states at once (with the granularity of a lane);
- to apply the permutation on all the states.

2.3 High-level services

In the Keccak code package, the currently implemented services are:

- the Keccak sponge functions;
- the Keccak duplex objects;
- the six approved FIPS 202 instances [13], i.e.,
  - the SHAKE128 and SHAKE256 extendable output functions and
  - the SHA-3 hash functions;
- a pseudo-random number generator based on Keccak duplex objects;
- NIST's fast parallel hash (FPH) proposal [14];
- the authenticated encryption schemes
  - River, Lake, Sea, Ocean and Lunar Keyak,
  - Ketje Jr and Ketje Sr;
- KangarooTwelve.

3 KangarooTwelve

KangarooTwelve is a recently proposed extendable output function (XOF), i.e., a generalized cryptographic hash function with arbitrary output length [9]. As illustrated on Figure 3, it combines the use of the Keccak-$p[1600, n_r = 12]$ permutation defined in FIPS 202 with the sponge construction, the parallelism of tree hashing, final node growing and Sakura coding [5]. It cuts the input string into chunks of $B = 8192$ bytes.

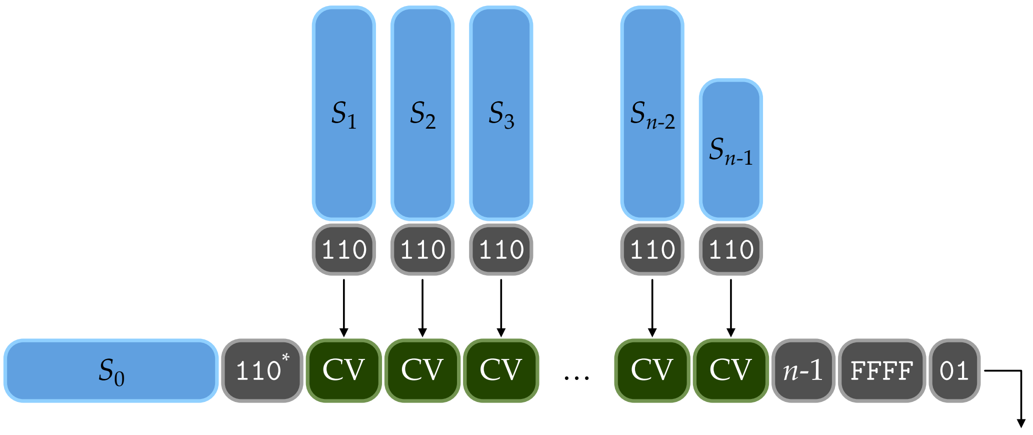

Fig. 3. Schematic of KANGAROOTWELVE, where the arrows denote calls to F defined as SPONGE[KECCAK-p[1600, $n_r = 12$], pad10*1, $r = 1344$].

3.1 Implementation

We implemented KANGAROOTWELVE in C and made it available in the KECCAK code package. The implementation has an interface that accepts the input message in pieces of arbitrary sizes. This is useful if a file, larger than the memory size, must be processed.

We have integrated the KANGAROOTWELVE code as illustrated on Figure 4. In particular, we instantiate the sponge construction on top of KECCAK-p[1600, $n_r = 12$] to implement the function F, at least to compute the final node. The function F on the leaves is computed as much in parallel as possible, i.e., if at least $8B$ input bytes are given by the caller, it uses a function that computes 8 times KECCAK-p[1600, $n_r = 12$] in parallel; if it is not available and if at least $4B$ bytes are given, it computes $4 \times$ KECCAK-p[1600, $n_r = 12$] in parallel; and so on. If no parallel implementation exists for the given platform, or if not enough bytes are given by the caller, it falls back on a serial implementation like for the final node.

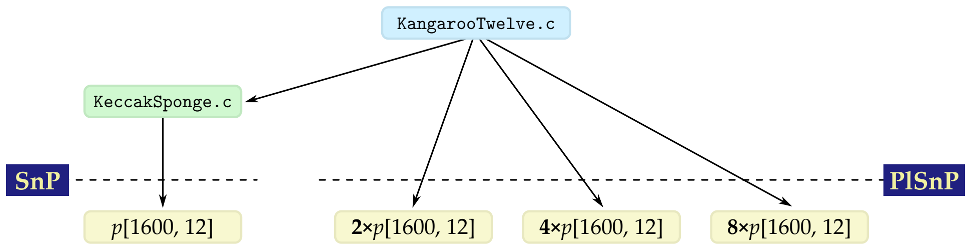

Fig. 4. The structure of the code implementing KANGAROOTWELVE in the KECCAK code package.

The KECCAK code package foresees that the serial and parallel implementations of the KECCAK-p permutation can be optimized for a given platform. Naturally, the code for the tree hash mode and the sponge construction is generic C, without optimizations for specific platforms, and it accesses the optimized permutation-level functions through SnP and PlSnP.

3.2 256-bit SIMD

Recent processors, in the Intel's® Haswell and Skylake families, support a 256-bit SIMD instruction set called AVX2™. We can exploit it to compute $4 \times$ Keccak-$p[1600, n_r = 12]$ efficiently.

On an Intel® Core™ i5-6500 (Skylake), we measured that $1 \times$ Keccak-$p[1600, n_r = 12]$ takes about 530 cycles, while $2\times$ about 730 cycles and $4 \times$ Keccak-$p[1600, n_r = 12]$ about 770 cycles. This does not include the time needed to add the input bytes to the state. Yet, this clearly points out that the time per byte decreases with the degree of parallelism.

Figure 5 displays the number of cycles for input messages up to $150,000$ bytes. Microscopically, the computation time steps up for every additional $R = 168$ bytes, but this is not visible on the figure. Macroscopically, when $|S| < B$, the time is a straight line with a slope of about 3.72 cycles/byte, i.e., the speed for F implemented serially. At $|S| = B = 8192$, there is a slight bump (a) as the tree gets a leaf, which causes an extra evaluation of Keccak-$p[1600, n_r = 12]$. When $|S| = 3B = 24,576$, two leaves can be computed in parallel and the number of cycles drops. When $|S| = 5B = 40,960$, four leaves can be computed in parallel and we see another drop. From then on, the same pattern repeats and one can easily identify the slopes of serial, $\times 2$ and $\times 4$ parallel implementations of Keccak-$p[1600, n_r = 12]$.

Note that a more advanced implementation could in principle remove the peaks of Figure 5 and make it monotonous. It could do so by using, e.g., the fast $4 \times$ Keccak-$p[1600, n_r = 12]$ implementation even if there are less than $4B$ bytes available, with dummy input bytes. However, at this point, we preferred code simplicity over speed optimization.

Figure 6 shows the implementation cost in cycles per bytes. To determine the speed in cycles per byte for long messages in our implementation, we need to take into account both the time to process $4B$ input bytes in 4 leaves (or a multiple thereof) and to process a whole block of chaining values in the final node. Regarding the latter, 21 chaining values fit in exactly 4 blocks of $R = 168$ bytes. Hence, we measure the time taken to process an extra $84B = \mathrm{lcm}(4B, 21B)$ bytes. These results are reported in Table 1, together with measurement on short messages.

In our implementation, the final node is always processed with a serial implementation. In principle, a more advanced implementation could buffer about B bytes of chaining values and process them in parallel to the leaves. Again, we preferred to keep our code simple.

4 512-bit SIMD

Intel® announced the development of processors with the AVX-512™ instruction set. This instruction set will support 512-bit SIMD instructions, enabling efficient implementations of $8 \times$ Keccak-$p[1600, n_r = 12]$. In addition to a higher degree of parallelism, we also expect that some new features of AVX-512™ will benefit to the implementation of KangarooTwelve, of FPH and of Keccak in general.

– *Rotation instructions.* With the exception of AMD's® XOP™, earlier SIMD instruction sets did not include a rotation instruction. This means that the cyclic shifts in θ and ρ had to be implemented with a sequence of three instructions (shift left, shift right,

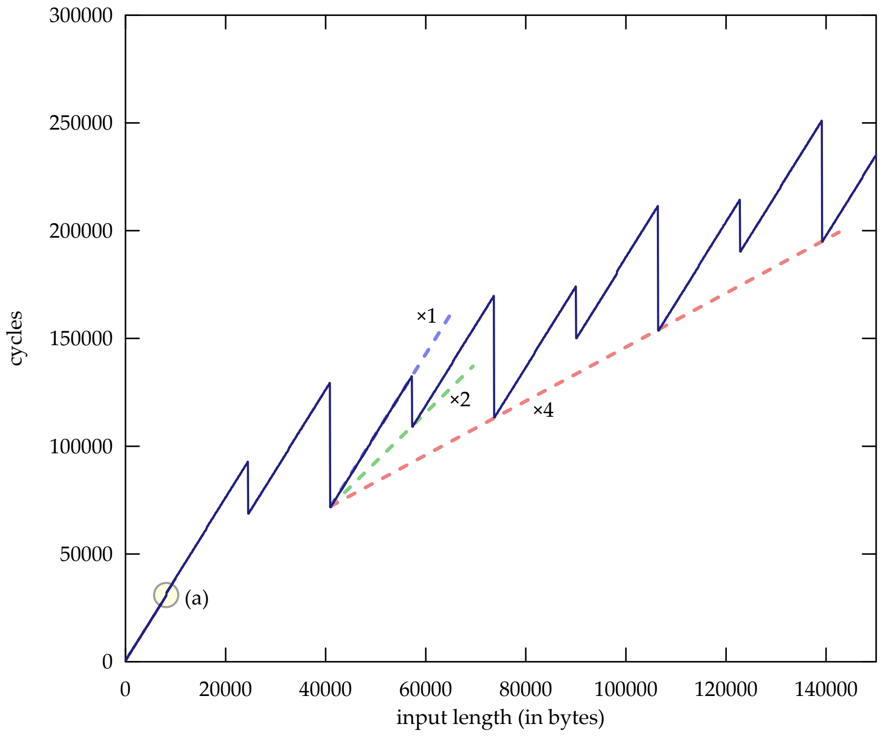

Fig. 5. The number of cycles of KangarooTwelve on an Intel® Core™ i5-6500 (Skylake) as a function of the input message size.

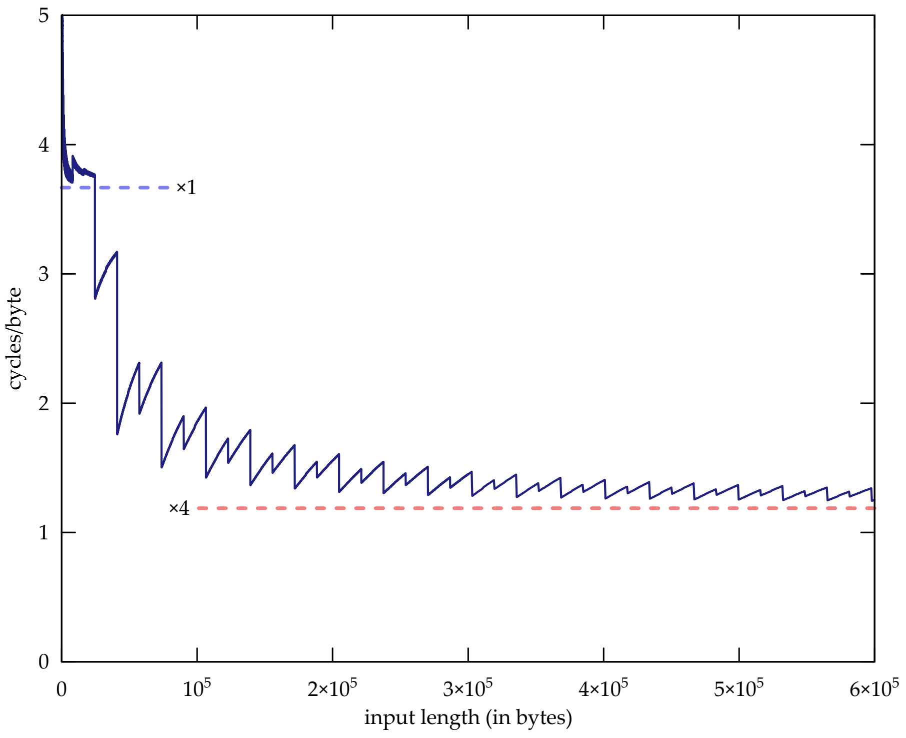

Fig. 6. The number of cycles per byte of KangarooTwelve on an Intel® Core™ i5-6500 (Skylake) as a function of the input message size.

XOR). With a rotation instruction, cyclic shifts are thus reduced from three to one instruction.

- *Three-input binary functions.* AVX-512™ offers an instruction that produces an arbitrary bitwise function of three binary inputs. In θ, computing the parity takes four XORs, which can be reduced to two applications of this new instruction. Similarly, the non-linear function χ can benefit from it to directly compute $a_x + (a_{x+1} + 1)a_{x+2}$.
- *32 registers.* Compared to AVX2™, the new processors will increase the number of registers from 16 to 32. As Keccak-p has 25 lanes, this will significantly decrease the need to move data between memory and registers.

At this time of writing, we do not have access to a machine that supports it, but we nevertheless developed an implementation based on a simulation. Romain Dolbeau reported that it works correctly on an Intel® Xeon Phi™ 7250 [11].

Processor	Short messages	Long messages
Intel® Core™ i5-4570 (Haswell)	4.15 c/b	1.44 c/b
Intel® Core™ i5-6500 (Skylake)	3.72 c/b	1.22 c/b
Intel® Xeon Phi™ 7250 (Knights Landing)	4.56 c/b	0.74 c/b

Table 1. The overall speed for short ($|S| = nR \leq B$) and for long ($|S| \gg B$) messages.

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, `http://sponge.noekeon.org/`, pp. 181–197.
2. ———, *Duplexing the sponge: single-pass authenticated encryption and other applications*, Selected Areas in Cryptography (SAC), 2011.
3. ———, *The* Keccak *reference*, January 2011, `http://keccak.noekeon.org/`.
4. ———, *Permutation-based encryption, authentication and authenticated encryption*, Directions in Authenticated Ciphers, July 2012.
5. ———, *Sakura: A flexible coding for tree hashing*, ACNS (I. Boureanu, P. Owesarski, and S. Vaudenay, eds.), Lecture Notes in Computer Science, vol. 8479, Springer, 2014, `http://dx.doi.org/10.1007/978-3-319-07536-5_14`, pp. 217–234.
6. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, Keccak *implementation overview*, May 2012, `http://keccak.noekeon.org/`.
7. ———, *CAESAR submission:* Ketje *v1*, March 2014, `http://ketje.noekeon.org/`.
8. ———, *CAESAR submission:* Keyak *v2*, September 2015, `http://keyak.noekeon.org/`.
9. ———, KangarooTwelve*: fast hashing based on* Keccak-*p*, Cryptology ePrint Archive, Report 2016/770, 2016, `http://eprint.iacr.org/2016/770`.
10. ———, Keccak *code package*, June 2016, `https://github.com/gvanas/KeccakCodePackage`.
11. R. Dolbeau, *private communications*, August 2016.
12. B. Mennink, R. Reyhanitabar, and D. Vizár, *Security of full-state keyed sponge and duplex: Applications to authenticated encryption*, Advances in Cryptology - ASIACRYPT 2015, New Zealand, 2015 (T. Iwata and J. H. Cheon, eds.), LNCS, vol. 9453, Springer, 2015, pp. 465–489.
13. NIST, *Federal information processing standard 202, SHA-3 standard: Permutation-based hash and extendable-output functions*, August 2015, `http://dx.doi.org/10.6028/NIST.FIPS.202`.
14. ———, *NIST special publication 800-185, SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash (draft)*, August 2016, `http://csrc.nist.gov/publications/drafts/800-185/sp800_185_draft.pdf`.