KangarooTwelve: fast hashing based on Keccak-p

Guido Bertoni[1], Joan Daemen[1,2], Michaël Peeters[1], Gilles Van Assche[1], and Ronny Van Keer[1]

[1] STMicroelectronics
[2] Radboud University

Abstract. We propose a fast and secure arbitrary output-length hash function aiming at a higher speed than the FIPS 202's SHA-3 and SHAKE functions, while retaining their flexibility and basis of security. Furthermore, it can exploit a high degree of parallelism, whether using multiple cores or the single-instruction multiple-data (SIMD) instruction set of modern processors. On Intel's® Haswell and Skylake architectures, KangarooTwelve tops at less than 1.5 cycles/byte for long messages on a single core. Short messages also benefit from about a factor two speed-up compared to the fastest FIPS 202 instance SHAKE128.

1 Introduction

Symmetric cryptography involves careful trade-offs between performance and security. The performance of a cryptographic primitive can be objectively measured, although it can yield a wide spectrum of figures depending on the variety of hardware and software platforms that the users may be interested in. Out of these, performance on widespread processors is easily measurable and naturally becomes the most visible feature. Security on the other hand cannot be measured. The best one can do is to obtain security assurance by relying on public scrutiny by skilled cryptanalysts. This is a scarce resource and the gaining of insight requires time and reflection. With the growing emphasis on provable security reduction of modes, the fact that the security of the underlying primitives is still based on this should not be overlooked.

In this note, we propose an *extendable output function* (XOF), i.e., a generalized cryptographic hash function with arbitrary output length, called KangarooTwelve. 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 [9,3].

An attractive feature of KangarooTwelve is that, for its security assurance, it directly inherits the security assurance built up for Keccak and the FIPS 202 functions. In particular, it uses the sponge construction with its simple and transparent indifferentiability proof [1]. The underlying permutations Keccak-p on the other hand have a furnished track record of public scrutiny since their publication in 2008. Its round function was never tweaked and hence all cryptanalysis during and after the SHA-3 competition remains relevant.

Our proposal gets its high speed, among other things, from using the Keccak-$f[1600]$ permutation reduced to 12 rounds. Clearly, 12 rounds provide less safety margin than the full 24 rounds in SHA-3 and SHAKE functions. Still, the safety margin provided by 12 rounds is comfortable as, e.g., the best published collision attacks at time of writing break Keccak only up to 5 rounds [6,7,11].

Another design choice that provides a significant speed-up is the use of a tree hash mode transparently for the user. A tree hash mode typically cuts the message into chunks, hashes these chunks independently, and hashes again the resulting chaining values to produce the final output. In April, the NIST posted on the hash forum two draft special publications, which recently became the SP 800-185 draft, including proposals a parallelized hash mode

(Fast Parallel Hash, or FPH) [10]. We implemented FPH in the KeccakCode Package and found it to be significantly faster than SHAKE128 and SHAKE256 for long messages when exploiting SIMD instruction sets such as Intel® AVX2™ [5].

Both FPH and KangarooTwelve offer a high degree of parallelism. The main advantage of the final-node growing approach is that implementers can decide on the degree of parallelism their programs support. A simple implementation could compute everything serially, while another would process two branches in parallel, or four, or more. Future processors can even contain more cores or wider SIMD instruction sets, such as Intel® AVX-512™, and KangarooTwelve will be readily able to exploit them. Another advantage is that it improves interoperability: Implementations do not need to have matching degrees of parallelism, although the function may need to be evaluated on different machines and maybe some years apart.

Compared to FPH, KangarooTwelve improves on the speed for short messages. A tree hash mode typically introduces some fixed overhead, which may be too costly for short messages. In our proposal, we instead make use of *kangaroo hopping*, which merges the hashing of the first chunk of the message and that of the chaining values [3]. As a result, there is no overhead until the input grows into more than one chunk.

After setting up some notation conventions in Section 2, we specify KangarooTwelve in Section 3. Section 4 gives a rationale and Section 5 introduces a closely related variant called MarsupilamiFourteen. In Section 6, we discuss implementation aspects and display benchmarks for recent processors. Finally, a reference implementation and test vectors are given in Appendices A and B, respectively.

2 Notation

A bit is an element of $\mathbb{Z}_2$. A string of bits is denoted using single quotes, e.g., '`0`' or '`111`'. The length in bits of a string s is denoted $|s|$. The concatenation of two strings a and b is denoted $a||b$. The truncation of a string s to its first n bits is denoted $\lfloor s \rfloor_n$. The empty string is denoted as $*$.

A byte is a string of 8 bits. The byte $(b_0, b_1, \ldots, b_7)$ can also be represented by the integer value $\sum_i 2^i b_i$ written in hexadecimal. E.g., the byte '`01100101`' can be equivalently written as `0xA6`.

The function $\text{enc}_8(x)$ encodes the integer x, with $0 \leq x \leq 255$, as a byte with value x.

3 Specifications of KangarooTwelve

KangarooTwelve is an extendable output function (XOF). It takes as input a pair of strings M and C, where

- M is the input message and
- C is a customization string.

Both M and C are assumed to be strings of bytes, i.e., $|M|$ and $|C|$ are multiples of 8.

KangarooTwelve produces unrelated outputs on two different pairs (M, C). Although it is not restricted in length or value, the customization string C is meant to provide *domain separation*. That is, for two different customization strings $C_1 \neq C_2$, KangarooTwelve gives

two independent functions of M. In practice, C is typically a short string, such as a name, an address or an identifier (e.g., URI, OID).

As a XOF, the output of KangarooTwelve is unlimited, and the user can request as many output bits as desired. It becomes a traditional hash function by restricting its output length to the desired digest size.

The core of KangarooTwelve is the Keccak-$p[1600, n_r = 12]$ permutation, i.e., a version of the one used in SHAKE and SHA-3 instances reduced to $n_r = 12$ rounds [9]. We build a sponge function F on top of this permutation with capacity set to $c = 256$ bits and therefore with rate $r = 1600 - c = 1344$, i.e.,

$$F = \text{sponge}[\text{Keccak-}p[1600, n_r = 12], \text{pad10}^*1, r = 1344].$$

On top of the sponge function F, KangarooTwelve uses a Sakura-compatible tree hash mode, which we now describe.

First, we merge M and C to a single input string S in a reversible way by concatenating:

- the input message M;
- the customization string C;
- the length in bytes of C encoded using `right_encode` $\left(\frac{|C|}{8}\right)$ as in Algorithm 1.

Algorithm 1 The function `right_encode`(x)

Input: a non-negative integer $x < 256^{255}$
Output: a string of bytes

Let l be the smallest number such that $x < 256^l$
Let $x = \sum_{i=0}^{l-1} x_i 256^i$ with $0 \leq x_i \leq 255$ for all i
return $\text{enc}_8(x_{l-1})||\ldots||\text{enc}_8(x_1)||\text{enc}_8(x_0)||\text{enc}_8(l)$

Then, the input string $S \neq *$ is cut into chunks of $B = 8192$ bytes, i.e.,

$$S = S_0||S_1||\ldots||S_{n-1},$$

with $n = \left\lceil \frac{|S|}{8B} \right\rceil$ and where all chunks except the last one must have exactly B bytes.

When $n > 1$, KangarooTwelve builds a tree with the following final node `Node`$_*$ and inner nodes `Node`$_i$ with $0 \leq i \leq n-2$:

$$\begin{aligned}
\texttt{Node}_i &= S_{i+1}||\text{`}\texttt{110}\text{'} \\
\text{CV}_i &= \lfloor F(\texttt{Node}_i) \rfloor_{256} \\
\texttt{Node}_* &= S_0||\text{`}\texttt{110}^{62}\text{'}||\text{CV}_0||\ldots||\text{CV}_{n-2}||\texttt{right_encode}(n-1) \\
&\quad ||\texttt{0xFF}||\texttt{0xFF}||\text{`}\texttt{01}\text{'} \\
\text{KangarooTwelve}(M, C) &= F(\texttt{Node}_*).
\end{aligned}$$

The chaining values CV_i have length $c = 256$ bits. This is illustrated in Figure 1.

When $n = 1$, the tree reduces to its single final node `Node`$_*$ and KangarooTwelve becomes:

$$\begin{aligned}
\texttt{Node}_* &= S||\text{`}\texttt{11}\text{'} \\
\text{KangarooTwelve}(M, C) &= F(\texttt{Node}_*).
\end{aligned}$$

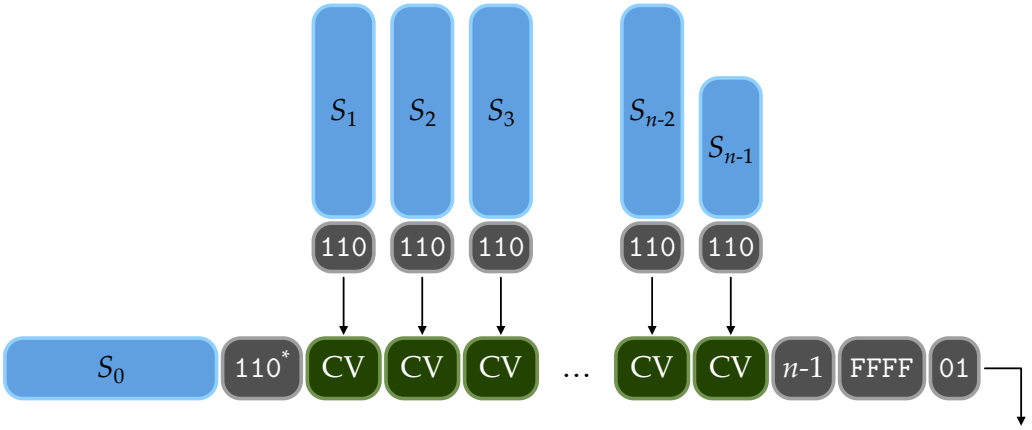

Fig. 1. Schematic of KangarooTwelve for $|S| > B$, with arrows denoting calls to F.

We make a flat sponge claim with 255 bits of claimed capacity as in Claim 1. In short, this means that KangarooTwelve shall resist against all attacks with complexity up to 2^{128}, unless easier on a random oracle [2].

Claim 1 (Flat sponge claim [2]) *The success probability of any attack on KangarooTwelve shall not be higher than the sum of that for a random oracle and*

$$1 - \mathrm{e}^{-\frac{N^2}{2^{256}}},$$

with N the attack complexity in calls to Keccak-p[1600, $n_\mathrm{r} = 12$] or its inverse. We exclude from the claim weaknesses due to the mere fact that the function can be described compactly and can be efficiently executed, e.g., the so-called random oracle implementation impossibility [8], as well as properties that cannot be modeled as a single-stage game [12].

Note that $1 - \mathrm{e}^{-\frac{N^2}{2^{256}}} < \frac{N^2}{2^{256}}$.

4 Rationale

In this section, we provide some more in-depth explanations on the design choices in KangarooTwelve.

4.1 Security of the tree hash mode

The tree hash mode is Sakura-compatible so that it automatically satisfies the conditions of soundness and guarantees security against generic attacks [4,3]. Claim 1 takes into account inner collisions both in F and in the chaining values of the tree hash mode, hence a claimed capacity of $256 - 1 = 255$ bits.

We use the terminology of Sakura [3]. A *hop* is either a chunk of the message or a sequence of chaining values. Multiple hops can be combined into *nodes*, namely, the strings that will be subject to the underlying hash function. The encoding of the nodes is as follows.

– When $n = 1$, there is only the final node. Following Sakura, the node contains the input string S followed by '1' to make a message hop, then followed by '1' to make a final node.

– When $n > 1$, there are inner nodes and the final node.
  - Each inner node contains a chunk S_i of the input string followed by '1' to make a message hop, then followed by '1' for simple padding and '0' to make an inner node.
  - The final node starts with the first chunk of the input string S_0 followed by '1' to make a message hop, then followed by padding of the form '1'||'0*' as part of the kangaroo hopping and to align what follows to a multiple of 64 bits. The chaining hop then contains the chaining values, followed by the coded number of chaining values (right_encode$(n-1)$), no interleaving ($I = \infty$, coded with two bytes 0xFF) and the bit '0'. This is followed by '1' to indicate it is the final node.

4.2 Choice of B

Although it could be defined as a user-chosen parameter, we decided to fix the size of the message chunks to $B = 8192$ bytes. The exact value of B has some impact on the performance, although we think that its importance is relatively small. Instead, we decided to relieve the user from the burden of this technical choice and to facilitate interoperability.

We chose $B = 8192$ for the following reasons. First, we think that a power of two can help bulk data input in time-critical applications. For instance, when hashing a large file, we expect the implementation to be faster and easier if the chunks contain a whole number of disk sectors.

The chaining values in the final node create an overhead, as the total length of the nodes that F has to process grows relatively by $\frac{c}{B}$. In our case, it amounts to about 0.4%.

Another concern is the number of *unused bytes* in the last r-bit block of the input to F. We have $r = 1344$ bits or $R = r/8 = 168$ bytes. When cutting the chunk S_i into blocks of R bytes, it leaves $W = -(B+1) \bmod R$ unused bytes in the last block. It turns out that W reaches a minimum for $B = 2^{7+6n}$ with $n \geq 0$ an integer. Its relative impact, $\frac{W}{B}$, decreases as B increases. For small values, e.g., $B \in \{128, 256, 512\}$, this is about 30%, while for $B = 8192$ it drops below 0.5%.

Increasing B would further reduce these two overhead factors. However, they are relatively small already and this would delay the benefits of parallelism to longer messages.

Finally, the choice of B bounds the degree of parallelism that an implementation can fully exploit. An implementation can in principle compute the final node and leaves in parallel, but if more than B/c leaves are processed at once, the final node grows faster than B bytes at a time. The chosen value of B allows a parallelism up to degree $B/c = 256$.

5 If 256-bit security is desired, welcome MarsupilamiFourteen

Defined in Section 3, KangarooTwelve provides 128-bit security, while some users may wish to use a hash function in a consistent combination with cryptographic functions of 256-bit security strength. Often, this higher security strength is requested as a way to achieve a thicker safety margin. We feel that, as such, 256-bit security does not provide a practical and tangible security improvement over 128-bit security. A cipher that stands by its claim of 128-bit security provides enough protection against any adversary in the foreseeable future.

To address both concerns, we propose a 256-bit security instance that also has some increased safety margin, hence an increased number of rounds. Our proposal is MarsupilamiFourteen, with the same specifications as KangarooTwelve, except that (i) the capacity and chaining values have $c = 512$ bits, (ii) the number of rounds is raised to $n_r = 14$ and (iii) we make a flat sponge claim with 511 bits of claimed capacity, i.e., MarsupilamiFourteen shall resist against all attacks with complexity up to 2^{256}, unless easier on a random oracle.

Of course, for even thicker safety margins, one can also use the standard FIPS 202 instances or FPH [9,10].

6 Implementation

We implemented KangarooTwelve in C and made it available in the Keccak code package (KCP) [5].

6.1 Structure

The implementation has an interface that accepts the input message M in pieces of arbitrary sizes. This is useful if a file, larger than the memory size, must be processed. The customization string C can be given at the end.

We have integrated the KangarooTwelve code in KCP as illustrated on Figure 2. 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.

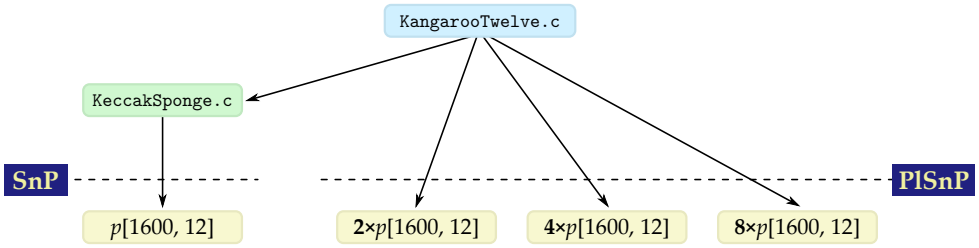

Fig. 2. The structure of the code implementing KangarooTwelve in the KCP.

The KCP foresees that the serial and parallel implementations of the Keccak-p permutation can be optimized for a given platform. In contrast, 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 an interface called *SnP* (for a single permutation) or *PlSnP* (for permutations computed in parallel) [5].

To input large messages M, the state to maintain between two calls internally uses two queues: one for the final node and one for the current leaf. To save memory, each of these

queues directly uses the state of F, to which the input bytes are added. Of course, if a message is known to be smaller than or equal to B bytes, one could further save one queue.

6.2 256-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 3 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 3 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 some dummy input bytes. However, at this point, we preferred code simplicity over speed optimization.

Figure 4 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 $84B = \text{lcm}(4B, 21B)$ bytes. These results are reported in Table 1, together with measurement on short messages.

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

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

6.3 512-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

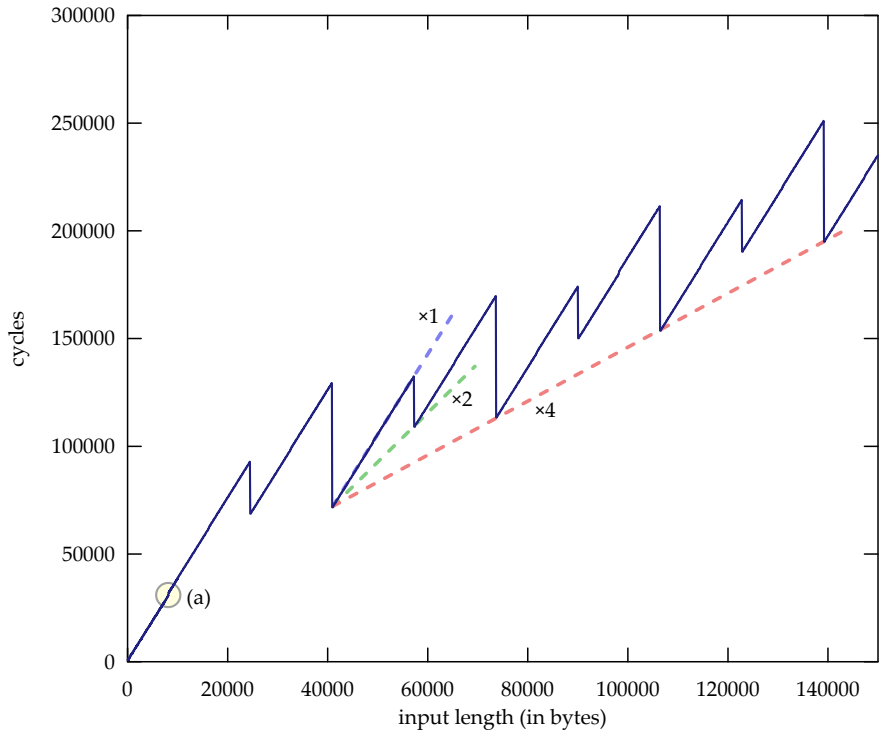

Fig. 3. The number of cycles of KangarooTwelve on an Intel® Core™ i5-6500 (Skylake) as a function of the input message size.

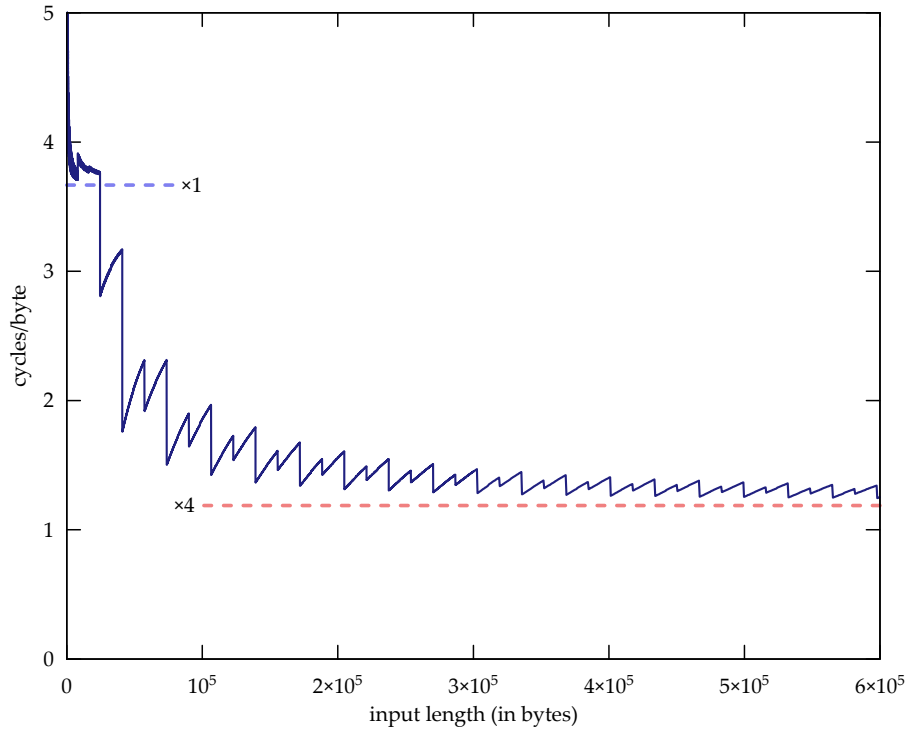

Fig. 4. The number of cycles per byte of KangarooTwelve on an Intel® Core™ i5-6500 (Skylake) as a function of the input message size.

of $8 \times$ KeccakP$[1600, n_r = 12]$. 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, 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 KeccakP 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 experimental implementation based on a simulation [5]. Romain Dolbeau reported that it works correctly on the actual hardware, although we could not measure it or optimize it yet.

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. ———, *Cryptographic sponge functions*, January 2011, http://sponge.noekeon.org/.
3. ———, *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.
4. ———, *Sufficient conditions for sound tree and sequential hashing modes*, International Journal of Information Security **13** (2014), 335–353, http://dx.doi.org/10.1007/s10207-013-0220-y.
5. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, Keccak *code package*, June 2016, https://github.com/gvanas/KeccakCodePackage.
6. I. Dinur, O. Dunkelman, and A. Shamir, *Collision attacks on up to 5 rounds of SHA-3 using generalized internal differentials*, Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers (S. Moriai, ed.), Lecture Notes in Computer Science, vol. 8424, Springer, 2013, pp. 219–240.
7. ———, *Improved practical attacks on round-reduced Keccak*, J. Cryptology **27** (2014), no. 2, 183–209.
8. U. Maurer, R. Renner, and C. Holenstein, *Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology*, Theory of Cryptography - TCC 2004 (M. Naor, ed.), Lecture Notes in Computer Science, no. 2951, Springer-Verlag, 2004, pp. 21–39.
9. 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.
10. ———, *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.
11. K. Qiao, L. Song, M. Liu, and J. Guo, *Solution to the 5-round collision challenge*, 2016, http://keccak.noekeon.org/crunchy_contest.html.
12. T. Ristenpart, H. Shacham, and T. Shrimpton, *Careful with composition: Limitations of the indifferentiability framework*, Eurocrypt 2011 (K. G. Paterson, ed.), Lecture Notes in Computer Science, vol. 6632, Springer, 2011, pp. 487–506.

A Reference source code

In this section, we give (unoptimized) reference code written in Python, which can also be downloaded from the KCP [5]. The pieces of code are organized in a bottom-up layering fashion. Listing 1.1 implements the Keccak-$p[1600, n_r]$ permutations, which is then used in Listing 1.2 to build the sponge function F. The `right_encode` function and KangarooTwelve are displayed in Listing 1.3.

Listing 1.1. The Keccak-$p[1600, n_r]$ permutations

```
def ROL64(a, n):
    return ((a >> (64-(n%64))) + (a << (n%64))) % (1 << 64)

def KeccakP1600onLanes(lanes, nrRounds):
    R = 1
    for round in range(24):
        if (round + nrRounds >= 24):
            # θ
            C = [lanes[x][0] ^ lanes[x][1] ^ lanes[x][2] ^ lanes[x][3] ^ lanes[x][4] for x in range(5)]
            D = [C[(x+4)%5] ^ ROL64(C[(x+1)%5], 1) for x in range(5)]
            lanes = [[lanes[x][y]^D[x] for y in range(5)] for x in range(5)]
            # ρ and π
            (x, y) = (1, 0)
            current = lanes[x][y]
            for t in range(24):
                (x, y) = (y, (2*x+3*y)%5)
                (current, lanes[x][y]) = (lanes[x][y], ROL64(current, (t+1)*(t+2)//2))
            # χ
            for y in range(5):
                T = [lanes[x][y] for x in range(5)]
                for x in range(5):
                    lanes[x][y] = T[x] ^((~T[(x+1)%5]) & T[(x+2)%5])
            # ι
            for j in range(7):
                R = ((R << 1) ^ ((R >> 7)*0x71)) % 256
                if (R & 2):
                    lanes[0][0] = lanes[0][0] ^ (1 << ((1<<j)-1))
        else:
            for j in range(7):
                R = ((R << 1) ^ ((R >> 7)*0x71)) % 256
    return lanes

def load64(b):
    return sum((b[i] << (8*i)) for i in range(8))

def store64(a):
    return bytes((a >> (8*i)) % 256 for i in range(8))

def KeccakP1600(state, nrRounds):
    lanes = [[load64(state[8*(x+5*y):8*(x+5*y)+8]) for y in range(5)] for x in range(5)]
    lanes = KeccakP1600onLanes(lanes, nrRounds)
    state = b''.join([store64(lanes[x][y]) for y in range(5) for x in range(5)])
    return bytearray(state)
```

Listing 1.2. The function $F = \text{SPONGE}[\text{KECCAK-}p[1600, n_r = 12], \text{pad10}^*1, r = 1344]$

```
def F(inputBytes, delimitedSuffix, outputByteLen):
    outputBytes = b''
    state = bytearray([0 for i in range(200)])
    rateInBytes = 1344//8
    blockSize = 0
    inputOffset = 0
    # === Absorb all the input blocks ===
    while(inputOffset < len(inputBytes)):
        blockSize = min(len(inputBytes)-inputOffset, rateInBytes)
        for i in range(blockSize):
            state[i] = state[i] ^ inputBytes[i+inputOffset]
        inputOffset = inputOffset + blockSize
        if (blockSize == rateInBytes):
            state = KeccakP1600(state, 12)
            blockSize = 0
    # === Do the padding and switch to the squeezing phase ===
    state[blockSize] = state[blockSize] ^ delimitedSuffix
    if (((delimitedSuffix & 0x80) != 0) and (blockSize == (rateInBytes-1))):
        state = KeccakP1600(state, 12)
    state[rateInBytes-1] = state[rateInBytes-1] ^ 0x80
    state = KeccakP1600(state, 12)
    # === Squeeze out all the output blocks ===
    while(outputByteLen > 0):
        blockSize = min(outputByteLen, rateInBytes)
        outputBytes = outputBytes + state[0:blockSize]
        outputByteLen = outputByteLen - blockSize
        if (outputByteLen > 0):
            state = KeccakP1600(state, 12)
    return outputBytes
```

Listing 1.3. The `right_encode` function and KANGAROOTWELVE

```
def right_encode(x):
    S = b''
    while(x > 0):
        S = bytes([x % 256]) + S
        x = x//256
    S = S + bytes([len(S)])
    return S

def KangarooTwelve(inputMessage, customizationString, outputByteLen):
    B = 8192
    c = 256
    S = inputMessage + customizationString + right_encode(len(customizationString))
    # === Cut the input string into chunks of B bytes ===
    n = (len(S)+B-1)//B
    Si = [bytes(S[i*B:(i+1)*B]) for i in range(n)]
    if (n == 1):
        # === Process the tree with only a final node ===
        return F(Si[0], 0x07, outputByteLen)
    else:
        # === Process the tree with kangaroo hopping ===
        CVi = [F(Si[i+1], 0x0B, c//8) for i in range(n-1)]
        NodeStar = Si[0] + b'\x03\x00\x00\x00\x00\x00\x00\x00' + b''.join(CVi) \
            + right_encode(n-1) + b'\xFF\xFF'
        return F(NodeStar, 0x06, outputByteLen)
```

B Test vectors

In this section, we give some test vectors for KangarooTwelve, organized in three parts:

1. The input is empty, $M = C = *$, and the output size varies.
2. The customization string is empty, $C = *$, and the input message M is an arbitrary string of length 17^i for $0 \leq i \leq 6$. The value M is constructed by repeating the pattern `0x00, 0x01, 0x02, ..., 0xFA` as many as necessary and by truncating it to the specified length.
3. The input message M is obtained by repeating $2^i - 1$ times `0xFF`, while the customization string C is constructed following the same pattern as for M above, but with length 41^i for $0 \leq i \leq 3$.

```
KangarooTwelve(M=empty, C=empty, 32 output bytes):
1a c2 d4 50 fc 3b 42 05 d1 9d a7 bf ca 1b 37 51 3c 08 03 57 7a c7 16 7f 06 fe 2c e1 f0 ef 39 e5

KangarooTwelve(M=empty, C=empty, 64 output bytes):
1a c2 d4 50 fc 3b 42 05 d1 9d a7 bf ca 1b 37 51 3c 08 03 57 7a c7 16 7f 06 fe 2c e1 f0 ef 39 e5
42 69 c0 56 b8 c8 2e 48 27 60 38 b6 d2 92 96 6c c0 7a 3d 46 45 27 2e 31 ff 38 50 81 39 eb 0a 71

KangarooTwelve(M=empty, C=empty, 10032 output bytes), last 32 bytes:
e8 dc 56 36 42 f7 22 8c 84 68 4c 89 84 05 d3 a8 34 79 91 58 c0 79 b1 28 80 27 7a 1d 28 e2 ff 6d

KangarooTwelve(M=pattern 0x00 to 0xFA for 17^0 bytes, C=empty, 32 output bytes):
2b da 92 45 0e 8b 14 7f 8a 7c b6 29 e7 84 a0 58 ef ca 7c f7 d8 21 8e 02 d3 45 df aa 65 24 4a 1f

KangarooTwelve(M=pattern 0x00 to 0xFA for 17^1 bytes, C=empty, 32 output bytes):
6b f7 5f a2 23 91 98 db 47 72 e3 64 78 f8 e1 9b 0f 37 12 05 f6 a9 a9 3a 27 3f 51 df 37 12 28 88

KangarooTwelve(M=pattern 0x00 to 0xFA for 17^2 bytes, C=empty, 32 output bytes):
0c 31 5e bc de db f6 14 26 de 7d cf 8f b7 25 d1 e7 46 75 d7 f5 32 7a 50 67 f3 67 b1 08 ec b6 7c

KangarooTwelve(M=pattern 0x00 to 0xFA for 17^3 bytes, C=empty, 32 output bytes):
cb 55 2e 2e c7 7d 99 10 70 1d 57 8b 45 7d df 77 2c 12 e3 22 e4 ee 7f e4 17 f9 2c 75 8f 0d 59 d0

KangarooTwelve(M=pattern 0x00 to 0xFA for 17^4 bytes, C=empty, 32 output bytes):
87 01 04 5e 22 20 53 45 ff 4d da 05 55 5c bb 5c 3a f1 a7 71 c2 b8 9b ae f3 7d b4 3d 99 98 b9 fe

KangarooTwelve(M=pattern 0x00 to 0xFA for 17^5 bytes, C=empty, 32 output bytes):
84 4d 61 09 33 b1 b9 96 3c bd eb 5a e3 b6 b0 5c c7 cb d6 7c ee df 88 3e b6 78 a0 a8 e0 37 16 82

KangarooTwelve(M=pattern 0x00 to 0xFA for 17^6 bytes, C=empty, 32 output bytes):
3c 39 07 82 a8 a4 e8 9f a6 36 7f 72 fe aa f1 32 55 c8 d9 58 78 48 1d 3c d8 ce 85 f5 8e 88 0a f8

KangarooTwelve(M=0 times byte 0xFF, C=pattern 0x00 to 0xFA for 41^0 bytes, 32 output bytes):
fa b6 58 db 63 e9 4a 24 61 88 bf 7a f6 9a 13 30 45 f4 6e e9 84 c5 6e 3c 33 28 ca af 1a a1 a5 83

KangarooTwelve(M=1 times byte 0xFF, C=pattern 0x00 to 0xFA for 41^1 bytes, 32 output bytes):
d8 48 c5 06 8c ed 73 6f 44 62 15 9b 98 67 fd 4c 20 b8 08 ac c3 d5 bc 48 e0 b0 6b a0 a3 76 2e c4

KangarooTwelve(M=3 times byte 0xFF, C=pattern 0x00 to 0xFA for 41^2 bytes, 32 output bytes):
c3 89 e5 00 9a e5 71 20 85 4c 2e 8c 64 67 0a c0 13 58 cf 4c 1b af 89 44 7a 72 42 34 dc 7c ed 74

KangarooTwelve(M=7 times byte 0xFF, C=pattern 0x00 to 0xFA for 41^3 bytes, 32 output bytes):
75 d2 f8 6a 2e 64 45 66 72 6b 4f bc fc 56 57 b9 db cf 07 0c 7b 0d ca 06 45 0a b2 91 d7 44 3b cf
```