KangarooTwelve: fast hashing based on Keccak-p

Guido Bertoni[3], Joan Daemen[1,2], Michal Peeters[1], Gilles Van Assche[1], Ronny Van Keer[1], and Benoît Viguier[2]

[1] STMicroelectronics
[2] Radboud University
[3] Security Pattern

Abstract. We present KangarooTwelve, 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 sharing many features with SHAKE128, like the cryptographic primitive, the sponge construction, the eXtendable Output Function (XOF) and the 128-bit security strength, KangarooTwelve offers two major improvements over its standard counterpart. First it has a built-in parallel mode that efficiently exploits multi-core or SIMD instruction parallelism for long messages, without impacting the performance for short messages. Second, relying on the cryptanalysis results on Keccak over the past ten years, we tuned its permutation to require twice less computation effort while still offering a comfortable safety margin. By combining these two changes KangarooTwelve consumes less than 0.55 cycles/byte for long messages on the latest Intel®'s SkylakeX architectures. The generic security of KangarooTwelve is guaranteed by the use of Sakura encoding for the tree hashing and of the sponge construction for the compression function.

Keywords: symmetric cryptography, hash function, tree hashing, Keccak, software performance

1 Introduction

Most cryptography involves careful trade-offs between performance and security. The performance of a cryptographic function 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 public scrutiny should not be overlooked.

In this paper we present the hash function KangarooTwelve, or more exactly an *eXtendable Output Function* (XOF). KangarooTwelve makes use of a tree hash mode with Sakura encoding [29,9] and the sponge construction [7], both proven secure. Its underlying permutation is a member of the Keccak-$p[1600, n_r]$ family, differing from that of Keccak only in the number of rounds. Since its publication in 2008, the round function of Keccak was never tweaked [6]. Moreover, as for most symmetric cryptographic primitives, third-party cryptanalysis has been applied to reduced-round versions of Keccak. Hence KangarooTwelve's security assurance directly benefits from nearly ten years of public scrutiny, including all cryptanalysis during and after the SHA-3 competition [12].

KangarooTwelve gets its low computational workload per bit 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 6 rounds [14,15,35,36].

The other design choice that gives KangarooTwelve great speed for long messages is the use of a tree hash mode. This mode is transparent for the user in the sense that the message length fully determines the tree topology. Basically, the mode calls an underlying sponge-based compression function for each 8192-byte chunk of message and finally hashes the concatenation of the resulting digests. We call this the *final node growing* approach. Clearly, the chunks can be hashed in parallel.

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, four or more branches in parallel using multiple cores, or more simply, a SIMD instruction set such as the Intel® AVX2. Future processors can even contain an increasing number of cores, or wider SIMD registers as examplified by the recent AVX-512 instruction set, and KangarooTwelve will be readily able to exploit them. The fixed length of the chunks and the fact that the tree topology is fully determined by the message length improves interoperability: The hash result is independent of the amount of parallelism exploited in the implementation.

KangarooTwelve is not the only Keccak-based parallel hash mode. In late 2016, NIST published the SP 800-185 standard, including a parallelized hash mode called ParallelHash [30]. Compared to ParallelHash, KangarooTwelve improves on the speed for short messages. ParallelHash compresses message chunks to digests in a first stage and compresses the concatenation of the digests in a second stage. This two-stage hashing introduces an overhead that is costly for short messages. In KangarooTwelve we apply a technique called *kangaroo hopping*: It merges the hashing of the first chunk of the message and that of the

chaining values of the remaining chunks [9]. As a result, the two stages reduce to one if the input fits in one chunk with no overhead whatsoever.

Finally, KangarooTwelve is a concrete application of the Sakura encoding, which yields secure tree hash modes by construction [9].

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, we provide a reference implementation and test vectors 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 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 n times repetition of a bit '`s`' is denoted '$\mathtt{s}^n$', e.g. '$\mathtt{110}^4$' = '`110000`'. 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 bit string '`11110010`' can be equivalently written as `0x4F` as depicted in Figure 1. The function $\mathrm{enc}_8(x)$ encodes the integer x, with $0 \leq x \leq 255$, as a byte with value x.

The length in of a byte string s is denoted $\|s\|$. $(\mathtt{0x00})^n$ denotes the n times repetition of the byte `0x00`.

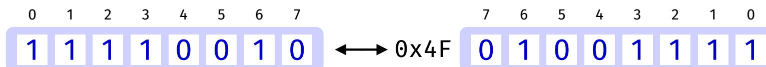

Fig. 1. Example of byte representation

3 Specifications of KangarooTwelve

KangarooTwelve is an eXtendable Output Function (XOF). It takes as input a message M and an optional customization string C, both byte strings of variable length.

KangarooTwelve produces unrelated outputs on different couples (M, C). The customization string C is meant to provide *domain separation*, namely, 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). KangarooTwelve naturally maps to a XOF with a single input string M by setting the customization string input C to the empty string. This allows implementing it with a classical hash function API.

As a XOF, the output of KangarooTwelve is unlimited, and the user can request as many output bits as desired. It can be used for traditional hashing simply by generating outputs of the desired digest size.

3.1 The inner compression function F

The core of KangarooTwelve is the Keccak-$p[1600, n_\mathrm{r} = 12]$ permutation, i.e., a version of the permutation used in SHAKE and SHA-3 instances reduced to $n_\mathrm{r} = 12$ rounds [29]. We build a sponge function F on top of this permutation with capacity set to $c = 256$ bits and therefore with rate $r = 1600 - c = 1344$. It makes use of multi-rate padding, indicated by pad10*1. Following [29], this is expressed formally as:

$$F = \textsc{sponge}[\textsc{Keccak}\text{-}p[1600, n_\mathrm{r} = 12], \mathrm{pad}10^*1, r = 1344].$$

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

3.2 The merged input string S

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 `length_encode`($\|C\|$) as in Algorithm 1.

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

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

with $n = \left\lceil \frac{\|S\|}{B} \right\rceil$ and where all chunks except the last one must have exactly B bytes. Note that there is always one block as S consists of at least one byte.

Algorithm 1 The function `length_encode`(x)

Input: an integer x in the range $0 \leq x \leq 256^{255} - 1$
Output: a byte string

Let l be the smallest integer in the range $0 \leq l \leq 255$ 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 $\text{enc}_8(x_{l-1})||\dots||\text{enc}_8(x_1)||\text{enc}_8(x_0)||\text{enc}_8(l)$

Examples:
`length_encode`(0) returns `0x00`
`length_encode`(12) returns `0x0C`||`0x01`
`length_encode`(65538) returns `0x01`||`0x00`||`0x02`||`0x03`

3.3 The tree hash mode

When $\|S\| > B$, we have $n > 1$ and KangarooTwelve builds a tree with the following final node $\texttt{Node}_*$ and inner nodes $\texttt{Node}_i$ with $1 \leq i \leq n-1$:

$$\begin{aligned}
\texttt{Node}_i &= S_i||\text{`}\texttt{110}\text{'} \\
\text{CV}_i &= \lfloor F(\texttt{Node}_i) \rfloor_{256} \\
\texttt{Node}_* &= S_0||\text{`}\texttt{110}^{62}\text{'}||\text{CV}_1||\dots||\text{CV}_{n-1}||\texttt{length_encode}(n-1) \\
&\quad ||\texttt{0xFF}||\texttt{0xFF}||\text{`}\texttt{01}\text{'}
\end{aligned}$$

$\text{KangarooTwelve}(M, C) = F(\texttt{Node}_*)$.

The chaining values CV_i have length $c = 256$ bits. This is illustrated in Figure 2.

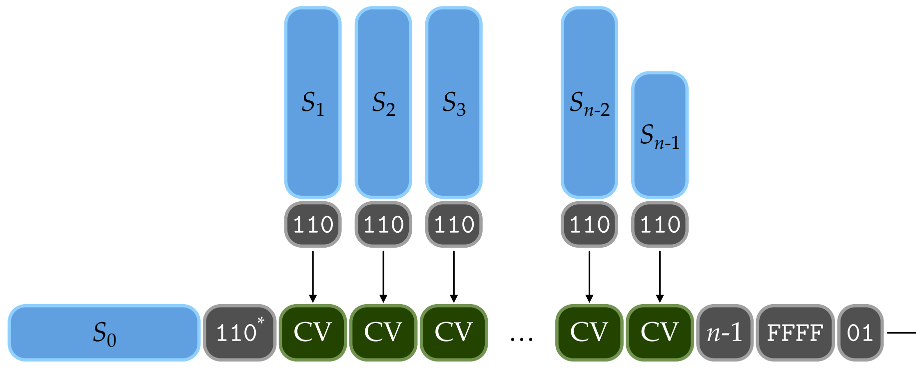

Fig. 2. Schematic of KangarooTwelve for $\|S\| > B$, with arrows denoting calls to F.

When $\|S\| \leq B$, we have $n = 1$ and the tree reduces to its single final node $\texttt{Node}_*$ and KangarooTwelve becomes:

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

3.4 Security claim

We make a flat sponge claim [8] with 255 bits of claimed capacity in Claim 1. Informally, it means that KangarooTwelve shall offer the same security strength as a random oracle whenever that offers a strength below 128 bits and a strength of 128 bits in all other cases. We discuss the implications of the claim more in depth in Section 4.1.

Claim 1 (Flat sponge claim [8]) *The success probability of any attack on* KangarooTwelve *shall not be higher than the sum of that for a random oracle and*

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

with N the attack complexity in calls to Keccak-$p[1600, n_r = 12]$ *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 [23], as well as properties that cannot be modeled as a single-stage game [32].*

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

4 Rationale

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

4.1 Implications of the security claim

The flat sponge claim covers all attacks up to a given *security strength* of 128 bits. Informally, saying that a cryptographic function has a security strength of s bits means that no attacks exist with complexity N and success probability p such that $N/p < 2^s$ [25].

The claim covers quasi all practically revelant security of KangarooTwelve including that of traditional hashing: collision, preimage and second preimage resistance. To achieve 128-bit security strength, the output n must be chosen long enough so that there are no generic attacks (i.e., also applicable to a random

oracle) that violate 128-bit security. So for 128-bit (second) preimage security the output should be at least 128 bits, and for 128-bit collision security the output should be at least 256 bits.

For many primitives the security strength that can be claimed degrades under multi-target attacks by $\log_2 M$ bits with M the number of targets. This is not the case for the flat sponge claim. As an example, let us take the case of a multi-target preimage attack versus a single-target preimage attack.

- In a (single-target) preimage attack, the adversary is given a n-bit challenge y and has to find an input x such that $\lfloor f(x) \rfloor_n = y$. A random oracle offers n bits of security strength: After N attempts, the total success probability is p with $p \approx N2^{-n}$. So we have that $N/p \approx 2^n$ for $N < 2^n$ and the security strength for a random oracle is n. For KangarooTwelve we claim security strength $\min(n, 128)$ bits in this case.
- In an M-target preimage attack, the adversary is given M challenges, y_1 to y_M, and she succeeds if she find an input x such that $\lfloor f(x) \rfloor_n = y_i$ for any of the challenges. A random oracle with N attempts has success probability p with of $p \approx MN2^{-n}$, and hence $N/p \approx 2^n/M$. So the security strength for the random oracle reduces to $n - \log M$ bits. For KangarooTwelve we claim security strength $\min(n - \log M, 128)$ bits in this case.

Clearly, the reduction in security due to M targets is generic and independent of the security strength. It can be compensated for by increasing the output length n by $\log M$ bits.

4.2 Security of the mode

The security of the mode, or the generic security, relies on both the sponge construction and on the tree hash mode. The latter is Sakura-compatible so that it automatically satisfies the conditions of soundness and guarantees security against generic attacks, see [9, Theorem 1] and [10, Theorem 1]. In both cases, the bottleneck is the ability to generate collisions in the chaining values, or equivalently, collisions of the inner hash function.

The probability of inner collisions in the sponge construction is $N^2/2^{c+1}$, with N the number of blocks [7]. Regarding the collisions in the chaining values of the tree hash mode, the probability is at most $q^2/2^{c+1}$ [10, Theorem 1] with q the number of queries to F. Since each query to F implies at least one block to be processed by the sponge construction, we have $q \leq N$ and we can bound the sum of the two probabilities as $N^2/2^{c+1} + q^2/2^{c+1} \leq N^2/2^{(c-1)+1}$. This expression is equivalent as if c was one bit less than with a single source of collisions, and Claim 1 takes this into account by setting the claimed capacity to $c - 1 = 255$ bits.

We formalize the security of KangarooTwelve's mode of operation in the following theorem. We can see the combination of the tree hash mode and the

sponge construction as applied in KangarooTwelve as a mode of operation of a permutation and call it $\mathcal{K}$.

Theorem 1. *The advantage of differentiating $\mathcal{K}$, where the underlying permutation is uniformly chosen among all the possible 1600-bit permutations, is upper bounded by*

$$\frac{2N^2 + N}{2^{c+1}},$$

with N the number of calls to the underlying permutation.

Proof. By the triangle inequality, the advantage in distinguishing $\mathcal{K}$ calling a random permutation from a random oracle is upper bounded by the sum of two advantages:

- that of distinguishing the tree hash mode calling as inner function a random function $\mathcal{F}$ from a random oracle;
- that of distinguishing the sponge construction calling a random permutation from a random function.

The former advantage is upper bounded by $q^2/2^{c+1}$, where q is the number of calls to $\mathcal{F}$. This follows from Theorem 1 of [10] for any sound tree hash mode, and from Theorem 1 of [9] that says that any Sakura-compatible tree hash mode is sound. We show that the tree hash mode is indeed Sakura-compatible in Section 4.3.

Following Theorem 2 of [7], the latter advantage is upper bounded by $(N^2 + N)/2^{c+1}$. Adding the two bounds and using $q \leq N$ proves our theorem. □

4.3 Sakura compatibility

To show Sakura-compatibility, we use the following terminology. The inputs to the underlying hash function are called *nodes*. Each node consists of one or more *hops*, and a hop is either a chunk of the message or a sequence of chaining values.

The encoding of the nodes follows [9, Section 3.1]:

- When $n = 1$, the tree reduces to a single node. This is the final node, and it contains a single message hop consisting of the input string S followed by the frame bits "message hop" '`1`' and "final node" '`1`'.
- When $n > 1$, there are inner nodes and the final node.
  - Each inner node contains a message hop consisting of a chunk S_i followed by the frame bit "message hop" '`1`'; a simple padding bit '`1`' and "inner node" '`0`'.

- The final node contains two hops: a message hop followed by a chaining hop. The message hop is the first chunk of the input string S_0 followed by the frame bit "message hop" '`1`' and a padding string '`1`'||'0^{62}' to align the chaining hop to 64-bit boundaries. The chaining hop consists of the concatenation of the chaining values, the coded number of chaining values (`length_encode`$(n-1)$), the indication that there was no interleaving ($I = \infty$, coded with the bytes `0xFF`||`0xFF`) and the frame bits "chaining hop" '`0`' and "final node" '`1`'.

4.4 Choice of B

We fix the size of the message chunks to make KangarooTwelve a function without parameters. This frees the user from the burden of this technical choice and facilitates interoperability.

In particular, we chose $B = 8192$. First, we chose a power of two as this can help fetching bulk data 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.

As for the order of magnitude of B, we took into account following considerations. For each B-byte block there is a 32-byte chaining value in the final node, giving rise to a relative processing overhead of about $32/B$. Choosing $B = 2^{13}$, this is only $2^{-8} \approx 0.4\%$.

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%.

There is a tension between a larger B and the exploitable parallelism. Increasing B would further reduce these two overhead factors, but it would also 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 $B/32$ 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 $B/32 = 256$.

4.5 Choice of the number of rounds

Opting for the Keccak-$p[1600, n_\mathrm{r} = 12]$ permutation is a drastic reduction in the number of rounds compared to the nominal Keccak and to the SHA-3

standard. Still, there is ample evidence from third-party cryptanalysis that the switch to Keccak-$p[1600, n_\mathrm{r} = 12]$ leaves a safety margin similar to the one in the SHA-2 functions.

Currently, the best collision attack applicable to KangarooTwelve or any SHA-3 instance works only when the permutation is reduced to 5 rounds [35]. The attack extends to 6 rounds if more degrees of freedom are available and requires a reduction of the capacity from 256 to 160 bits. Preimage attacks reach an even smaller number of rounds [20]. Hence our proposal has a safety margin of 7 out of 12 rounds w.r.t. collision and (second) preimage resistance.

Structural distinguishers is the term used for properties of a specific function that are very unlikely to be present in a random function. Zero-sum distringuishers were applied to the Keccak-$p[1600, n_\mathrm{r}]$ family of permutations in a number of publications [3,13,20]. They allow producing a set of input and of output values that both sum to zero, and this in about half the time it would be needed on a random permutation with only black-box access. These structural distinguishers are of nice theoretical interest, but they do not pose a threat as they do not extend to distringuishers on sponge functions that use Keccak-$p[1600, n_\mathrm{r}]$, see, e.g., [34].

The structural distinguisher on the Keccak sponge function that does reach the highest number of rounds is the keystream prediction by Dinur et al. [16]. It works when the permutation is reduced to 9 rounds, with a time and data complexity of 2^{256}, and allows to predict one block of output. This is above the security claim of KangarooTwelve, but the same authors propose a variant that works on 8 rounds with a time and data complexity of 2^{128}, leaving a safety margin of 4 rounds or 33% for KangarooTwelve against this rather academic attack. Examples of structural distinguishers for the Keccak sponge function with practical complexity and reaching the highest number of rounds are reported by Huang et al. and work up to 7 rounds [21].

In comparison, SHA-256 has a collision attack on 31 (out of 64) steps and its compression function on 52 steps [22,24]. SHA-512's compression function admits collision attacks with practical complexities for more than half of its steps [17].

5 MarsupilamiFourteen

While KangarooTwelve claims a strong notion of 128-bit security strength, and we believe any security beyond it is purely of theoretical interest, some users may wish to use a XOF or hash function with higher security strength. In particular, when defining a cipher suite or protocol aiming for 256-bit security strength, all cryptographic functions shall have at least 256-bit security. Coming forward to such requests, in this section we present a variant of KangarooTwelve with 511-bit claimed capacity.

Addressing a claimed capacity of 511 bits requires an increase of both the capacity in F and length of chaining values in the tree hash mode to at least 512 bits. Taking exactly $c = 512$ bits is sufficient for resisting generic attacks below the claim. As for Keccak-p-specific attacks, the increase of the claimed capacity to 511 bits increases the available budget of attackers and hence reduces the safety margin. In many types of attack, adding a round in the primitive (permutation or block cipher) increases the attack complexity by a large factor. Or the other way round, if one wishes to keep the same safety margin, an increase of the attack complexity must be compensated by adding rounds.

We did the exercise and the result is MarsupilamiFourteen. It has the same specifications as KangarooTwelve, with the following exceptions:

- The capacity and chaining values are 64-byte long instead of 32 bytes. This reduces the sponge rate in F to 136 bytes.
- The number of rounds of Keccak-$p[1600, n_r]$ is 14 instead of 12.
- The claimed capacity in the flat sponge claim is 511 bits instead of 255.

The computational workload per bit is roughly 45% higher than that of KangarooTwelve.

Naturally, even thicker safety margins are achieved with the standard FIPS 202 instances or ParallelHash [29,30].

6 Implementation

We implemented KangarooTwelve in C and made it available in the Keccak code package (KCP) [11]. We now review different aspects of this implementation and its performance.

6.1 Byte representation

KangarooTwelve assumes that its inputs M and C are byte strings. Sakura encoding works at the bit level and adds padding and suffixes so that the input to the function F is a string of bits whose length is in general not a multiple of 8.

It is common practice in implementations of Keccak to represent the last few bits of a string as a *delimited suffix* [11]. The delimited suffix is a byte that contains the last $|X| \bmod 8$ bits of a string X, with $|X|$ the length of X in bits, followed by the delimiter bit '1', and ending with the necessary number of bits '0' to reach a length of 8 bits. When absorbing the last block in the sponge function F, the delimiter bit coincides with the first bit '1' of the pad10*1 padding rule. An implementation can therefore process the first $\lfloor |X|/8 \rfloor$ bytes of the string S

and, in the last block, simply add the delimited suffix and the second bit of the pad10*1 padding rule at the last position of the outer part of the state (i.e., at position $r-1$, with r the rate).

Following the convention in Section 2, the delimited suffix of a string with last bits $(s_0, \ldots, s_{n-1})$ can be represented by the value $2^n + \sum_{i=0}^{n-1} s_i 2^i$ in hexadecimal. For KangarooTwelve, this concretely means that the final node with $\|S\| \leq B$ has suffix '`11`' and delimited suffix `0x07`. With $\|S\| > B$ the intermediate nodes with trailing bits '`110`' use `0x0B` (as depicted in Figure 3), and the final node ending with '`01`' will have `0x06` as delimited suffix.

Fig. 3. Example of delimited suffix

On a similar note, the 64-bit string '$\texttt{110}^{62}$' in the final node is represented by the bytes $\texttt{0x03}||(\texttt{0x00})^7$, still following the convention in Section 2.

This approach is taken by the Internet Research Task Force RFC draft describing KangarooTwelve [37] and in the reference source code in Appendix A.

6.2 Structuring the implementation

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 4. In particular, we instantiate the sponge construction on top of Keccak-$p[1600, n_\mathrm{r} = 12]$ 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-$p[1600, n_\mathrm{r} = 12]$ in parallel; if it is not available and if at least $4B$ bytes are given, it computes $4 \times$ Keccak-$p[1600, n_\mathrm{r} = 12]$ 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.

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) [11].

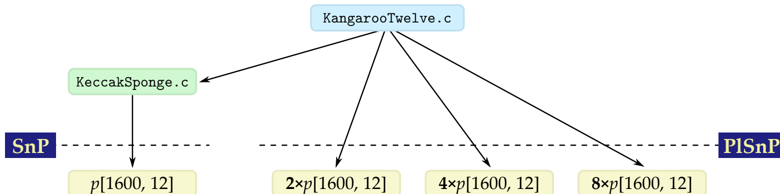

Fig. 4. The structure of the code implementing KangarooTwelve in the KCP.

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, the input bytes are absorbed directly into the state of F as they arrive. Hence, the state reduces to two times the state of F. Of course, if a message is known to be smaller than or equal to B bytes, one could further save one queue.

6.3 256-bit SIMD

Current mainstream PC processors, in the Intel® 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, even on a single core.

On an Intel® Core i5-6500 (Skylake), we measured that one evaluation of Keccak-$p[1600, n_r = 12]$ takes about 450 cycles, while 2 in parallel 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. The time needed to hash messages of length smaller than 168 bytes thus represents the smallest granularity and is reported in Table 1. Note that if many very short messages have to be processed, they can be batched so as to use a parallel implementation. This case is also reported in Table 1.

Macroscopically, when $\|S\| < B$, the time is a straight line with a slope of about 2.89 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-$p[1600, n_r = 12]$.

Table 1. The overall speed for very short messages ($\|S\| < 168$) in cycles, very short messages when batched in cycles/message, for short messages ($\|S\| \leq 8192$) and for long ($\|S\| \gg 8192$) messages in cycles/byte. The figures assume a single core in each case.

Processor	Very short m.	Batched v.s.m.	Short m.	Long m.
Intel®Core i5-4570 (Haswell)	618 c	242 c/m	3.68 c/b	1.44 c/b
Intel®Core i5-6500 (Skylake)	486 c	205 c/m	2.89 c/b	1.22 c/b
Intel®Core i7-7800X (SkylakeX)	395 c	92 c/m	2.35 c/b	0.55 c/b

In our implementation, the final node is always processed with a serial implementation. In principle, a more advanced implementation could process the final node in parallel with the leaves. In more details, it would process the first chunk S_0 in parallel with the first few leaves, and it would buffer about B bytes of chaining values and so as to process them in parallel with leaves. However, at this point, we preferred code simplicity over speed optimization. Similarly, one could in principle remove the peaks of Figure 5 and make it monotonous. It could be achieved 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.

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 4 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 measurements on short messages.

6.4 512-bit SIMD

Recently, Intel® started shipping processors with the AVX-512 instruction set. It supports 512-bit SIMD instructions, enabling efficient implementations of $8 \times$ Keccak-$p[1600, n_r = 12]$. In addition to a higher degree of parallelism, some new features of AVX-512 benefit to the implementation of KangarooTwelve, of ParallelHash 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

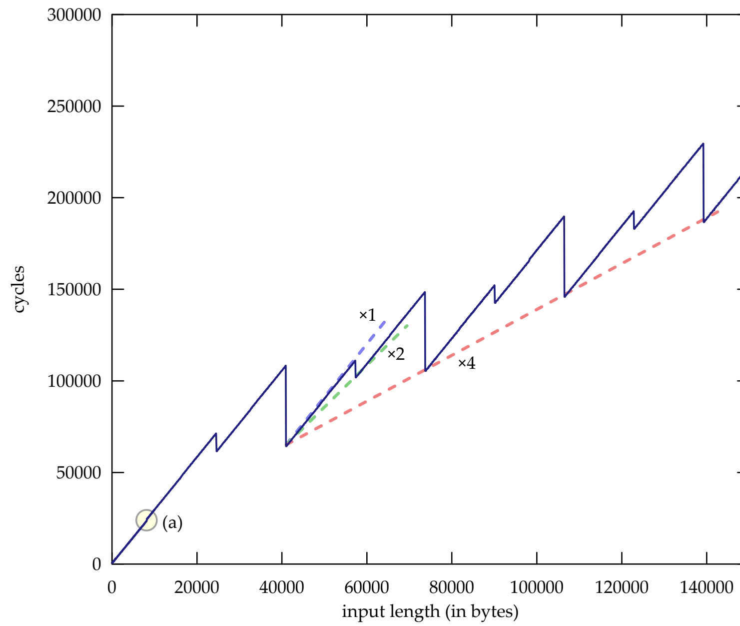

Fig. 5. The number of cycles of KangarooTwelve on an Intel®Core i5-6500 (Skylake) as a function of the input message size.

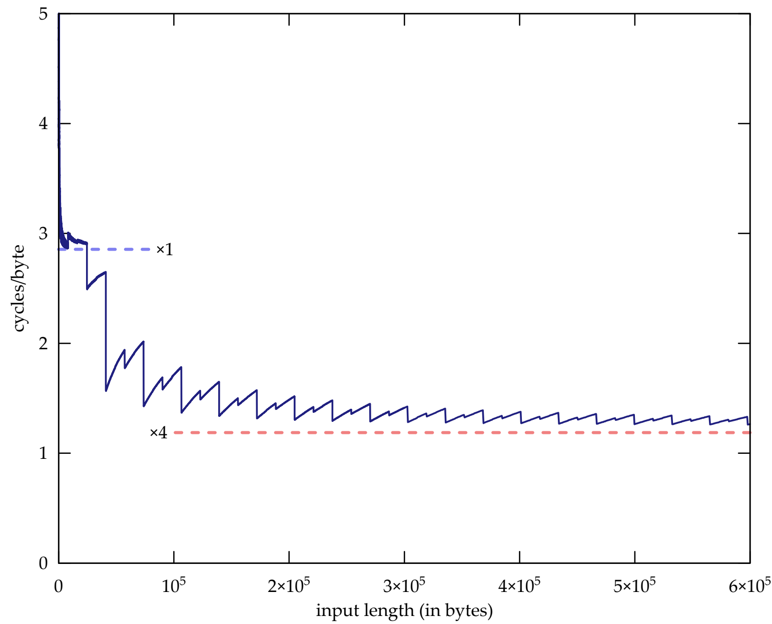

Fig. 6. The number of cycles per byte of KangarooTwelve on an Intel®Core i5-6500 (Skylake) as a function of the input message size.

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 increase the number of registers from 16 to 32. As Keccak-p has 25 lanes, this significantly decreases the need to move data between memory and registers.

We report in Table 1 the speed of our current implementation on a machine equipped with a processor in the Intel® SkylakeX family, supporting this instruction set [11].

6.5 Comparison with other functions

To put the speed of KangarooTwelve in perspective, we compare it to typical hash functions, including the traditional standards MD5, SHA-1 and SHA-2 [33,27,28], the SHA-3 finalists [2,19,38,18], the popular Blake2 functions [4] and some SHA-3 instances [29,30]. For consistency, wherever possible we performed benchmarks on three machines in our possession. Moreover, we cross-checked with the publicly available eBASH results [5] and in case of discrepancy, we selected the fastest. For the traditional hash functions, the fastest implementation often came from OpenSSL [31]. For Blake2, we included some specific AVX2 code by Samuel Neves [26]. Note that the comparison on SkylakeX must be taken with care, as not all implementations available at the time of this benchmarking are fully optimized for the AVX-512 instruction set.

Table 2 shows the results. We first list hash functions that explicitly exploit SIMD instructions with a built-in tree hash mode, such as ParallelHash and Blake2{b,s}p, and compare them to KangarooTwelve for long messages (or when it is used for hashing multiple messages in parallel).

It is interesting to compare the other hash functions to KangarooTwelve when it is restricted to serial processing (as for short messages), to see its speed gain already before the parallelism kicks in. Of course, such a restriction does not exist when hashing a large file, and in practice the comparison should also be made with KangarooTwelve for long messages.

7 Conclusion

KangarooTwelve can be seen as a new member of the Keccak family. It inherits all the properties of the family such as suitability in hardware and resistance against side-channel attacks, but grew up with a strong focus on software performance and interoperability. We tuned the mode and the primitive to offer a tremendous computational speedup in many applications while keeping a comfortable security margin. The latter is confirmed by the cryptanalysis results on Keccak accumulated over the last ten years, which are directly applicable to

Table 2. Speed comparison. All figures are in cycles per byte for long messages, unless otherwise specified.

Function	SkylakeX	Skylake	Haswell
KangarooTwelve	0.55	1.22	1.44
KangarooTwelve ($\leq$ 8KiB)	2.35	2.89	3.68
ParallelHash128	0.96	2.31	2.73
Blake2bp	1.39	1.34	1.37
Blake2sp	1.22	1.29	1.39
SHAKE128	4.28	5.56	7.09
MD5	4.33	4.54	4.93
SHA-1	3.05	3.07	4.15
SHA-256	6.65	6.91	9.27
SHA-512	4.44	4.64	6.54
Blake2b	2.98	3.04	3.08
Blake2s	4.26	4.85	5.34
Blake-256	5.95	6.76	7.52
Blake-512	4.48	5.19	5.68
Grøstl-256	7.24	8.13	9.35
Grøstl-512	9.95	11.31	13.51
JH	13.04	15.14	15.09
Skein	4.48	5.18	5.34

the new sibling. Also, all existing Keccak implementations can be reused with minimal effort thanks to the layered approach in the design. For instance, KangarooTwelve benefits immediately from the new SHA-3 hardware support recently introduced in the ARMv8.2 instruction set [1].

The speedup benefits to both low-end and high-end processors. For the low end, one immediately benefits from the reduction in the number of rounds, and care was taken not to add overhead in the case of short messages.

At the high end, we observed that KangarooTwelve gets significant performance improvements in recent processors, which go beyond the mere gain due to parallelism. Part of these improvements come from the choice of low-latency Boolean operations in the primitive that superscalar architectures can implement efficiently, as demonstrated in the latest Intel®'s SkylakeX processors with the introduction of three-input binary functions.

On such a processor, KangarooTwelve processes long messages at 0.55 cycles/byte. At this speed, it would require only one of its cores to process, in real-time, the output of 10 high-speed solid-state drives (SSD), i.e., a cumulated bandwidth of more than 7 GB/s per core (assuming a clock frequency of 4 GHz). This simply illustrates that with KangarooTwelve the speed of hashing is no longer a bottleneck in software applications.

Acknowledgements

Our implementation for the serial processing is based on the AVX2 code written by Andy Polyakov for OpenSSL. We would also like to thank the anonymous reviewers for their constructive comments.

References

1. ARM corporation, *ARM architecture reference manual ARMv8, for ARMv8-A architecture profile*, document ARM DDI 0487C.a (ID121917), `http://www.arm.com/`.
2. J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, *SHA-3 proposal BLAKE*, Submission to NIST, 2008.
3. J.-P. Aumasson and W. Meier, *Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi*, Available online, 2009, `http://131002.net/data/papers/AM09.pdf`.
4. J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, *BLAKE2: simpler, smaller, fast as MD5*, Applied Cryptography and Network Security (M. J. Jacobson Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, eds.), Lecture Notes in Computer Science, vol. 7954, Springer, 2013, pp. 119–135.
5. D. J. Bernstein and T. Lange (editors), *eBACS: ECRYPT Benchmarking of cryptographic systems*, `http://bench.cr.yp.to`.
6. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, Keccak *specifications*, NIST SHA-3 Submission, October 2008.
7. ———, *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, pp. 181–197.
8. ———, *Cryptographic sponge functions*, January 2011, `https://keccak.team/files/SpongeFunctions.pdf`.
9. ———, *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.
10. ———, *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`.
11. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, Keccak *code package*, June 2016, `https://github.com/gvanas/KeccakCodePackage`.
12. ———, Keccak *third-party cryptanalysis*, 2017, `https://keccak.team/third_party.html`.
13. C. Boura, A. Canteaut, and C. De Cannire, *Higher-order differential properties of Keccak and Luffa*, Fast Software Encryption 2011, 2011.
14. 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.

15. ———, *Improved practical attacks on round-reduced Keccak*, J. Cryptology **27** (2014), no. 2, 183–209.
16. I. Dinur, P. Morawiecki, J. Pieprzyk, M. Srebrny, and M. Straus, *Cube attacks and cube-attack-like cryptanalysis on the round-reduced Keccak sponge function*, Advances in Cryptology - EUROCRYPT 2015 (E. Oswald and M. Fischlin, eds.), Lecture Notes in Computer Science, vol. 9056, Springer, 2015, pp. 733–761.
17. C. Dobraunig, M. Eichlseder, and F. Mendel, *Analysis of SHA-512/224 and SHA-512/256*, Advances in Cryptology - Asiacrypt (T. Iwata and J. H. Cheon, eds.), Lecture Notes in Computer Science, vol. 9453, Springer, 2015, pp. 612–630.
18. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, *The Skein hash function family*, Submission to NIST (Round 2), 2009.
19. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlffer, and S. S. Thomsen, *Grøstl – a SHA-3 candidate*, Submission to NIST (round 3), 2011.
20. J. Guo, M. Liu, and L. Song, *Linear structures: Applications to cryptanalysis of round-reduced Keccak*, Advances in Cryptology - Asiacrypt (J. H. Cheon and T. Takagi, eds.), Lecture Notes in Computer Science, vol. 10031, 2016, pp. 249–274.
21. S. Huang, X. Wang, G. Xu, M. Wang, and J. Zhao, *Conditional cube attack on reduced-round Keccak sponge function*, Advances in Cryptology - Eurocrypt (J.-S. Coron and J. B. Nielsen, eds.), Lecture Notes in Computer Science, vol. 10211, 2017, pp. 259–288.
22. J. Li, T. Isobe, and K. Shibutani, *Converting meet-in-the-middle preimage attack into pseudo collision attack: Application to SHA-2*, Fast Software Encryption (FSE) (A. Canteaut, ed.), Lecture Notes in Computer Science, vol. 7549, Springer, 2012, pp. 264–286.
23. 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.
24. F. Mendel, T. Nad, and M. Schläffer, *Improving local collisions: New attacks on reduced SHA-256*, Advances in Cryptology - Eurocrypt (T. Johansson and P. Q. Nguyen, eds.), Lecture Notes in Computer Science, vol. 7881, Springer, 2013, pp. 262–278.
25. D. Micciancio and M. Walter, *On the bit security of cryptographic primitives*, Eurocrypt, 2018, to appear.
26. S. Neves, *BLAKE2 AVX2 implementations*, `https://github.com/sneves/blake2-avx2`.
27. NIST, *Federal information processing standard 180-1, secure hash standard*, April 1995.
28. ———, *Federal information processing standard 180-2, secure hash standard*, August 2002.
29. ———, *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`.
30. ———, *NIST special publication 800-185, SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash*, December 2016, `https://doi.org/10.6028/NIST.SP.800-185`.
31. OpenSSL community, *OpenSSL – cryptography and SSL/TLS toolkit*, `https://github.com/openssl/openssl`.

32. 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.
33. R. Rivest, *The MD5 message-digest algorithm*, Internet Request for Comments, RFC 1321, April 1992.
34. D. Saha, S. Kuila, and D. R. Chowdhury, *Symsum: Symmetric-sum distinguishers against round reduced SHA3*, IACR Trans. Symmetric Cryptol. **2017** (2017), no. 1, 240–258.
35. L. Song, G. Liao, and J. Guo, *Non-full sbox linearization: Applications to collision attacks on round-reduced Keccak*, Advances in Cryptology - CRYPTO 2017 (J. Katz and H. Shacham, eds.), Lecture Notes in Computer Science, vol. 10402, Springer, 2017, pp. 428–451.
36. ______, *Solution to the 6-round collision challenge*, 2017, `https://keccak.team/crunchy_contest.html`.
37. B. Viguier, *KangarooTwelve*, Internet Research Task Force draft, March 2018, `https://datatracker.ietf.org/doc/draft-viguier-kangarootwelve/`.
38. H. Wu, *The hash function JH*, Submission to NIST (round 3), 2011.

A Reference source code

In this section, we give (unoptimized) reference code written in Python 3, which can also be downloaded from the KCP [11]. 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 `length_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)]
            # 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 $F = \textsc{sponge}[\textsc{Keccak}\text{-}p[1600, n_r = 12], \text{pad}10^*1, r = 1344]$

```
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 `length_encode` function and KangarooTwelve

```
def length_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 + length_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) \
            + length_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 byte string of length 17^i for $0 \leq i \leq 6$. The value M is constructed by repeating the pattern $\texttt{0x00}, \texttt{0x01}, \texttt{0x02}, \ldots, \texttt{0xFA}$ as many times 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
```