

Asymptotic Analysis of Plausible Tree Hash Modes for SHA-3

Kevin Atighehchi¹ and Alexis Bonnetcaze²

¹ Aix Marseille Univ, CNRS, LIF, Marseille, France
`kevin.atighehchi@univ-amu.fr`

² Aix Marseille Univ, CNRS, I2M, Marseille, France
`alexis.bonnetcaze@univ-amu.fr`

Abstract. Discussions are currently underway about the choice of a tree hash mode of operation for a standardization. It appears that a single tree mode cannot address the specificities of all possible uses and specifications of a system. In this paper, we review the tree modes which have been proposed, we discuss their problems and propose remedies. We make the reasonable assumption that communicating systems have different specifications and that software applications are of different types (securing stored content or live-streamed content). More particularly, we propose modes of operation that address the memory usage problem. When designing a parallel algorithm, one major question is how to improve the running time (using as many processors as we want) while minimizing the required memory of an implementation using as few as one processor. Conversely, an interesting question is how to obtain a near-optimal running time while containing the memory consumption.

Keywords: SHA-3, Hash functions, Sakura, Keccak, SHAKE, Parallel algorithms, Merkle trees, Live streaming

1 Introduction

1.1 Context

Hash functions are widely used in domains like cryptography or, more generally, computer science. They are often used in data structures like bloom filter or hash table, and they require to meet certain properties which depend on the intended application. In this article, we are interested in cryptographic hash functions that must satisfy the usual properties of *pre-image resistance*, *second pre-image resistance* and *collision resistance*. In addition to these specific properties, hash functions are generally required to be efficient.

Historically, a cryptographic hash function makes use of an underlying function, denoted f , having a fixed input size, like a compression function or a block cipher. Recently the use of a permutation [8, 7] as underlying function has also been proposed. This underlying function, so-called *inner* function according to the terminology of Bertoni et al. [8], is called iteratively on the message blocks in

order to process a message of arbitrary length. When this mode of operation is sequential, it makes it difficult to exploit parallel architectures. In fact, a sequential (or serial) hash function can only use Instruction-Level Parallelism (ILP) and Single Instruction Multiple Data (SIMD) [18, 19], because the amount of computation which can be done in parallel between two consecutive synchronization points is too small.

Some operating modes use hash functions as inner functions and can exploit a particular tree structure either for *parallelism* or *incrementality*³ purposes. A (non-degenerated) tree structure not only allows for further use of SIMD instructions, but also permits the use of multithreading in order to process in parallel several parts of a message on different processors/cores. Several conventions exist to describe tree structures in the context of hashing. The first convention, denoted C1 and often used to deal with Merkle trees, consists in considering a node as the result of f applied on the concatenation of its children. A leaf is the result of the inner function applied on an individual block of the message. A second convention, denoted C2, is a variant of C1 in which the leaves are simply the blocks of the message. The term *hash tree* is used to refer to this type of tree, where the nodes are the f -images and the leaves are the blocks of the message. The last encountered convention [6, 10], denoted C3, considers the nodes as being the inputs of the inner function. Throughout this paper, unless otherwise specified, we use the convention C3. Thus, a node is an f -input, while a chaining value is an f -image. The tree height, denoted h , corresponds to the number of levels, indexed from 1 (for the base level) up to h (for the final level). For the sake of simplicity, a node contains either chaining values or message blocks, but not both. A level is said to be of arity a if all its nodes contain exactly a chaining values (or message blocks), except the rightmost node whose arity may be smaller. Although the present paper deals essentially with hashing modes using only f as operator, we will also explain at the end the benefits of using additional operators from the parallelism standpoint.

In this article, we focus on the efficiency of hash functions, depending on the chosen modes. Instead of working at finite distance, we choose to consider asymptotic complexities as it is usually done in algorithmic (*e.g.*, sorting, exponentiation, etc). Let M be a message of n blocks, each block being of fixed-length N . Since a sequential hash function iterates a “low-level” primitive on fixed-size blocks of the message (with maybe a constant number of added blocks for padding or other coding purposes), its running time is asymptotically linear in n . In terms of memory usage, such a function needs to store only one hash state. The hash state size corresponds, at a given point in time, to the amount of bits processed by the inner function. Depending on the type of inner function, this size corresponds to the width of the underlying iterated primitive. For instance, in the case of Keccak [8], the hash state size corresponds to the width of the underlying permutation.

³ Using a balanced binary tree is particularly efficient when we have to update the hash of an edited message. For the change of one block in the message, we update the digest in logarithmic time.

1.2 Tree hash modes

Tree hash modes have been proposed in the SHA-3 candidates Skein [15] and MD6 [28], and also in BLAKE2 [2]. These tree modes are slightly parameterizable since the arity of the tree can be chosen. Better still, in Skein, the node arities are slightly more customizable: a parameter λ_{in} indicates that the inner nodes (*i.e.* nodes of level ≥ 2) are of arity $2^{\lambda_{in}}$ and a parameter λ_{leaf} indicates that the base level nodes each contain $2^{\lambda_{leaf}}$ message blocks. Skein, BLAKE2 and MD6 have also a parameter restricting the tree height. If this last parameter has a too small value, the root node can have an arity proportional to the size of the message.

Bertoni *et al.* [6, 10] give sufficient conditions for a tree-based hash function to ensure its indistinguishability from a random oracle. They define the Sakura coding [9] which ensures these conditions, and allows any hash algorithm using it to be indistinguishable from a random oracle, automatically.

They also propose several tree hash modes for different usages. We can compare the efficiency of these algorithms using Big-O notation. For example, a mode – called Mode 1 in this article (and “final node growing” in [9]) – can make use of a tree of height 2, defined in the following way: the message is divided into fixed-size chunks which have to be hashed separately. The hash computations are distributed among the processors, and the concatenation of the resulting digests is hashed (sequentially) by a single processor. The advantages of this mode is its scalability (the number of processors can be linear in the number of blocks) and its reduced memory usage when executed sequentially. Its drawback is its ideal running time which remains linear in the message size. In Mode 2, the message is divided into as many parts (of roughly equal size) as there are processors so that each processor hashes each part, and then the concatenation of all the results is sequentially hashed by one processor. In order to divide the message into parts of roughly equal size, the algorithm needs to know in advance the size of the message, which limits its use to the hashing of stored (or streamed stored) contents. Bertoni *et al.* use an idea from Gueron [17] to propose a variant (Mode 2L) which still makes use of a two-level tree and a fixed number of processors, but this one interleaves the blocks of the message. This interleaving, which consists in distributing the message bits (or blocks) in a *round-robin* fashion among q clusters, has several advantages. It allows an efficient parallel hashing of a streamed message, a roughly equal distribution of the data processed by each processor in the first level of the tree (without prior knowledge of the message size), and finally a correct alignment of the data in the processors’ registers (for SIMD implementations). The major drawback of this interleaving is that the memory consumption is $O(q)$ if the message bits have to be processed in the order of their arrival, no matter whether this tree hash function is sequentially implemented or not. Finally, the classic binary tree, in Mode 3, offers the best ideal running time but it consumes a lot of storage when executed by a single processor. Table 1 compares the efficiency of these algorithms.

Mode	Live streaming	Memory (sequential)	Memory (with p processors)	Parallel running time (ideal case)	Comments
1	✓	1	p	n	
2S	-	1	p	n/q	not scalable
2L	✓	q	q		
3	✓	$\log n$	$p \log n$	$\log n$	

Table 1. Asymptotic efficiency using Big-O notation of existing tree hash modes, where n is the number of blocks of the message. The “ideal” parallel running time refers to the running time when we have the maximum allowed number of processors. Mode 2 is dedicated to a “fixed” number q of processors (with the assumption that $q \geq p$). Its asymptotic efficiency is given without hiding the quantities p , q and $1/q$ in the Big-O.

There is a continuing debate [23] about the way of standardizing tree hash modes. On the one hand, some would like to have a single (and simple) tree hash mode allowing unrestricted depth⁴ (like Skein [15], MD6 [28] or BLAKE2 [2]), with maybe several sets of parameters for the node arities. These tree topologies are flexible and have a good potential parallelism, because they support live streaming, are scalable and allow a nice ideal speedup (in running time). The problem is that, when its height is unbounded, such a tree brings a performance penalty for sequential execution, as the amount of *work* (*i.e.* computations) to do is much greater than for a serial (traditional) hash function. Note that the asymptotic efficiency of such a tree is the same as that of Mode 3. However, if a parameter restricts its height (as allowed by Skein or MD6), its asymptotic (parallel) efficiency can fall into the case of Mode 1.

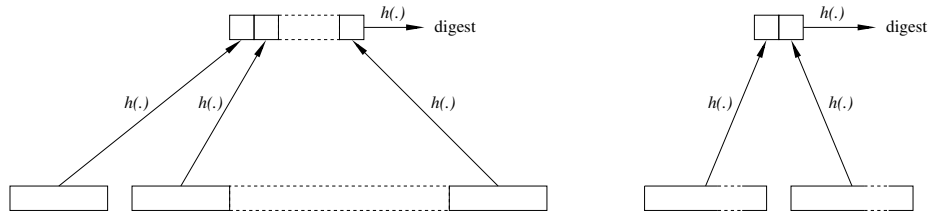


Fig. 1. Illustration of Mode 1 (on the left side) and Mode 2 (on the right side). On the left side, the message is divided into chunks of fixed size, while on the right side it is divided into two chunks of roughly the same size. In this example, Mode 2 is dedicated to the use of two processors.

⁴ The term unrestricted can be misused. For some of these modes, there is a parameter defining the depth of the tree, and this one can be set large enough so that, in practice, it does not have any impact on the tree topology.

On the other hand, some argue that there should be as many tree modes as application usages. According to Kelsey [21], there should be two standards: one standard for parallel hashing and one standard for tree hashing. The standard for tree hashing would focus on trees of arbitrary (unrestricted) depth, with small node arities. These tree topologies are suitable for *timestamping*, *authenticated dictionaries* or *Merkle signatures* (and their variants). The standard for (fast) parallel hashing would focus on trees having a small height, because the evaluation of a hash function should remain efficient on resource-constrained machines (having few memory and maybe a single processor). Indeed, as we will see later, the memory consumption is linear in the tree height. Moreover, a small tree height means a reduced amount of *work*. The modes discussed for this last standard correspond to the two variants of Mode 2.

Thus, the discussed modes are roughly the ones summarized in Table 1. Even if it is scalable and allows an optimal running time (in ideal conditions), Mode 3 seems to be left aside. It is just recommended for *incremental hashing*, or for the other cited cryptographic algorithms (or protocols). Regarding the other proposed modes (an illustration is depicted Figure 1), it seems that a choice has to be made between scalability (Mode 1) and a reduced sequential part of the computation (the root node computation in Mode 2). In practice, for a small number of processors, this makes a difference. Asymptotically, in either case, the running time is still linear in the size of the message.

1.3 Our contributions

This paper first describes strategies for sequential and parallel implementations of tree hash modes, with particular attention to the memory consumption. It then proposes several scalable tree modes using Sakura coding [9]. These modes, whose complexities are summarized in Table 2, address the memory usage problem:

- We show how to parameterize the tree topology and give 6 modes suitable for the hashing of (streamed or not) stored content.
- Then, we show that it is interesting to have a sequence of increasing levels arities. This leads us to propose 3 modes suitable for the hashing of live-streamed content. While at first glance this seems somewhat contradictory, we show that without knowing in advance the size of the message, these 3 adaptive tree constructions actually lead to different asymptotic complexities.
- We discuss the way of decreasing the number of processors required to obtain the ideal (asymptotic) parallel running time.
- We give some guidelines for the use of interleaving.
- We make suggestions for generating a digest of arbitrary length using a parallel construction.

Our modes	Live streaming	Memory usage (sequential)	Memory usage (with p processors)	Parallel running time (ideal case)
4	-	1	p	n^ϵ
4L	✓			
5	-	$\log \log n$	$p \log \log n$	$n^{\frac{1}{\log \log n}} \log \log n$
6S	-	$\sqrt{\log n}$	$p\sqrt{\log n}$	$n^{\frac{1}{\sqrt{\log n}}} \sqrt{\log n}$
6L	✓			
7S	-	$\frac{\log n}{\log \log n}$	$p \frac{\log n}{\log \log n}$	$\frac{\log^2 n}{\log^2 \log n}$
7L	✓	$\frac{\log n}{\log \log \log n}$	$p \frac{\log n}{\log \log \log n}$	$\frac{\log n \log \log n}{\log \log \log n}$
8S	-			
8L	✓			
9	-	$\frac{\log n}{\log \log n}$	$p \frac{\log n}{\log \log n}$	$\log^{1+\epsilon} n$

Table 2. Asymptotic efficiency (using Big-O notation) of our tree modes, where n is the number of blocks of the message and ϵ is a positive constant, such that $\epsilon < 1$. The “ideal” parallel running time refers to the running time when we have the maximum allowed number of processors.

1.4 Organization of the article

After this brief survey and summary of our results, the paper is organized as follows. Section 2 contains background information regarding hash functions and tree hash modes. We discuss their security, implementation strategies and their time-space efficiency. Using a parameterizable tree hash mode described in Section 3, we derive several modes addressing the memory usage problem. In particular, Section 4 gives parameters that produce tree topologies suitable for streamed stored content. Then, parameters suitable for live-streamed content are given in Section 5. Finally, we discuss in Section 6 how we can conciliate scalability and interleaving, and concluding remarks are given in the last section.

2 Preliminaries

2.1 Security

Bertoni *et al.* [6, 10] give some guidelines to design correctly a tree hash mode τ operating an inner hash (or compression) function f . They define three sufficient conditions which ensure that the constructed hash function τ_f , which makes use of an ideal hash (or compression) function f , is indifferentiable from an ideal hash function. They propose to use in nodes particular frame bits (*i.e.* meta information bits) in order to meet these conditions. We refer to [6, 10] for the detailed definitions, and we give here a short description for each of them:

- *message-completeness*: Given all the nodes (*i.e.* f -inputs), we are able to uniquely determine the message. A deterministic algorithm should be able to do that in a time linear in the number of bits in the tree.

- *final-node-separability*: There must be a domain separation between the root node and any other nodes. For instance, a different prepended (or appended) bit in each of these two cases can be used to differentiate them.
- *tree-decodability*: Ensuring this property means several things. For any tree of nodes generated by the mode of interest, we should be unable to extract a *final subtree* (*final* in the sense that this subtree contains the original root node) which could have been generated legitimately by this mode. We can describe a deterministic algorithm whose running time is linear in the number of bits in the tree and which performs the following: it decodes all the nodes (*i.e.* *f*-inputs) in order to retrieve message bits, chaining value bits and frame bits. Moreover, given a tree of nodes as input, if it detects that it has been generated legitimately by the mode, it returns *compliant*. If it detects that it can be extracted from a tree legitimately generated by the mode, it returns *final-subtree-compliant*. Otherwise, it returns *incompliant*.

These conditions ensure that no weaknesses will be introduced when using the inner function. For instance, with *tree-decodability*, an inner collision in the tree is impossible without a collision for the inner function. Andreeva *et al.* have shown in [1] that a hash function indiffereniable from a random oracle satisfies the usual security notions, up to a certain degree, such as pre-image and second pre-image resistance, collision resistance and multicollision resistance.

In the modes we propose, we use Sakura coding which is specified with an ABNF grammar [9]. Sakura allows any tree-based hash function using it to be automatically indiffereniable from a random oracle, without the need of further proofs. The coding used in a node depends on some information about it. For instance, if a node has children, the information about their number is encoded inside it using Sakura.

2.2 Implementation strategies and complexities

Sequential hash function. Since a hash function processes a number of bits which is a multiple of a certain block size N , its time complexity behaves like a staircase function of this number of bits. We say that the time complexity of an iterated hash function f for the operation $f(x)$ can approximately be described as a function of its input size l (in number of blocks) by the function

$$T(l) = a \cdot l + b$$

where a and b are constants which depend on the choice of the hash function and its parameters. For instance, we can use the Keccak algorithm [8] which is based on a sponge construction [5] and is the winner of the SHA-3 competition [27]. The two important parameters of this sponge construction are the rate r and the capacity c . This construction uses a permutation P to process a state S of $r + c$ bits at each iteration, and is divided in two phases: the absorbing phase which processes the message blocks and the squeezing phase which generates the hash output. In the absorbing phase, the rate r corresponds to the speed of the

processing of the message bits, while in the squeezing phase it corresponds to the speed of the generation of the hash output bits. In the first phase, the state S is initially 0. At each iteration, a block of size r bits from the padded message is XORed with the first r bits of S , and P is applied on S to obtain the new state S' . The squeezing phase starts once all the message blocks have been processed, and, since the output bits are extracted from the first r bits of the state, this state is transformed via P as many times as needed to extract bits to complete the hash output (possibly of size $> r$). The collision resistance and pre-image resistance strengths are related to the bit-size $c/2$. Throughout the paper, we suppose the use of the *extendable-output function* RawSHAKE256 which needs a capacity of 512 bits and which, according to the standard FIPS 202 [25], can have a state size of 1600 bits and subsequently a rate of 1088 bits. For its use in tree hash algorithms, Bertoni *et al.* [9] suggest to set its digest size equal to the capacity c .

Tree hashing. A tree hash mode uses an inner function f to compute the hashes of nodes based on the values of their children. Depending on the target application, the result can simply be the final digest of the hash tree (*i.e.* the hash of the root node), or all the computed f -images. Tree hashing is due to Merkle and Damgård [14, 24] and has several applications: *Post-Quantum Cryptography* with Merkle signatures, *Incremental Cryptography*, *Authenticated Dictionaries*, and the field we are concerned with here, *Parallel Cryptography*. A hash tree can be evaluated sequentially (*i.e.* with a single processor), or in parallel by using SIMD instructions or *multithreading* (using several processors/cores). The time and space complexities of a sequential execution of the tree mode are particularly important for resource-constrained systems.

Memory usage. The memory space used by the execution of a hash function can be divided into two quantities, the space used by the message to hash, and the *auxiliary space* used to execute the function on the message. This last one is particularly important for memory-constrained devices. Besides, in the case of streaming applications, a message can be processed by a system as it arrives without being stored. In such a case, the total memory space used is approximately reduced to the *auxiliary space*. In this paper, we refer to the *auxiliary space* when speaking of memory usage. A sequential hash function needs to store $\Theta(1)$ hash states in memory. For evaluating f on all the nodes of a tree of height h , a sequential implementation needs to store $\Theta(h)$ hash states in memory, regardless of the node arities. This memory consumption is due to a *highest node first* strategy, which consists to apply f on the highest node first. For a classic k -ary Merkle tree where k is a small number, this node is the highest node that has all its chaining values ready (*i.e.* instantiated by f) to be processed. This strategy is particularly used in Merkle tree traversal techniques [29, 13]. For a tree having nodes of very high arity (*e.g.* of arity a , possibly dependent of the message size), this strategy has to be changed in the following way: start (or continue) to evaluate f on the highest node that has d chaining values not yet

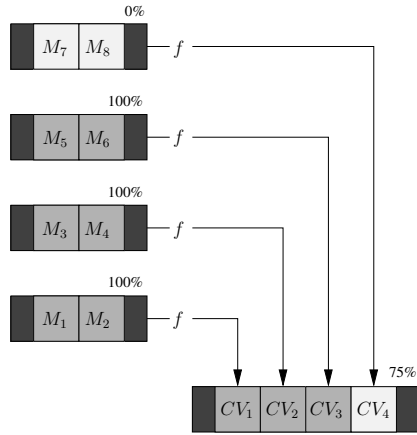


Fig. 2. Application of the variant of the *highest node first* strategy. We represent the subtree of nodes containing the first 8 blocks of the message. Levels 1 and 2 are represented from left-to-right and nodes at a same level are represented vertically from bottom-to-top. In these nodes, black blocks contain frame bits and grey blocks message blocks or chaining values. For reasons of simplicity, it is assumed that the cost to process frame bits is negligible and that one message block (or chaining value) is processed per unit of time. Percentages at the right corner of a node indicates the progression of its consumption by f , and dark grey blocks indicate which data have been used. We also suppose that $d = 1$, meaning that the result of f applied on a node is immediately used to advance the computation of its parent node. As a result, when 9 units of time have elapsed, the node at level 2 has been consumed by f up to the three-quarter mark.

consumed (with d a constant much lower than a). Thus, this number d serves as a threshold value to trigger the (continuation/progression of the) processing by f of the node. With such a variant, there is no need to wait until all the children be processed to process their parent node, and, as a consequence, there can at most be d hash states in memory per level. An example is depicted Figure 2.

Multithreading. Having multiple processors/cores, we would like to use them to improve performances. We need to employ *multithreading* to distribute, by means of *working threads*, the parallel computations among these processors. We assume here that we have p processors and that we use a fixed thread pool containing p threads (one thread per processor). Multithreaded implementations are very efficient if the threads do not need to communicate and/or synchronize, or as little as possible. Synchronization delays are indeed very time-consuming. Depending on the scheduling strategy used, tree hashing can require a lot of synchronizations. Many situations have to be explored:

- If the message to hash is already available (*i.e.* locally stored on the system), we recommend to assign⁵ to each thread one of the p biggest subtrees (of

⁵ Assigning the processing of a subtree to a processor means that this latter is responsible for evaluating f on all its nodes.

roughly the same size). If the tree topology makes it hard to find p biggest subtrees covering the whole message, an other solution is to seek the highest level having a number of nodes $q \geq p$, and to assign to each thread $\lceil q/p \rceil$ of the q subtrees rooted at these nodes. For the rest of the nodes, the method can vary. For instance, we can distribute the evaluations of f on the remaining nodes as fairly as possible between the processors (at the cost of some synchronizations), or merely perform these computations sequentially. Such a strategy reduces greatly the number of synchronization points, and thus improves performances. Note that a thread processes its subtree as done by a sequential implementation, *i.e.*, using the *highest node first* algorithm described above.

- If the message is received via a streaming system (no matter if it is stored or generated on the fly on the remote server), the allocation strategy is necessarily fine-grained, with a grain size depending on the bandwidth with which the message is received. Thus, compared to the previous case, such a parallel implementation has to cope with more synchronization delays. Assuming that the link bandwidth is not a problem, we describe here a scheduling strategy for the simple case where the tree arity a is small. This one could be named *higher level first* as it is similar to the strategy used in a sequential implementation. For a set of p threads, at each level we use a buffer which can receive pa values (they are message blocks or chaining values). At any time, the threads are working at a same level of the tree and their goal is to fill up as soon as possible the highest level buffer. On a same level these threads have to apply f on pa nodes in order to move up and push p chaining values in the buffer of the next level. Once these p chaining values are computed, the buffer at the level below is flushed (*i.e.* the pa just processed chaining values are removed). If the current level occupied by threads is greater than 1 and the lack of resources on the level below prevents them from filling up the current buffer (*i.e.* they cannot finish the computation of the pa values), then they return down to level 1, otherwise they continue, and so on. We let the reader deduce a termination phase for the end of the message, where buffers' contents of less than pa values have to be processed. Note that depending on the throughput with which the message is received, this strategy could be adapted to process subtrees instead of nodes with the aim of increasing the amount of work done between two synchronization points. In other words, there could be a trade-off between the algorithm we just described and the one presented at the first point above. An other simple alternative to increase the amount of work between two synchronization points is to consider buffers of pak values, with a parameter $k > 1$ to select appropriately.
- There are possibly other situations in which the message blocks are not received in the order. They could be interleaved in a certain way for the need of the hash function. In this paper, we do not discuss further more such a situation which seems unreasonable from a *transport layer* standpoint (*e.g.*, in the OSI or DOD layered model for network protocol).

For processing a tree of height h , a parallel implementation using p processors requires $O(ph)$ hash states in memory.

SIMD implementations. The single-instruction multiple-data (SIMD) units are present in a modern x86 processor or core. Well known instruction sets are MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4, AVX and AVX2. These units can apply a same instruction on several data simultaneously. They are thus very useful to compute several instances of a hash (or compression) function in parallel. The size of SIMD registers determines how many parallel hash (or compression) functions can be evaluated.

Interleaving. It is very useful for streamed content and serves two purposes: first, for multithreaded implementations, it allows the feeding of each processor as soon as possible. If each processor is responsible for the computation of a node, *interleaving* allows the distribution of the message bits/blocks among the nodes in a *round-robin* fashion. Second, for SIMD implementation, it allows a perfect alignment of the message blocks in the processor registers when loading the message sequentially. To the best of our knowledge, *interleaving* has been proposed to be used in a 2-level tree (Mode 2L of Table 1), under the name of *j-lanes hashing* [17]. In this proposition, the number of nodes of the first level (*i.e.* lanes) is a fixed parameter. Its use in trees of height $O(h)$, where h is a function of the message size, is an interesting question that will be discussed in Section 6.

3 A parameterizable tree hash mode

We remind the reader that the number of hash states in memory corresponds, in the worst case, to the height h of the tree. Besides, if we denote by u_i the biggest arity at level i , for $i = 1 \dots h$, the “ideal” parallel running time is (asymptotically) a linear function of $\sum_{i=1}^h u_i$. In what follows, we present a tree hash mode using Sakura which, when the parameters are adequately chosen, produces modes offering interesting trade-offs between memory consumption and parallel running time. The parameters suitable for streaming stored content and streaming live content will be discussed in Section 4 and Section 5 respectively.

3.1 Notations

Let f be a hash function which takes as input a message M of an arbitrary length, and maps it to an output $f(M)$ of a fixed bit-length N . More concretely, f can be the intermediate *extendable-output function* RawSHAKE256 [25] with a digest size of $N = 512$ bits. Beforehand, the following notations are used in the description of the hashing mode:

- The operator \parallel denotes the concatenation.

- $\text{I2OSP}(x, xLen)$ is a function specified in the standard PKCS#1 [22]. It converts a non-negative integer, denoted x , into an byte stream of specified length, denoted $xLen$.
- $\{xLen\}$ is a single byte that represents the binary encoding of $xLen$. The arity x of a node is encoded by using $\text{I2OSP}(x, xLen)$ and $\{xLen\}$, where $xLen$ is appropriately chosen.
- The encoding of the arity x is defined as:

$$\text{enc}(x) := \text{I2OSP}(x, \lfloor \log_{256} x \rfloor + 1) \| \{ \lfloor \log_{256} x \rfloor + 1 \}.$$

- 0^* indicates a non-negative number of bits 0 to be used in an f -input for padding, for alignment purposes. We will assume that this is the minimum number of bits 0 so that the bit-length of the f -input is a multiple of a word bit-length (*i.e.* 32 or 64 bits).
- I is the *interleaving block size* of a node that determines how the message bits are distributed over its children. To the first child are given the first I bits, to the second child is given the second sequence of I bits and so on. After reaching the last child, we return to the first child, and so on. A child node (of a node having an *interleaving block size*) can have its own *interleaving block size*, meaning that the bits it is responsible for are distributed over its children according to this attribute. This process can be repeated recursively if several generations of descendants have their own *interleaving block size*. The notation I_∞ means the absence of interleaving.
- $\{I\}$ represents two bytes that encode I . It is defined with a floating point representation, with one byte for the mantissa and one byte for the exponent. For the absence of interleaving, $\{I_\infty\}$ is represented (using hexadecimal notation) with the coded mantissa 0xFF and the coded exponent 0xFF. We refer to the Sakura specification [9] for further information.

These encoded elements serve to identify the type of node and to delimit the embedded *message hop*, *chaining hops* or *kangaroo hops*, in accordance with Sakura coding [9]. These hops embed useful data (message blocks and chaining values). The term *payload* of a node refers to the list of its embedded message blocks and chaining values.

3.2 The tree hash mode

Given a message M of bit-length $|M|$ and a sequence of arities $(u_i)_{i \geq 1}$, SHAKE256 can be defined with the following tree hash mode using Sakura:

1. Let $l_0 = \lceil |M|/N \rceil$ and $M_0 = M$. The quantity l_0 is the number of blocks of M , where the last block may be shorter than N bits. We set $i = 0$.
2. We split M_i into blocks $M_{i,1}, M_{i,2}, \dots, M_{i,l_{i+1}}$ where:
 - (1) $l_{i+1} = \lceil l_i/u_{i+1} \rceil$;
 - (2) all blocks but the last one are $u_{i+1}N$ bits long and the last block may

be shorter than $u_{i+1}N$ bits.

We set the node arities $u_{i,j}$ as follows:

$$u_{i,j} = \begin{cases} u_{i+1} & \text{for } j = 1 \dots l_{i+1} - 1, \\ l_i - u_{i+1} \lfloor l_i / u_{i+1} \rfloor & \text{for } j = l_{i+1}. \end{cases}$$

Then, we check certain conditions to apply Sakura coding correctly:

- If $i = 0$ and $l_{i+1} \geq 2$, we compute the message

$$M_{i+1} := \prod_{j=1}^{l_{i+1}} f(M_{i,j} \| 110^*0).$$

- If $i = 0$ and $l_{i+1} = 1$, we compute the message

$$M_{i+1} := f(M_{i,1} \| 11).$$

- If $i > 0$ and $l_{i+1} > 1$, we compute the message

$$M_{i+1} := \prod_{j=1}^{l_{i+1}} f(M_{i,j} \| \text{enc}(u_{i,j}) \| \{I_\infty\} \| 010^*0).$$

Remark: the number of children of the rightmost node may be lower than u_i . For implementation purposes, it is suggested (in [9]) that the number of padding bits for the rightmost f -input be such that all the f -inputs, at this level of the tree, have same length.

- If $i > 0$ and $l_{i+1} = 1$, we compute the message

$$M_{i+1} := f(M_{i,1} \| \text{enc}(u_{i,j}) \| \{I_\infty\} \| 01).$$

3. We set $i = i + 1$. If $l_i = 1$, we return the hash value M_i . Otherwise, we return to step 2.

Using the proposed mode, we remark that

$$\lceil \lceil \dots \lceil \lceil n / u_1 \rceil / u_2 \rceil \dots \rceil / u_i \rceil = \lceil n / (u_1 u_2 \dots u_i) \rceil$$

for a sequence of (strictly) positive integers $(u_j)_{j=1\dots i}$. Consider a sequence of arities $(u_j)_{j \geq 1}$. At level i , there are exactly $\lceil n / (u_1 u_2 \dots u_i) \rceil$ nodes. If this sequence has an increasing number of terms greater than or equal to 2, then there exists a non-zero positive integer h such that

$$\prod_{j=1}^h u_j \geq n$$

where n is the number of blocks of the message. This ensures that we obtain a tree structure since, at level h , it remains a single node, the final (root) node. The problem is to find a sequence of arities $(u_j)_{j \geq 1}$ such that the tree height

h is $O(g_1(n))$ and $\sum_{j=1}^h u_j$ is $O(g_2(n))$, where $g_1(n)$ and $g_2(n)$ are the desired complexities. Indeed, the memory usage of a sequential implementation and the ideal parallel running time are related to these two quantities. Note that the ideal parallel running time is accurately related to $\sum_{i=1}^h (a \cdot u_i + b)$ with $u_i \geq 1$, and a and b strictly positive integer constants.

Note that this parameterizable mode is as simple as possible and that certain computational overheads are not avoided. In particular, all the leaves are at the same depth. To decrease slightly the number of calls to the underlying permutation, the concept of kangaroo hopping⁶ [9] can be used. Such an optimization reduces the computational costs of a mode at finite distance but does not change its asymptotic behaviour⁷.

In the following sections, we give sets of parameters with the goal of obtaining interesting trade-offs in terms of both time and space complexities.

4 Parameters for streaming stored content

The Internet is recognized as a network interconnecting heterogeneous computers, and this becomes particularly true with the advent of the Internet of Things (IoT). When choosing a hashing mode, we have to determine what kind of devices will process the message (*e.g. by checking its integrity*). According to [20], IoT devices can be classified in two categories, based on their capability and performance: *high-end IoT devices* which regroup single-board computers and smartphones, and *low-end IoT devices* which are much more resource-constrained. Based on the memory capacity of its devices, this last category has been further subdivided into three categories (denoted *Class 0*, *Class 1* and *Class 2* by increasing order of memory capacity [12]) by the Internet Engineering Task Force (IETF). We refer to [20] for further information.

When we have to hash a stored file or a streamed stored media, its size is known in advance. We can then use this information to define a finite sequence of arities $(u_i)_{i=1\dots h}$ with $u_i = a$ for all $i \in \llbracket 1, h \rrbracket$. We give six interesting pairs (a, h) of parameters:

- **Mode 4.** $a = \lceil n^\epsilon \rceil$ and $h = \frac{1}{\epsilon}$, with a positive constant $\epsilon < 1$ such that $1/\epsilon$ is a strictly positive integer. For instance $\epsilon = 1/2$.
- **Mode 5.** $a = \left\lceil n^{\lceil \frac{1}{\log \log n} \rceil} \right\rceil$ and $h = \lceil \log \log n \rceil$.

⁶ The number of calls to the permutation can be slightly decreased by replacing the first chaining value of a node by the payload of the f -input (node) by which this chaining value has been produced. The payload of a resulting node corresponds to a message block followed by one or more chaining values. By using sakura, it is augmented with adequate frame bits. The message block is then encoded in a message hop and, if kangaroo hopping is used once or several times, the chaining values are encoded in one or several chaining hops.

⁷ If kangaroo hopping is used to transform all the nodes, starting from the bottom level up to the final node, a more accurate asymptotic parallel running time is $u_1 + \sum_{i=2}^h (u_i - 1)$.

- **Mode 6S.** $a = \left\lceil n^{\frac{1}{\lceil \sqrt{\log n} \rceil}} \right\rceil$ and $h = \lceil \sqrt{\log n} \rceil$.
- **Mode 7S.** $a = \left\lceil \frac{\log n}{W(\log n)} \right\rceil$ and $h = \left\lceil \frac{\log n}{W(\log n)} \right\rceil$, where $W(\cdot)$ is the first branch of the Lambert function.
- **Mode 8S.** $a = \lceil \log \log n \rceil$ and $h = \left\lceil \frac{\log n}{\log \log \log n} \right\rceil$.
- **Mode 9.** $a = \left\lceil \frac{W_{-1}(-\ln^{-\epsilon} n)}{-\ln^{-\epsilon} n} \right\rceil$ and $h = \left\lceil \frac{-\ln n}{W_{-1}(-\ln^{-\epsilon} n)} \right\rceil$, where $W_{-1}(\cdot)$ is the second branch of the Lambert function.

Let us consider that we have a secured application making use of hashing. Choosing a hashing mode suitable for this application involves determining what are the most memory-constrained devices using it and what their proportion is. If this application is very likely to be used by a *Class 1* device, then Mode 1, 2 or 4 should be used. If *Class 0* is absent, but *Class 1* is sufficiently represented, then Mode 5 or 6S could be a good choice. If neither *Class 0* nor *Class 1* is present, but *Class 2* is, then Mode 7S or 8S is an interesting choice. If none of these classes are present, but *high-end IoT devices* are, then we propose to use Mode 9. Finally, if none of these devices are present, the memory consumption is not really a problem and then Mode 3 is the best choice. An example of tree resulting from the application of Mode 4 on a small message is illustrated in Figure 3.

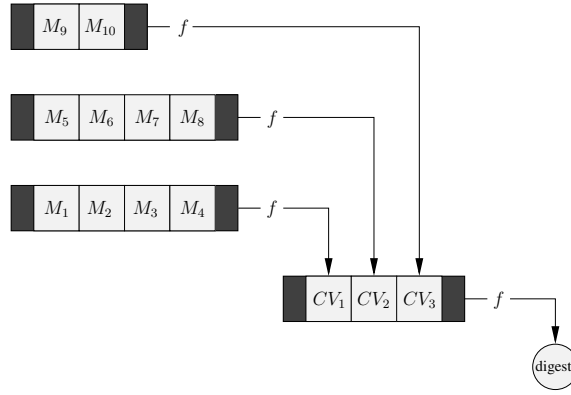


Fig. 3. Tree of nodes used by f to compress a message of 10 blocks, assuming the use of Mode 4 with the parameter $\epsilon = 1/2$. Levels 1 and 2 are represented from left-to-right and nodes at a same level are represented vertically from bottom-to-top. In these nodes, black blocks contain frame bits and light grey blocks message blocks or chaining values.

Theorem 1. *There are 6 tree hashing modes having the following efficiency complexities:*

- Mode 4 has an ideal running time in $O(n^\epsilon)$, for a sequential memory consumption of $O(1)$ hash states. There exists a variant conserving these same complexities, but requiring only $O(n^{1-\epsilon})$ processors.
- Mode 5 has an ideal running time in $O(n^{\frac{1}{\log \log n}} \log \log n)$, for a sequential memory consumption of $O(\log \log n)$ hash states. There exists a variant conserving these same complexities, but requiring only $O\left(\frac{1}{\log \log n} n^{1-\frac{1}{\log \log n}}\right)$ processors.
- Mode 6S has an ideal running time in $O(n^{\frac{1}{\sqrt{\log n}}} \sqrt{\log n})$, for a sequential memory consumption of $O(\sqrt{\log n})$ hash states. There exists a variant conserving these same complexities, but requiring only $O\left(\frac{1}{\sqrt{\log n}} n^{1-\frac{1}{\sqrt{\log n}}}\right)$ processors.
- Mode 7S has an ideal running time in $O\left(\frac{\log^2 n}{\log^2 \log n}\right)$, for a sequential memory consumption of $O\left(\frac{\log n}{\log \log n}\right)$ hash states. There exists a variant conserving these same complexities, but requiring only $O\left(\frac{n \log^2 \log n}{\log^2 n}\right)$ processors.
- Mode 8S has an ideal running time in $O\left(\frac{\log n \log \log n}{\log \log \log n}\right)$, for a sequential memory consumption of $O\left(\frac{\log n}{\log \log \log n}\right)$ hash states. There exists a variant conserving these same complexities, but requiring only $O\left(\frac{n \log \log \log n}{\log n \log \log n}\right)$ processors.
- Mode 9 has an ideal running time in $O(\log^{1+\epsilon} n)$, for a sequential memory consumption of $O\left(\frac{\log n}{\log \log n}\right)$ hash states. There exists a variant conserving these same complexities, but requiring only $O\left(\frac{n}{\log^{1+\epsilon} n}\right)$ processors.

Proof. First, we examine the modes, in terms of parallel running time and memory consumption, one after the other:

- Mode 4 is consistent since $\lceil n^\epsilon \rceil^{1/\epsilon} \geq n$. Its asymptotic parallel running time is clearly $O(n^\epsilon/\epsilon)$ and the height of the tree is $O(1/\epsilon)$, where ϵ is a constant.
- For Mode 5 and Mode 6S, we would like a tree of height $h(n)$ where $h(n)$ is $\lceil \log \log n \rceil$ or $\lceil \sqrt{\log n} \rceil$. We seek the smallest a such that $a^{h(n)} \geq n$, *i.e.* $\log a \geq \log n/h(n)$. We find $a = \lceil n^{1/h(n)} \rceil$, and the product height \times arity gives the expected asymptotic parallel running time.
- For Mode 7s, the approach is different. We seek the smallest a such that $a^a \geq n$. We first note that the Lambert function, denoted W , solves the equations of type $xe^x = y$, where the solution for the unknown variable x is $W(y)$. This function can be used to solve the equation $a^a = n$. We get $a = \frac{\log n}{W(\log n)}$ which is increasing for $n \geq 1$. Thus, the smallest a solution to the inequation $a^a \geq n$ is $\lceil \log n/W(\log n) \rceil$. Note that $W(a) = \Theta(\log a - \log \log a)$, meaning that a is $O\left(\frac{\log n}{\log \log n}\right)$. Consequently, the tree height is $O\left(\frac{\log n}{\log \log n}\right)$ and the ideal parallel running time is $O\left(\frac{\log^2 n}{\log^2 \log n}\right)$.
- For Mode 8S, we seek the smallest a such that $a^{\frac{\log n}{\log \log \log n}} \geq n$, which gives $a = \lceil \log \log n \rceil$. The product height \times arity gives the expected parallel running

time. We thus notice a slight decrease in the memory consumption for a near-optimal parallel running time.

- With Mode 9, the parallel running time is obtained by construction. Indeed, we seek the smallest h such that $hn^{1/h} = \log^{1+\epsilon} n$. We have:

$$n^{1/h} = \frac{1}{h} \log^{1+\epsilon} n \Leftrightarrow \frac{-1}{\log^\epsilon n} = \left(\frac{-1}{h} \log n \right) e^{\frac{-1}{h} \log n},$$

and it follows that $\frac{-1}{h} \log n = W\left(\frac{-1}{\log^\epsilon n}\right)$. Thus, $h = \frac{-\log n}{W(-\log^{-\epsilon} n)}$. Since $-\log^{-\epsilon} n$ is negative and tends to 0, we can evaluate h thanks to the two branches $W_{-1}(\cdot)$ or $W(\cdot)$ of the Lambert function. We choose to use $W_{-1}(\cdot)$ since we have the asymptotic approximation

$$W_{-1}(x) = \Omega(\log(-x) - \log(-\log(-x)))$$

when $x \in [-1/e, 0]$. Consequently, h is $O\left(\frac{\log n}{\epsilon \log \log n}\right)$.

Hence, there are variants of all these modes reducing the number of processors required to obtain the aforementioned parallel running times. Let $T(n)$ be the ideal parallel running time of a mode having $a = \alpha(n)$ as parameter. We consider the following tree topology: the first level has approximately $n/T(n)$ nodes, each containing $T(n)$ message blocks, except maybe the rightmost one whose number of message blocks can be smaller. A tree of arity a is constructed on top of these $n/T(n)$ nodes. Thus, the first level is of arity $T(n)$ while the others are of arity a . The nodes are then formatted using Sakura coding in accordance with this tree representation. The overall parallel running time to compute the digest is approximately $T(n) + T(n/T(n))$, by using only $n/T(n)$ processors.

5 Parameters for streaming live content

In this section, we discuss the parallel operating modes which do not require the message size as input. These modes are scalable and can process the message as it is received. They are essential for any application making use of live streaming. In order to reduce the memory consumption, we propose to use an increasing sequence $(u_i)_{i \geq 1}$ of arities. First, we note that for this kind of tree it is impossible to adapt Mode 4 to support streaming live content while keeping the same complexities (a digression will be made about other kind of trees at the end of the section). Suppose that the tree height h is constant. It is impossible to have a finite sequence $(u_i)_{i=1 \dots h}$ such that $\prod_{i=1}^h u_i \geq n$ for all possible message size n , and where the terms u_i do not depend on n . Thus, we have to make do with Mode 1 or Mode 2. However, in what follows, we show that certain sequences of arities give interesting asymptotic efficiencies.

Mode 3w (weak compromise). We fix the arity of the tree to a constant k which is a power of 2. For instance, if $k = 2^4$, the number of hash states of a sequential implementation is approximately divided by 4, with the counterpart that the

parallel running time is approximately multiplied by 4. Generally, if $k = 2^i$ where i is a positive integer, then the memory consumption of a sequential implementation is approximately divided by i while the parallel running time is multiplied by $k/\log_2 k$. There is no major change asymptotically.

Mode 6L. The arities u_i of the levels $i \geq 1$ grow as follows:

$$u_i = 2^i \quad \forall i \geq 1.$$

Theorem 2. *Suppose that we have as many processors as we want. Mode 6L has $O(n^{\frac{1}{\sqrt{\log n}}}\sqrt{\log n})$ parallel running time and requires $O(\sqrt{\log n})$ hash states in memory to process the message with a single processor.*

Proof. Asymptotically, the amount of memory consumed in a sequential implementation corresponds, in the worst case, to the height k of the tree. We seek the lowest k such that $\prod_{i=1}^k u_i \geq n$. We have $\prod_{i=1}^k 2^i \geq n$ which is equivalent to $\sum_{i=1}^k i \geq \frac{\log n}{\log 2}$. Then, $k^2 \sim \frac{\log n}{\log 2}$ and $\sum_{i=1}^k 2^i \sim n^{\frac{1}{\sqrt{\log_2 n}}}\sqrt{\log_2 n}$.

Mode 7L. The arities u_i of the levels $i \geq 1$ grow as follows:

$$u_i = i + 1 \quad \forall i \geq 1.$$

Theorem 3. *Suppose that we have as many processors as we want. Mode 7L has $O\left(\frac{\log^2 n}{\log^2 \log n}\right)$ parallel running time and requires $O\left(\frac{\log n}{\log \log n}\right)$ hash states in memory to process the message with a single processor.*

Proof. Asymptotically, the amount of memory consumed in a sequential implementation corresponds, in the worst case, to the height k of the tree. We seek the lowest k such that $\prod_{i=1}^k u_i \geq n$. According to the Stirling Formula, we have $k! \sim k^k e^{-k} \sqrt{2\pi k}$. Thus, we have $\log n \sim \log(k^k e^{-k} \sqrt{2\pi k}) = k \log k - k + \frac{1}{2} \log(2\pi k) \sim k \log k$. Hence, $\log \log n \sim \log(k \log k) = \log k + \log \log k \sim \log k$. Then, we have $k \sim \frac{\log n}{\log k} \sim \frac{\log n}{\log \log n}$. If we have as many processors as we want, then the amount of blocks processed sequentially is asymptotically the sum of the k level arities. Considering that $\sum_{i=2}^k i \sim k^2$, we deduce the expected result.

Mode 8L. The arities u_i of the levels $i \geq 1$ grow as follows:

$$u_i = \lfloor \log(i + 3) \rfloor \quad \forall i \geq 1.$$

Theorem 4. *Suppose that we have as many processors as we want. Mode 8L has $O\left(\frac{\log n \log \log n}{\log \log \log n}\right)$ parallel running time and requires $O\left(\frac{\log n}{\log \log \log n}\right)$ hash states in memory to process the message with a single processor.*

Proof. For the sake of simplification, we can seek the lowest k such that $\prod_{i=2}^k \log i \geq n$, which is an equivalent problem. Asymptotically, we can then

seek k such that $k \log \log k \sim \sum_{i=2}^k \log \log i \sim \log n$. The solution k is a fixed point of the function $k \mapsto \frac{\log n}{\log \log k}$, meaning that k is close to $\frac{\log n}{\log \log \log n}$ when n tends to infinity. Regarding the parallel running time, it follows that $\sum_{i=2}^k \log i \sim k \log k \sim \frac{\log n}{\log \log \log n} \log \left(\frac{\log n}{\log \log \log n} \right)$, giving the expected result.

Note that this paper does not investigate sequences behaving like superfactorial or hyperfactorial, which probably lead to other interesting trade-offs.

As for the nodes dedicated to stored content, we can reduce the number of processors allowing the ideal parallel running time to be reached by means of a rescheduling technique. Indeed, since the processors have more computations to do at higher levels of the tree (and the lower level nodes have smaller arity), we could assign to each processor a subtree that represents the amount of computations of a higher level node. In doing so, the parallel running time is only increased by a constant factor. This aspect will be developed in future works.

Until now, we have discussed the use of trees having increasing level arities. Can we benefit from increasing arities at a same level of the tree? We respond to this question with a sketch of tree mode, denoted Mode 4L, which minimizes the tree height and allows the processing of live-streamed content. We suppose that we have $k + 1$ processors denoted $P_0, P_1, P_2, \dots, P_k$. Processors P_1, \dots, P_k each compute the hash of a distinct message part⁸. These chaining values are denoted cv_1, \dots, cv_k . For the sake of simplicity, we suppose that *kangaroo hopping* is not used. Processor P_0 then merely collects and processes these chaining values as soon as they are evaluated. We suppose that the processors start their computations at the same time (P_0 is delayed since *kangaroo hopping* is not used). It appears that during the processing of a chaining value cv_i by P_0 , Processor P_{i+1} can process one or more message blocks. Thus, the message parts processed could be of increasing lengths without producing a significant impact on the running time of P_0 . Let us suppose the following complexities:

- the running time of P_i is $ai + b$ where a and b are positive constants, for $i \geq 1$.
- the running time of P_0 is related to the number of chaining values. For instance, it is $a'k + b'$, where a' and b' are positive constants. The waiting time for cv_1 is supposed to be taken into account.

We have $\max\{a'k + b', \max_{1 \leq i \leq k}\{ai + b\}\} = O(k)$, where k is the smallest integer such that $\sum_{i=1}^k i \geq n$. The total parallel running time is then in $O(n^{1/2})$.

6 Conciliating interleaving and scalability

The interleaving of the message bits/blocks is an interesting feature if it is applied to nodes having a low number of children. For instance, in Mode 2L, it is applied

⁸ A message part consists of a certain number of blocks.

to a single node having q children, and we can think that q is small enough so that the memory space consumption is small as well. Even though Sakura allows the coding of an interleaving parameter for each node, we can wonder whether it is judicious to use it for all nodes in the tree. Let us imagine that our hash function is based on a tree structure with a logarithmic height and a constant arity (*e.g.* all nodes are of arity 4). In fact, with the assumption that message bits/blocks are processed in sequence, using interleaving for all nodes leads to a memory consumption in the order of the size of the message. It is then preferable to use it for the nodes of a single level (or of a constant number of levels) in the tree.

Interleaving is bounded to a processor architecture, that is, it is difficult to imagine an interleaving mechanism that could be, in some sense, scaled on different architectures (with different register sizes). Having this in mind, we can still imagine a mode that could be both bounded to a single architecture and scalable from the multithreading standpoint. In order to use interleaving in the tree structures having nodes of high arity (possibly dependent of the message size), we propose to add a single level of small and constant arity nodes. The interleaving parameter would be set⁹ for all nodes of this level (and only this one). The location where this level is inserted in the tree depends on the type of application (stored or live-streamed content). In the case of stored content, this level would be inserted near the root. In the case of live-streamed content, this level would probably be inserted near the base level in order to allow both the use of multithreading and SIMD instructions. In fact, in this second case, the most adequate location of the insertion depends on the bandwidth with which the message is received. It should be inserted at the highest level¹⁰ which allows the correct feeding of SIMD instructions executed in parallel by the processors. Note that, in either case, the asymptotic complexity remains unchanged by this modification.

7 Concluding remarks

In this paper, we introduced different ways of constructing a SHAKE function based on an inner function RawSHAKE and a parameterizable tree of nodes. Resulting modes are Sakura-compatible and lead to asymptotical bounds with regard to the memory usage in a sequential implementation and the ideal parallel running time (with as many processors as wanted). We showed that interesting trade-offs are possible in terms of space-time complexity, including for streaming live content. However, constant factors hidden inside the big-O notation may have a significant impact when working at a finite distance. Thus, we think that it would be of interest to undergo an in-depth analysis on the effectiveness of these modes at finite distance, while using various optimizations dedicated to software implementations. This could also lead to introduce new modes.

⁹ A value different from $\{I_\infty\}$ is used.

¹⁰ Considering a constant bandwidth and an infinitely large message, this level is clearly near the base level.

Besides, we supposed the result of the SHAKE function to be of fixed size (512 bits), while its main functionality is to generate a hash of arbitrary length. If we nevertheless decide to allow this functionality by using the "squeezing" phase of the last evaluation of RawSHAKE (on the final node), our asymptotic results lose their relevance. Indeed, the "squeezing" phase being purely sequential, if it is decided to generate a hash of, say, n bits from a message of length n , the total number of evaluations of the permutation is then in $O(n)$, no matter the type of tree used to compress the message. To remedy this problem, that one "squeezing" phase can be demultiplied using an inverted tree of type GGM (Goldreich-Goldwasser-Micali [16]) having for root the final node. In this configuration, there are two trees: one to compress the message and the other one to expand the hash to the desired length. Nodes of the second tree are defined such that each node is a part of an f -image except for the root node which is an f -input. Originally, GGM construction allows a long pseudo-random bit-string to be generated from a short (pseudo-)random seed, by means of a binary tree. In our case, the used tree is not necessarily binary. Rather, its topology coherently depends on the parameters used in the first tree, *i.e.* with possibly different level arities. More precisely, the arity of a node determines the length of the corresponding hash generated by RawSHAKE. Indeed, it seems that these parameters can be reused in this second inverted¹¹ tree so as to maintain the asymptotic running time. A GGM tree normally has all its leaves at the same depth. Its height is then chosen such that the total number of bits of the leaves is greater than or equal to the desired hash length. In order to generate a hash of a given size, some of its branches should be pruned and one leaf should probably be truncated. We finally note that interleaving techniques could be used in this second tree to take better advantage of SIMD instructions. We think that these last aspects deserve particular attention. If we do not want to use a GGM tree construction to expand the hash, an alternative is a Counter based PRG, *e.g.* CTR-DRBG (Counter mode Deterministic Random Byte Generator [3]) whose ideal parallel running time is constant.

At the time of writing these lines, a NIST draft report [26] on SHA-3 derived functions is available. We can find in this report a concrete variant of Mode 1. At the same time, Bertoni *et al.* [11] propose another variant implemented with numerous optimizations, in particular the use of SIMD instructions, kangaroo hopping and a Round-Reduced version of Keccak, to name just a few.

Our paper has focused on the asymptotic analysis of tree hash modes but there is also an interest for their optimizations at finite distance in order to produce efficient tree hash circuits. For instance, in the case of Mode 3, an interesting work is to reduce at best the number of rounds¹² of parallel calls to the permutation (*i.e.* the depth of the circuit, which is in $O(\log n)$) by taking into account the frame bits overheads of Sakura. A much more interesting challenge is to optimize the mode in depth and width: the goal is to minimize the maximum

¹¹ It is inverted in the sense that the flows of computation are from root to leaves.

¹² The running time of a round is one unit of time, that is, one call to the permutation.

number of parallel calls to the permutation that can occur in one round, subject to the constraint of a minimal number of rounds.

As a last remark, there were no modes referenced in the tables (Section 1) having an asymptotically sublogarithmic (or even constant) parallel running time. The reason is that using only f as operator, such complexities are not possible. In order to make a parallel running time in $O(1)$ possible, the hashing mode should require a constant number of rounds of parallel calls to the underlying permutation, and the intermediate results of the last round should be combined with a second operator of negligible cost (*e.g.* the XOR operator). The *Randomize-then-Combine* paradigm has been proposed by Bellare *et al.* [4], with several examples of combining operators. The XOR operator must be avoided for arbitrary-length messages, due to an attack based on Gauss elimination. More generally, some of the proposed operators in finite groups are not resistant to the subexponential *Generalized Birthday Attack* [30]. The possibility of such a scheme nevertheless looks interesting. As we have seen previously, the use of a traditional tree hash mode (using only f) by a set of computers having different computational resources leads to a race to the bottom with respect to the potential parallelism. Indeed, when transmitting a signed message, the parameters of the tree topology chosen by the transmitting computer are not necessarily optimal for the receiving one. A hashing scheme with a $O(1)$ ideal parallel running time resolves this problem. Besides, if the combining operator is both associative and commutative, the distribution of work among the computing units can be done in any manner, without any tree topology constraint.

References

1. Elena Andreeva, Bart Mennink, and Bart Preneel. Security Reductions of the Second Round SHA-3 Candidates. Cryptology ePrint Archive, Report 2010/381, 2010.
2. Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and Christian Winnerlein. BLAKE2: Simpler, Smaller, Fast As MD5. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security, ACNS’13*, pages 119–135, Berlin, Heidelberg, 2013. Springer-Verlag.
3. Elaine B. Barker and John M. Kelsey. SP 800-90A. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, Gaithersburg, MD, United States, 2012.
4. Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques, EUROCRYPT’97*, pages 163–192, Berlin, Heidelberg, 1997. Springer-Verlag.
5. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. In *Ecrypt Hash Workshop*, May 2007.
6. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sufficient conditions for sound tree and sequential hashing modes. Cryptology ePrint Archive, Report 2009/210, 2009.
7. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, January 2011.

8. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 313–314, 2013.
9. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sakura: A Flexible Coding for Tree Hashing. In *Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings*, pages 217–234, 2014.
10. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sufficient Conditions for Sound Tree and Sequential Hashing Modes. *Int. J. Inf. Secur.*, 13(4):335–353, August 2014.
11. Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. KangarooTwelve: fast hashing based on Keccak-p. *Cryptology ePrint Archive*, Report 2016/770, 2016.
12. Carsten Bormann, Mehmet Ersue, and Ari Keränen. Terminology for Constrained-Node Networks. RFC 7228, October 2015.
13. Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle Tree Traversal Revisited. In *Proceedings of the 2Nd International Workshop on Post-Quantum Cryptography*, PQCrypto '08, pages 63–78, Berlin, Heidelberg, 2008. Springer-Verlag.
14. Ivan Damgård. A Design Principle for Hash Functions. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 416–427, London, UK, 1990. Springer-Verlag.
15. Niels Ferguson, Stefan Lucks Bauhaus, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family (version 1.2), 2009.
16. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to Construct Random Functions (Extended Abstract). In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 464–479, 1984.
17. Shay Gueron. Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256. *IACR Cryptology ePrint Archive*, 2014:170, 2014.
18. Shay Gueron and Vlad Krasnov. Parallelizing message schedules to accelerate the computations of hash functions. *J. Cryptographic Engineering*, 2(4):241–253, 2012.
19. Shay Gueron and Vlad Krasnov. Simultaneous Hashing of Multiple Messages. *J. Information Security*, 3(4):319–325, 2012.
20. O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes. Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal*, PP(99):1–1, 2015.
21. John Kelsey. What Should Be In A Parallel Hashing Standard? NIST, SHA3 Workshop, 2014.
22. RSA Laboratories. PKCS # 1 v2.2 RSA Cryptography Standard, 2012.
23. Stefan Lucks. Tree hashing: A simple generic tree hashing mode designed for SHA-2 and SHA-3, applicable to other hash functions. In *Early Symmetric Crypto (ESC)*, 2013.
24. Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford, CA, USA, 1979.
25. National Institute of Standards and Technology. FIPS PUB 202: Secure Hash Standard (SHS). Technical report, aug 2015.
26. NIST special publication 800-185. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and Parallel-Hash (draft). Technical report, aug 2016.

27. Andrew Regenscheid, Ray Perlner, Shu jen Chang, John Kelsey, Mridul Nandi, and Souradyuti Paul Nistir. The SHA-3 Cryptographic Hash Algorithm Competition, 2009.
28. Ronald L. Rivest, Benjamin Agre, Daniel V. Bailey, Christopher Crutchfield, Yevgeniy Dodis, Kermin Elliott, Fleming Asif Khan, Jayant Krishnamurthy, Yuncheng Lin, Leo Reyzin, Emily Shen, Jim Sukha, Drew Sutherland, Eran Tromer, and Yiqun Lisa Yin. The MD6 hash function: A proposal to NIST for SHA-3, 2008.
29. Michael Szydlo. Merkle Tree Traversal in Log Space and Time. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 541–554, 2004.
30. David Wagner. A generalized birthday problem. In *Proceedings of the 22Nd Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '02*, pages 288–303, London, UK, UK, 2002. Springer-Verlag.