Sakura: a flexible coding for tree hashing

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

[1] STMicroelectronics
[2] NXP Semiconductors

Abstract. We propose a flexible, fairly general, coding for tree hash modes. The coding does not define a tree hash mode, but instead specifies a way to format the message blocks and chaining values into inputs to the underlying function for any topology, including sequential hashing. The main benefit is to avoid input clashes between different tree growing strategies, even before the hashing modes are defined, and to make the SHA-3 standard tree-hashing ready.
Keywords: hash function, tree hashing, indifferentiability, SHA-3

1 Introduction

A *hashing mode* can be seen as a recipe for computing digests over messages by means of a number of calls to an underlying function. This underlying function may be a fixed-input-length compression function, a permutation or even a hash function in its own right. We use the term *inner function* and symbol f for the underlying function and the term *outer hash function* and symbol F for the function obtained by applying the hashing mode to the inner function.

The hashing mode splits the message into substrings that are assembled into inputs for the inner function, possibly combined with one or more *chaining values* and so-called *frame bits*. Such an input to f is called a *node* [6]. The chaining values are the results of calls to f for other nodes.

Hashing modes serve two main purposes. The first is to build a variable-input-length hash function from a fixed-input-length inner function and the second is to build a tree hash function. In tree hashing, several parts of the message may be processed simultaneously and parallel architectures can be used more efficiently when hashing a single message than in sequential hashing [16,8,22,3,9,6].

1.1 Motivation and prior art

The motivation for standardizing a tree hash mode, or to have a tree-hash-ready SHA-3 standard, was discussed at various occasions during the SHA-3 competition on the NIST hash-forum mailing list [17]. A few candidates, like MD6, SANDstorm and Skein, proposed built-in tree hash modes [21,23,10]. At the Third SHA-3 Candidate Conference, Lucks, McGrew and Whiting motivated why the SHA-3 standard should support parallelized tree hashing [14].

Different applications or use cases call for different approaches to tree hashing and different tree topologies. For instance, some environments favor cutting the input message in consecutive pieces and hashing these pieces independently, while others favor to hash interleaved pieces of data, see, e.g., [11]. In his presentation at ESC 2013, Lucks suggested to use a n-ary tree with much potential parallelism and to let the implementation choose the most appropriate evaluation strategy [13]. As another example, some applications require to keep the intermediate hash values (e.g., to be able to re-compute the

digest if only a part of the input changes), whereas the mere exploitation of parallelism does not require it.

Given all this diversity, it seems difficult to agree on a "one-size-fits-all" tree hash mode. Instead, we take the different approach of allowing different tree hash modes to co-exist. However, the co-existence of different modes on top of existing (serial) hash functions calls for caution. While each individual hash mode can be proven secure, the joint use of several modes can become insecure, in particular due to the different coding conventions that could collide into equal inputs to the inner function. This paper proposes a way to bring together different tree hash modes in a secure way and follows ideas presented in [5, Slides 54-59].

1.2 Our contribution

We show that it is possible to define a tree hash coding, i.e., a way to format the input to the inner function, that can cover a wide range of tree hash modes. For a carefully designed tree hash coding, one can prove that the union of all tree hash modes compatible with it is *sound*. By sound we mean that it does not introduce any weaknesses on top of the risk of collisions in the inner function. More precisely, a hashing mode is sound if the advantage of differentiating F from a random oracle, assuming f has been randomly selected, is upper bound by $q^2/2^{n+1}$, with q the number of queries to f and n the length of the chaining values [1,15,7,6].

As a result, tree hash modes compatible with the defined coding can be progressively introduced while preserving their joint security. Also, as an additional benefit, a tree hash mode following the coding convention is sound by construction, without the need of additional proofs.

For proving soundness, we use the results of [6], in which we specify a set of conditions for a tree (or sequential) hashing mode to be sound. We assume that to the choice of f is attached a security parameter, like the capacity in the specific case of sponge functions or the security strength [18,2]. We consider this security parameter to be specified together with f and to remain constant for its entire use in a tree hash mode.

The remainder of this paper is structured as follows. In Section 2 we explain the range of possibilities of our proposed sound tree hash coding and illustrate it with some examples. In Section 3 we specify Sakura, the coding we propose, while in Section 4 we define what it means for a hashing mode to be compatible with Sakura and prove that any such tree hash mode is sound. In Section 5 we give some examples of modes and in Section 6 we discuss the use of Sakura in the context of making the SHA-3 standard tree-hashing ready.

2 Functionality supported by Sakura

We start by recalling the very general concept of node and tree of nodes. We then capture the functionality of Sakura with trees of hops and how nodes and hops relate to one another. Finally, some figures illustrate the concepts.

2.1 Modeling tree hash modes

We refer to [6, Section 2] for a detailed description of the model. We here give a short summary.

A tree is a directed graph of *nodes*. Informally speaking, each node is hashed with the inner function f and the output is given to its parent node as a chaining value. The exception is for the final node (i.e., the root of the tree), which does not have a parent, and the output of the outer hash function $F(M)$ is the output of f applied to this final node.

A tree hash mode $\mathcal{T}$ specifies a tree of nodes as a function of the input message length $|M|$ and some specific parameters A. In particular, it is up to the mode to define how the tree scales as a function of $|M|$, how the message bits are spread on the nodes, which nodes takes chaining values from which nodes, etc.

For a fixed $|M|$ and A, a tree hashing mode specifies precisely how to format the inputs to the inner function f with bits from the message, chaining values and frame bits. The latter are constant bits for padding or domain separation. The union of tree hash modes is defined in [6, Section 7.3]. The union $\mathcal{T}_{\text{union}}$ of k tree hashing modes $\mathcal{T}_i$ simply means that the user has a choice parameter indicating the chosen mode i composed with the tree parameters A_i for the particular mode i. With $\mathcal{T}_{\text{union}}$, the user can thus reach any node tree that some $\mathcal{T}_i$ can produce.

2.2 From generality to functionality

The model of the tree using nodes is very general and allows modeling even the most cumbersome tree hash mode, e.g., where a node inputs 2 chaining value bits from child #4 then 7 message bits, etc. We now introduce some concepts that restrict this general model to one that can be easily represented and yet is sufficiently flexible to cover all practical cases we can think of.

We represent trees in terms of *hops* that model how message and chaining values are distributed over nodes. Any tree of hops uniquely maps to a tree of nodes, so they are still supported by the model mentioned above. However, not all trees of nodes (such as the cumbersome example above) can be represented in trees of hops.

In Sakura, any tree of hops is encoded into a tree of nodes. In other words, the functionality supported by Sakura is exactly that of all possible trees of hops that can be built. Sakura-compatible tree hash modes are not required to generate all possible hop trees, but instead they can focus on the desired subset of them. In the sequel, we define what the hops are and how they are encoded into nodes.

2.3 Hops and hop trees

Unlike a node that may simultaneously contain message bits and chaining values, there are two distinct types of hops: *message hops* that contain only message bits and *chaining hops* that contain only chaining values.

The hops form a tree, with the root of the tree called the *final hop*. Such a hop tree determines the parallelism that can be exploited by processing multiple message hops or chaining hops in parallel.

Each hop has a single outgoing edge. A message hop has no incoming edges. The number of incoming edges of a chaining hop is called its *degree* d. The hops at the other end of these edges are called the *child hops* of that chaining hop. The edges to a hop are labeled with numbers 0 to $d-1$ and the hop at the end of edge 0 is called the *first child hop*. There is exactly one hop that has no outgoing edge and we call it the *final hop*. There is exactly one path from each hop to the final hop.

We define the position of a hop in a hop tree by an index, that specifies the path to follow to reach this hop starting from the final hop. It consists of a sequence of integers $\alpha = \alpha_0\alpha_1 \ldots \alpha_{n-1}$. Indexing is defined in a recursive way:

- The index of the final hop is the empty sequence, denoted $*$.
- The index of the i-th child of a hop with index α has index $\alpha||i-1$.

The length of this sequence specifies the distance of the specified hop to the final hop and is called its *height*. The height of the hop tree is the maximum height over all hops.

2.4 Interleaving the input over message hops

In general, message bits are distributed onto message hops from the first to the last child.

In streaming applications, one may wish to divide message substrings over multiple hops as the message becomes available. For this purpose chaining hops have an attribute called *interleaving block size* I that determines how this shall be done. The principle is that a chaining hop distributes the message bits it receives over its child hops. It hands the first I bits to its first child, the second sequence of I bits to its second child and so on. After reaching the last of its child hops, it returns to its first child and so on. When a receiving hop is also a chaining hop, it will distribute the message bits over its child hops according to its own interleaving block size. When this process ends is determined by the hashing mode. For example, it can be when the end of the message is reached or when the hops have reached some maximum size specified in the mode's parameters.

A mode that does not make use of message block interleaving can set the interleaving block size of the chaining hops to a value that is larger than any message that may be presented, and we say $I = \infty$.

The way message bits are distributed is formally captured by the `GetMessage` function in Definition 1 below. For examples with block interleaving, please see Sections 5.2 and 5.3.

2.5 Mapping hops to nodes

One can define hashing modes where the concepts of node and hop coincide by imposing that each node contains exactly one hop. With *kangaroo hopping* defined below, however, the first child hop is coded before its parent in the same node.

In a mode without kangaroo hopping, the node tree is constructed from the hop tree using the same topology. A node contains exactly one hop. The nodes are constructed by putting message bits in nodes containing a message hop and by putting chaining values in nodes containing a chaining hop.

The motivation for kangaroo hopping is the following. The length of (a node mapped from) a chaining hop is the number of children multiplied by the length of the chaining value. Compared to sequential hashing, this corresponds to an overhead. Also, there is typically some additional computational overhead per call to f. Kangaroo hopping reduces this overhead by putting multiple hops per node in a way that does not jeopardize the potential parallelism expressed in the hop tree. A chaining hop has an attribute that says whether kangaroo hopping must be applied on it, and if so, the chaining hop is also called a *kangaroo hop*. When encoding a kangaroo hop into a node, the node contains its first child hop itself instead the chaining value (its f-image). For the other child hops it contains the chaining values as usual. Hence, when evaluating $F(M)$, instances of f can

process child hops in parallel and then the instance of f for the first child continues processing the parent hop.

Kangaroo hopping can be applied in a recursive way, i.e., the first child hop may also be a kangaroo hop. All in all, a node may contain a message hop followed by zero, one or more chaining hops, or one or more chaining hops. Kangaroo hopping reduces the number of nodes to the total number of hops minus the number of kangaroo hops. It is easy to see that the number of nodes can be reduced to the number of message hops, but not to less.

The result of applying f to the final node is the output of F. The last hop in this node is the final hop. The result of applying f to an inner node is a chaining value.

2.6 Illustrations

We illustrate these concepts with some examples in Figures 1, 2 and 3. These figures depict hop trees with the following conventions. Message hops have sharp corners, chaining hops have rounded corners. The final hop has a grey fill, the others a white fill. An edge between child and parent has an arrow and enters the parent from above if the chaining value obtained by applying f to the child hop is in the parent hop. It has a short dash and enters the parent hop from the left in the case of kangaroo hopping. Hops on the same horizontal line are in the same node.

In Figure 1 there are in total 5 hops: 4 message hops M_0 to M_3 and one chaining hop Z_*. The final node contains both the final hop Z_* and M_0 because of kangaroo hopping. The total number of nodes is 4.

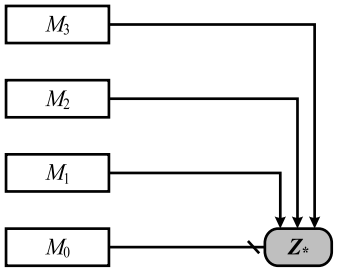

Fig. 1. Example of a hop tree with application of kangaroo hopping. M_0 and Z are in the same node.

In Figure 2 there are in total 7 hops: 4 message hops M_{00}, M_{01}, M_{10}, M_{11}, and three chaining hops Z_0, Z_1 and Z_*. The final node contains only the final hop Z_*. The hops M_{00} and Z_0 are in a single node. Similarly, M_{10} and Z_1 are in a single node. The total number of nodes is 5.

In Figure 3 there is only a single hop, that is at the same time a message hop and the final hop. Clearly, there is only a single node containing this hop.

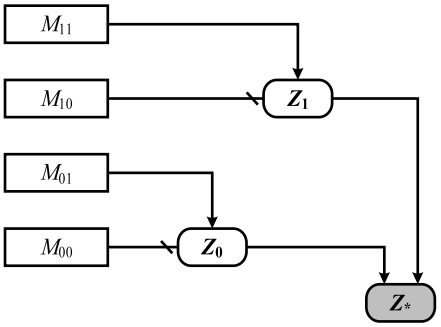

Fig. 2. Another example of a hop tree. M_{00} and Z_0 are in the same node, as well as M_{10} and Z_1.

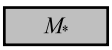

Fig. 3. Example of a hop tree with a single node

3 The Sakura tree coding

In this section we specify the Sakura tree coding. The goal of this coding is to allow a tree hash mode to encode a hop tree into the input of f. From this definition, it should be clear how the evaluation of $F(M)$ must be processed.

For a Sakura-compatible tree hash mode to be sound, the individual parts (e.g., message bits, chaining values) must be unambiguously recovered by parsing the node tree. Of course, such a decoding never occurs in practice but must be ensured for satisfying tree-decodability. The coding adds frame bits for tree-decodability, as well as to ensure domain separation between inner nodes and the final node.

The coding is based on a number of simple principles:

- Nodes, namely inputs to f, can be unambiguously decoded into hops *from the end*. This is done by
  - coding in a trailing frame bit whether it is a chaining hop or a message hop;
  - allowing at most a single message hop per node, and this at the beginning;
  - allowing the parsing of a chaining hop from the end.
- The parsing of a chaining hop from the end is made possible in the following way:
  - it is a series of chaining values followed by an interleaving block size;
  - an interleaving block size consists of 2 bytes;
  - at the end of the chaining values their number is appended in suffix-free coding;
  - the length of the chaining values is determined by the security strength of f.
- We apply simple padding between the hops in a node, so as to allow the alignment of these elements to byte boundaries, 64-bit word boundaries or to any other desired boundaries. (This is up to the mode to define.)
- We apply simple padding at the end of inner nodes. Where appropriate, this can be used by a mode to ensure that different sibling inner nodes have the same length. This may simplify the implementation, e.g., if sibling inner nodes are processed in parallel using SIMD instruction. (Again, this is up to the mode to define.)

3.1 Formal description of Sakura

We specify the Sakura tree coding in Figure 4 below. In our specification we use the Augmented Backus-Naur Form (ABNF), which is used for describing the syntax of programming languages or document formats [20]. (We refer to the Wikipedia entries for ABNF.)

In short, an ABNF specification is a set of derivation rules, where a *non-terminal* symbol is assigned a sequence of symbols or a choice of a set of sequences of symbols, separated by |. Symbols that never appear on a left side are terminals. Non-terminal symbols are enclosed between the pair ⟨⟩. In our case, the *terminals* are either the frame bits '0' and '1', frame bits whose value is specified in the text (FRAME_BIT), bits coming from the message (MESSAGE_BIT), bits coming from chaining values (CHAINING_BIT), or the empty string ''. The expression $n\langle x\rangle$ denotes a sequence of n elements of type $\langle x\rangle$. In the language of [6], the produced nodes compose a *tree template*, i.e., a tree with placeholders for message bits and chaining values.

⟨*final node*⟩ ::= ⟨*node*⟩ '1'

⟨*inner node*⟩ ::= ⟨*node*⟩ ⟨*padSimple*⟩ '0'

⟨*node*⟩ ::= ⟨*message hop*⟩ | ⟨*chaining hop*⟩ | ⟨*kangaroo hopping*⟩

⟨*kangaroo hopping*⟩ ::= ⟨*node*⟩ ⟨*padSimple*⟩ ⟨*chaining hop*⟩

⟨*message hop*⟩ ::= ⟨*message bit string*⟩ '1'

⟨*message bit string*⟩ ::= '' | ⟨*message bit string*⟩ MESSAGE_BIT

⟨*chaining hop*⟩ ::= nrCVs⟨*CV*⟩ ⟨*coded nrCVs*⟩ ⟨*interleaving block size*⟩ '0'

⟨*CV*⟩ ::= *n*CHAINING_BIT

⟨*coded nrCVs*⟩ ::= ⟨*integer*⟩ ⟨*length of integer*⟩

⟨*integer*⟩ ::= ⟨*frame byte string*⟩

⟨*frame byte string*⟩ ::= '' | ⟨*frame byte string*⟩ 8FRAME_BIT

⟨*length of integer*⟩ ::= 8FRAME_BIT

⟨*interleaving block size*⟩ ::= ⟨*mantissa*⟩ ⟨*exponent*⟩

⟨*mantissa*⟩ ::= 8FRAME_BIT

⟨*exponent*⟩ ::= 8FRAME_BIT

⟨*padSimple*⟩ ::= '1' | ⟨*padSimple*⟩ '0'

Fig. 4. Definition of Sakura tree hash coding

The production rules for ⟨*node*⟩ express which sequences of hops can be encoded in a node. E.g., if the node contains one message hop followed by two chaining hops because of kangaroo hopping, ⟨*node*⟩ expands to ⟨*message hop*⟩ ⟨*padSimple*⟩ ⟨*chaining hop*⟩ ⟨*padSimple*⟩ ⟨*chaining hop*⟩.

The length of the chaining values ⟨*CV*⟩ is n bits, where n is a multiple of 8 to ensure byte-alignment. If the function f has worst-case (or collision resistance) security strength s [18], then we take n equal to s multiplied by two and rounded to a multiple of 8, i.e.,

$n = 8\lceil s/4 \rceil$. In the case of a sponge function with capacity c, $n = 8\lceil c/8 \rceil$, e.g., if $c = 256$ bits, then a $\langle CV \rangle$ consists of 32 bytes [2]. We assume that the security strength of the inner function is known from the context.

When interpreted as an integer, a byte has the value

$$\sum_{0 \le i < 8} b_i 2^i, \tag{1}$$

where the first bit in a byte has index 0 and the last 7.

The $\langle$*coded nrCVs*$\rangle$ codes the number of chaining values and is a positive integer. It consists of two fields:

- $\langle$*integer*$\rangle$: a byte string that can be decoded to an integer using the function `OS2IP(X)` specified in the RSA Labs standard PKCS#1[12],
- $\langle$*length of integer*$\rangle$: a single byte that codes the length (in bytes) of the $\langle$*integer*$\rangle$ field.

The interleaving block size codes an integer using a floating point representation. Its first byte is the mantissa m and its second byte is the exponent e. The value of the interleaving block size I is then given by

$$I = 2^e(2m + 1) .$$

The largest possible value that the interleaving block size can have with this coding is $(2^9 - 1)2^{255}$, obtained by setting all bits in its coding to 1. In practice no message will ever attain this length and we use it to denote that there is no interleaving. This value will be denoted by $I = \infty$ in the remainder of this paper.

Within a node, the chaining bits must come from child nodes with increasing indexes, starting from 0 at the beginning of the node, across all chaining hops of the node. When kangaroo hopping is not used, the node indexing matches the hop indexing, but not in general.

The encoding of the message bits in the tree should allow the reconstruction of the message by applying `GetMessage` to the final hop according to following definition. Note that reconstructing the message from the nodes is an operation that is relevant in proving soundness rather than something to be used in practice.

Definition 1. `GetMessage` *is defined by the following recursion:*

- `GetMessage(`*message hop*`)` *is the message hop's message string*
- `GetMessage(`*chaining hop*`)` = `DeInterleave`*(L, I), where*
  - *L is an ordered list obtained by calling* `GetMessage()` *on each child hop,*
  - *I is the input chaining hop's interleaving block size attribute, and*
  - `DeInterleave(`*L, I*`)` *extracts the first I bits from* L_0*, then the first I bits from* L_1*, …, up to the last item of list, then back to* L_0*, and so on, until all strings in L are empty. Extracting more bits than available reduces to extracting all remaining bits.*

Definition 2. *A tree template is* SAKURA*-compatible if its nodes are compliant with the coding specified in Figure 4, if the number of* $\langle CV \rangle$ *and the block interleaving size are coded as explained above, and if the chaining bits and message bits are as defined above.*

3.2 Illustrations

We apply the SAKURA encoding to the examples depicted on Figures 1, 2 and 3. In these examples, we use the following conventions. Bit values are written as `0` or `1`, while sequences of 8 bits can be written in hexadecimal notation prefixed with `0x` with numerical value following Eq. (1). Spaces are inserted only for reading purposes. If M_α is a message hop, we denote by $\mathtt{M}_\alpha$ its message bits. Similarly, if Z_α is a chaining hop, we denote by $\{\mathtt{I}_\alpha\}$ the encoding of its interleaving block size. Then, $\mathtt{CV}_\beta$ is the chaining value resulting from the application of f to the node with index β. Finally, $\mathtt{0}^*$ indicates a non-negative number of bits `0` determined by the tree hash mode, typically inserted for alignment purposes.

The example corresponding to Figure 1 is given in Table 1. In the final node, ⟨*node*⟩ expands to ⟨*message hop*⟩ ⟨*padSimple*⟩ ⟨*chaining hop*⟩, while in all other nodes it simply expands to ⟨*message hop*⟩.

The example corresponding to Figure 2 is given in Table 2. In two inner nodes, ⟨*node*⟩ expands to ⟨*message hop*⟩ ⟨*padSimple*⟩ ⟨*chaining hop*⟩ and in two other inner nodes, ⟨*node*⟩ expands to ⟨*message hop*⟩. In the final node, ⟨*node*⟩ simply expands to ⟨*chaining hop*⟩.

For sequential hashing (Figure 3), this reduces to a single final node containing `M11`, and the relationship between the inner and outer hash functions reduces to

$$F(M) = f(M||\mathtt{11}). \tag{2}$$

Node index	**Encoding**
2	$\mathtt{M}_3\mathtt{1}\ \mathtt{10}^*\ \mathtt{0}$
1	$\mathtt{M}_2\mathtt{1}\ \mathtt{10}^*\ \mathtt{0}$
0	$\mathtt{M}_1\mathtt{1}\ \mathtt{10}^*\ \mathtt{0}$
$*$	$\mathtt{M}_0\mathtt{1}\ \mathtt{10}^*\ \mathtt{CV}_0\ \mathtt{CV}_1\ \mathtt{CV}_2\ \mathtt{0x03}\ \mathtt{0x01}\ \{\mathtt{I}_*\}\mathtt{0}\ \mathtt{1}$

Table 1. Encoding for the hop tree example depicted in Figure 1

Node index	**Encoding**
10	$\mathtt{M}_{11}\mathtt{1}\ \mathtt{10}^*\ \mathtt{0}$
1	$\mathtt{M}_{10}\mathtt{1}\ \mathtt{10}^*\ \mathtt{CV}_{10}\ \mathtt{0x01}\ \mathtt{0x01}\ \{\mathtt{I}_1\}\mathtt{0}\ \mathtt{10}^*\ \mathtt{0}$
00	$\mathtt{M}_{01}\mathtt{1}\ \mathtt{10}^*\ \mathtt{0}$
0	$\mathtt{M}_{00}\mathtt{1}\ \mathtt{10}^*\ \mathtt{CV}_{00}\ \mathtt{0x01}\ \mathtt{0x01}\ \{\mathtt{I}_0\}\mathtt{0}\ \mathtt{10}^*\ \mathtt{0}$
$*$	$\mathtt{CV}_0\ \mathtt{CV}_1\ \mathtt{0x02}\ \mathtt{0x01}\ \{\mathtt{I}_*\}\mathtt{0}\ \mathtt{1}$

Table 2. Encoding for the hop tree example depicted in Figure 2

4 SAKURA-compatible tree hash modes and soundness

We define SAKURA-compatible tree hash modes in the following way.

Definition 3. *A tree hash mode is SAKURA-compatible if it generates only SAKURA-compatible templates.*

We will now prove that any Sakura-compatible tree hash mode, as well as the union of any set of Sakura-compatible tree hash modes, is sound by proving a number of lemmas.

We start by defining $\mathcal{S}$ as a tree hash mode that can generate all Sakura-compatible templates. By construction, this mode is Sakura-compatible. Its parameters A must describe the whole hop tree structure with each hop's attributes, plus the length of all message blocks and the number of zeroes inserted by $\langle padSimple \rangle$. This mode is not meant to be used in practice but only in the scope of this proof.

Lemma 1. *Given a node instance produced by $\mathcal{S}$ (i.e., with Sakura coding) and the knowledge of the security strength of f, one can recover indices of all hops, the message strings of the message hops, the location and indices (relative to the given node instance index) of the chaining values, and the interleaving block size attributes of all chaining hops.*

Proof. From the definition of Sakura in Figure 4, it is clear that a $\langle node \rangle$, obtained after removing the trailing bit from a $\langle final\ node \rangle$ or $\langle inner\ node \rangle$ (and in the latter case, also removing the $\langle padSimple \rangle$ padding), consists of a possible $\langle message\ hop \rangle$ followed by one or more $\langle chaining\ hop \rangle$s, with simple padding in between. A $\langle chaining\ hop \rangle$ in turn consists of a sequence of $\langle CV \rangle$s followed by an encoding of their number and a $\langle interleaving\ block\ size \rangle$.

Parsing a $\langle node \rangle$ can be done starting at the end. If the last bit is 1 it simply consists of a single message hop. Otherwise, it ends with a chaining hop. In the latter case, the last two bytes code the interleaving block size of the chaining hop and the byte before that denotes the length of the field coding the number of chaining values and allows localizing it. Decoding this field yields the number of chaining values and together with their lengths uniquely determines their positions in the node, including the start of the chaining hop in the node. This allows continuing the parsing until reaching the beginning of the $\langle node \rangle$ or the end of the $\langle message\ hop \rangle$ in the beginning of the $\langle node \rangle$.

The interleaving block size of a chaining hop can be computed from the coding in $\langle interleaving\ block\ size \rangle$ at its end and the message string of the $\langle message\ hop \rangle$ (if any) can be obtained by removing the trailing bit 1.

The index of the last $\langle chaining\ hop \rangle$ is that of the $\langle node \rangle$. Whenever kangaroo hopping is used, the index of a $\langle chaining\ hop \rangle$ or $\langle message\ hop \rangle$ is recursively the index of the next $\langle chaining\ hop \rangle$ with 0 concatenated to it. This is in line with the node indexing specified in Section 3.1.

The indices of the nodes corresponding with the $\langle CV \rangle$s in a $\langle node \rangle$ can be obtained by appending to the last hop index 0 for the first CV, 1 for the second and so on, throughout all the $\langle chaining\ hop \rangle$s of the node instance from beginning to end. □

To prove the soundess of $\mathcal{S}$, we use the three conditions that are shown to be sufficient in [6]. We now informally summarize them.

- The mode must be *tree-decodable*. This means that the tree can be parsed to retrieve the frame bits, message bits and chaining bits unambiguously. There must be a decoding algorithm A_{decode} that can parse the tree progressively on subtrees, starting from the final node only, and each time adding a new inner node and pointing at the corresponding chaining value. Also, the process must terminate by requiring that one can distinguish between complete and compliant trees, subtrees that are compliant except for some missing nodes (called *final-subtree-compliant*), and incompliant trees.
- The mode must be *message-complete*. This means that the message can be reconstructed from the complete tree.

– The mode must be *final-node separable*. This essentially means that one can tell the difference between final nodes and inner nodes.

Lemma 2. *The tree hash mode $\mathcal{S}$ is tree-decodable.*

Proof. First, there are no tree instances that are both compliant and final-subtree-compliant. Lemma 1 proves that one can always unambiguously decode chaining values and distinguish them from other kind of bits given only one node instance. This means that a final subtree S is a proper final subtree iff there are chaining values pointing to nodes missing in S.

Second, the algorithm A_{decode} can be defined as follows. Given a tree instance S with index set J, it first recursively decodes tree node instances of S as in the proof of Lemma 1. If at any point, the coding does not follow the grammar defined in Figure 4 or when the string is too short to contain the number of $\langle CV \rangle$s coded in $\langle \textit{coded nrCVs} \rangle$, it returns "incompliant".

The algorithm A_{decode} then looks for nodes that have chaining values pointing to nodes missing in S (i.e., whose index is not in J). If there no such chaining values, return "compliant". Otherwise, return "final-subtree-compliant" and the index of such a missing node using a deterministic rule (e.g., the missing node with the first index in lexicographical order).

The algorithm A_{decode} runs in linear time in the number of bits in the tree instance, as can be seen in the proof of Lemma 1. □

Lemma 3. *The tree hash mode $\mathcal{S}$ is message-complete.*

Proof. Given a compliant tree instance S, the algorithm A_{message} can be defined similarly to the `GetMessage` function in Definition 1. From Lemma 1, the necessary hop attributes can be extracted from the tree instance.

Clearly, this algorithm runs in linear time in the number of bits in the tree instance. □

Lemma 4. *The tree hash mode $\mathcal{S}$ is final-node separable.*

Proof. Sakura enforces domain separation between final and inner nodes, as the trailing bit of a final node is always `1` and that of an inner node is always `0`. □

Theorem 1. *Any Sakura-compatible tree hash mode, as well as the union of any set of Sakura-compatible tree hash modes, is sound.*

Proof. From the previous lemmas and [6, Theorem 1], it follows that $\mathcal{S}$ is sound.

The set $\mathcal{Z}_{\mathcal{T}}$ of tree templates that a Sakura-compatible tree hash mode $\mathcal{T}$ produces is included in those produced by $\mathcal{S}$, i.e., $\mathcal{Z}_{\mathcal{T}} \subseteq \mathcal{Z}_{\mathcal{S}}$. Therefore, $\mathcal{T}$ can be implemented by running $\mathcal{S}$ as a sub-procedure, after encoding $\mathcal{T}$'s parameters in the format that $\mathcal{S}$ accepts. This only restricts what an attacker can query, so $\mathcal{T}$ is at least as secure as $\mathcal{S}$.

When taking the union of two or more Sakura-compatible tree hash modes, if the tree instances produced by each of the united modes are Sakura-compatible, then so are the tree instances produced by the union. It follows that the union of Sakura-compatible tree hash modes is Sakura-compatible and the argument above carries over to the union. □

5 Examples of tree hash modes

In this section we give some examples of tree hash modes that can be realized with the SAKURA coding. In general, specifying a mode mainly comes down to specifying how the tree grows as a function of the size of the input message. These modes are parameterized and the value of the parameters must be known at the time of hashing a message.

For fully specifying a tree hash mode compliant with SAKURA, one has to specify the number of hops and their indices, how the message bits are distributed onto message hops, and for each chaining hop whether kangaroo hopping is applied. In addition, the mode has to specify the length of the padding elements as they appear in the grammar of Figure 4. For the padding between hops, this can be derived from a simple strategy, such as always align on bytes, on 64-bit boundaries or on the input block size (or rate) of the inner hash function f. If desired, the mode can also specify how to use the padding at the end of inner nodes to ensure that sibling nodes executed in parallel branches have the same length.

In our examples, unless otherwise specified, the message is split into B-bit blocks M_i, i.e.,

$$M = M_0||M_1||\dots||M_{n-1},$$

with $n = \lceil |M|/B \rceil$ and where the last block M_{n-1} may be shorter than B bits.

5.1 Final node growing

With final node growing, the hop tree has fixed height 1 and the number of leaves increases as a function of the input message length. There is only a single chaining hop, namely the final hop. The indices of the message hops are integers 0 to $n-1$ and the message string in message hop with index i is M_i, hence each message hop has a fixed maximum size B. Interleaving is not applied, so the interleaving block size in the final hop is $I = \infty$.

This mode can be useful to enable a large amount of potential parallelism, namely up to $n = \lceil |M|/B \rceil$ message hops can be processed in parallel if the corresponding message blocks are available at the same time. In practice, a number p of independent processes P_j, $j = 0, \dots, p-1$ can be set up, which does not depend on the tree structure other than in the total number of message hops. Each process P_j could take care of message hops with indices $j + kp$.

The drawback of this method is an extra cost proportional to the message length, as n chaining values of length c must be processed in the final node. This extra cost represents approximately a fraction c/B of the nominal work, which can be made arbitrarily small by choosing B large enough.

This mode has two parameters:

- B, the maximum size of message string in message hops, and
- whether or not kangaroo hopping shall be applied in the final hop.

5.2 Leaf interleaving

With leaf interleaving, the hop tree has a fixed topology, i.e., its height is 1 and it has D message hops, with D a parameter. The size of the message hops depends on the input message length. The message is distributed over the leaves as it arrives in blocks of

size B. The message hops have indices $i \in \{0, 1, \ldots, D-1\}$ and their message string is $M_i||M_{i+D}||\ldots||M_{i+(s_i-1)D}$ with $s_i = \lceil (n-i)/D \rceil$. The interleaving block size in the final hop shall be set to $I = B$. If $|M| < DB$, there are message hops with zero message bits. (Note that an alternate message assignment procedure is proposed later in this section.)

This mode is useful if one wants to hash a message in up to D parallel threads. The drawback is that D represents a limit in the potential parallelism, and this value must be chosen beforehand.

This method has a fixed extra cost, independent of the message length, as the final node has to process D chaining values.

This mode has three parameters:

- B, the interleaving block size,
- D, the number of message hops, and
- whether or not kangaroo hopping shall be applied in the final hop.

Ensuring equal-length inner nodes In the implementation, it may be interesting to ensure that all the D nodes processed simultaneously have equal block length w.r.t. the inner function f. For the D (or $D-1$, if kangaroo hopping is applied) inner nodes, this can be achieved by systematically adding bits with value `0` in the $\langle padSimple \rangle$ padding of the $\langle inner\ node \rangle$ production rule. A simple procedure consists in adding padding bits so as to match the length of the longest inner node.

When kangaroo hopping is applied, the final node has the possibility to add padding bits after the message hop, just before the chaining values of the $D-1$ inner nodes are added, i.e., in the $\langle padSimple \rangle$ padding of the $\langle kangaroo\ hopping \rangle$ production rule. The processing of all D pieces of message can therefore be aligned, even with kangaroo hopping.

Avoiding systematic block alignment Implementations can also be made easier when the interleaving block size B is equal to, or a multiple of, the input block size (or rate) r of the inner hash function f. This avoids re-shuffling of the input message bytes, in particular for implementations that process less than D nodes in parallel.

But there is a potential efficiency problem in this case if care is not taken in the way the message bits are spread on the D message hops, in particular for the last $|M| \bmod DB$ bits. If the message bits are cyclically spread by blocks of B bits onto the D message hops until exhaustion, message hops will very often contain a whole number of r-bit blocks. After adding frame and padding bits, the resulting nodes will systematically be just a few bits longer than a whole number of r-bit blocks. This would be unfortunate, as the inner function f would need to process an additional block containing only frame and padding bits and no message payload, and this amounts to quite an extra fixed cost compared to just processing the final hop. E.g., if $B = r = 1024$, $D = 4$ and the message length is $3208 \pmod{4096}$, the last 3208 bits would be split as $1024 + 1024 + 1024 + 136$, causing 3 extra blocks to be absorbed without any payload.

To address this, the mode can simply spread the last $|M| \bmod DB$ bits as equally as possible (up to, say, bytes) onto the D hops. The mode remains Sakura-compatible since the `GetMessage` function in Definition 1 simply concatenates the last blocks of each nodes, even if they have less than $I = B$ bits. Taking the same example as above, the last 3208 bits could instead be spread as $800 + 800 + 800 + 808$ and avoid the 3 extra blocks mentioned above. Note that this technique requires to know the end of the message DB bits in advance or to have a buffer of DB bits.

Let us specify a possible alternate procedure, which we illustrate in the case that the message and interleaving block sizes are byte-aligned, i.e., $|M|$ and B are multiples of 8. With $m = |M|/8$ and $b = B/8$, we concentrate on the last m mod Db bytes. If m mod $Db = 0$, message hops all contain whole blocks, and there is nothing to do. If m mod $Db > 0$, we proceed as follows.

- Let M' be the last m mod Db bytes of M.
- For i from 0 to $D - 1$:
  - Move the first $\lfloor \frac{m+i}{D} \rfloor$ remaining bytes from M' to the i-th message hop.

5.3 Macro- and microscopic leaf interleaving

Different orders of magnitudes for the block interleaving size I can be useful depending on the kind of parallelism that one wishes to exploit. At one end of the spectrum is a single-instruction multiple-data (SIMD) unit of a modern processor or core. Such a unit can naturally compute two (or more) instances of the same primitive in parallel. For the processor or core to be able to fetch data in one shot, it is interesting to process simultaneously data blocks that are located close to one another. Suitable I values for addressing this are, e.g., 64 bits or the input block size (or rate) of f.

At the other end of the spectrum is the case of independent processors, cores or even machines that process different parts of the input in parallel. In contrast, it is here important to avoid different processors or cores having to fetch the same memory addresses, or to avoid copying identical blocks of data for two different machines. Suitable I values for addressing this are in the order of kilobytes or megabytes.

The two cases can coexist, for instance, if several cores are used to hash in parallel and each core has a SIMD unit. A suitable tree structure is one with height 2, as depicted in Figure 2. The subtrees rooted by Z_0 and Z_1 are handled by different cores, whereas the leaves are processed together in the SIMD units. The final hop Z_* splits the message to hash into macroscopic blocks (large I), while the intermediate chaining hops Z_0 and Z_1 further split the macroscopic blocks into microscopic blocks suitable for the SIMD unit (small I).

The tree hash mode of Section 5.2 can be generalized to support such mixed interleaving block sizes.

5.4 Binary tree

With a binary tree, the tree topology evolves as a function of the input message size. All chaining hops have degree 2, and the message strings in the message hops have a fixed maximum size B. The height of the message hops depends on the length of the message and the position of the message string of that message hop in the message. Interleaving is not applied, so the interleaving block size in all chaining hops is $I = \infty$.

This mode is useful if one wants to limit the effort to re-compute the hash when only a small part of the message changes. This requires that the chaining values are stored. Hence, in this application, kangaroo hopping is not interesting.

The hop tree can be defined in the following way. We first arrange the message blocks M_i in a linear array to form the message hops. Each message hop can be seen as a tree with height 0. Then we apply the following procedure iteratively: combine the trees in pairs starting from 0 by adding a chaining hop and connecting the two root hops to it. If

the number of trees is odd, the last tree is just kept as such. Applying this $\lceil \log_2 n \rceil$ times will reduce the number of trees to a single one. The most recently added hop is the final hop. The indices of the hops follow directly from the tree topology. Figure 5 illustrates the case for three different numbers of blocks.

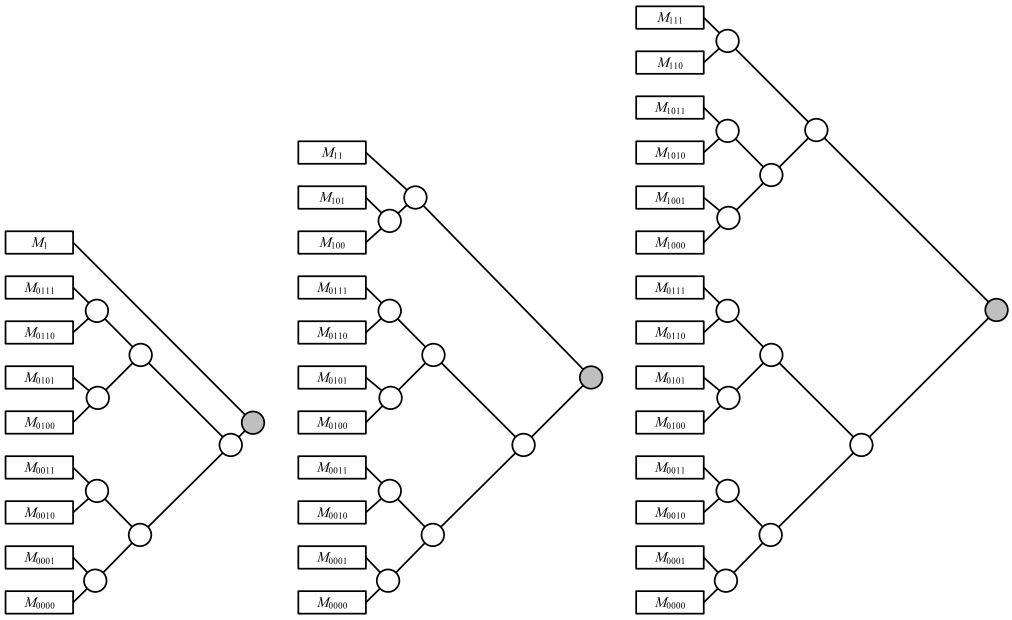

Fig. 5. Examples of binary trees with 9, 11 and 14 leaves

This mode has one parameter:

– B, the maximum size of message strings in the message hops.

5.5 Equal parts

If the length of the input message is known in advance, one can choose to divide the message in a chosen number D of (almost) equal parts. The hop tree has a fixed topology, i.e., it has height 1, with the final hop and D message hops. The size of the message hops depends on the message size and on D, namely, $B = \lceil |M|/D \rceil$, and message hop with index i contains M_i. Interleaving is not applied, so the interleaving block size in the final hop is $I = \infty$.

This mode has two parameters:

– D, the number of message hops, and
– whether or not kangaroo hopping shall be applied in the final hop.

6 Application to Keccak and SHA-3

In the future, one may standardize tree hash modes. By adopting Sakura coding from the start, any future Sakura-compatible tree hash mode using Keccak [4] as inner function

can be introduced while guaranteeing soundness of the union of that new mode and any compatible tree hash mode(s) defined up to that point. The sequential hash mode will then simply correspond with the single-hop case of Figure 3. As shown in Eq. (2), this comes down to appending two bits to the message before presenting to the inner function:

– a bit `1` to indicate it is a message hop, and
– a bit `1` to indicate it is the final node.

The draft of FIPS 202 contains both the arbitrary output length instances SHAKE128 and SHAKE256, called *extendable-output functions*, and the SHA-2 drop-in replacement instances SHA3-224 to SHA3-512 with their traditional fixed output length [19]. In addition, the different SHA-3 instances and possible future uses of KECCAK are domain-separated by appending a (short) fixed suffix. While KECCAK's multi-rate padding already offers domain seperation between instances with different capacities, the suffix also separates instances with equal capacities. In particular, the last bit of the suffix is always `1`, reserving `0` for future uses, and the one before last bit is `1` for the extendable-output functions and `0` for the SHA-2 drop-in's.

All the instances in the FIPS 202 draft are sequential. The extendable-output functions SHAKE128 and SHAKE256 have a suffix that allows for future SAKURA-compatible tree hash modes, as we discuss below. We now focus on these two functions, as it would not make much sense to combine tree hashing with the SHA-2 drop-in's. The reason is that to carry over the full security of the underlying hash function, one has to set the tree-level chaining value length n equal to the capacity c (or n equal to twice the security strength in general). As for SHA3-n, the FIPS 202 draft sets $c = 2n$, one would need to define some ad-hoc construction on top of it to get two output blocks (like a mask generating function), and this would be absurd given that SHA3-n is obtained by truncating KECCAK's output.

The extendable-output functions are defined in two steps:

– first RawSHAKE128$(\triangle)$ = KECCAK$[c = 256](\triangle||\texttt{11})$, with `11` the domain separation suffix of the extendable-output functions, and
– then SHAKE128(M) = RawSHAKE128$(M||\texttt{11})$,

and similarly for SHAKE256 [19]. This means that RawSHAKE128/256 can be seen as the inner function $f(\triangle)$ and SHAKE128/256 as the outer function $F(M)$ as in Eq. (2). Future tree hash modes can possibly use SAKURA coding on top of RawSHAKE128/256. In the expressions above, the notation $\triangle$ suggests a SAKURA-compatibly formatted input.

As domain separation and SAKURA coding are realized by appending sufficiently few bits, there is no performance penalty induced by these suffixes for messages that consist of byte sequences and rate values that are a multiple of 8. The domain separation suffix consists of two bits (i.e., `11`), while the last bits induced by SAKURA coding depend on the type of node, i.e., whether the last hop is a message (`1`) or a chaining hop (`0`), followed by whether the node is final (`1`) or not ($\texttt{10}^*\texttt{0}$). In addition, the implementer has also to apply the multi-rate padding ($\texttt{10}^*\texttt{1}$). Together, all this fits in one byte (assuming that the mode does not add any extra bit `0` at the end of inner nodes, the use of which being discussed in Section 5.2).

Let us take as example RawSHAKE128$(\triangle)$ = KECCAK$[c = 256](\triangle||\texttt{11})$, although this works equally well with RawSHAKE256 as it uses the same suffixes and only the capacity differs. With sequential hashing, $\triangle$ reduces to $\triangle = M||\texttt{11}$ and, together, the suffixes and multi-rate padding can be seen as padding with $\texttt{111110}^*\texttt{1}$. In hexadecimal notation, using

KECCAK's bit numbering conventions [4], namely from the least to the most significant bit and consistently with Eq. (1), this becomes `0x1F 0x00`* `0x80` if $|M| \bmod r < r - 8$ or `0x9F` otherwise. For other node types, the different byte-level paddings are shown in Table 3.

Node type	Bit-level	Byte-level padding
inner node with chaining hop	`010 11 10`*`1`	`0x3A 0x00`* `0x80` (or `0xBA`)
inner node with only message hop	`110 11 10`*`1`	`0x3B 0x00`* `0x80` (or `0xBB`)
final node with chaining hop	`01 11 10`*`1`	`0x1E 0x00`* `0x80` (or `0x9E`)
final node with only message hop	`11 11 10`*`1`	`0x1F 0x00`* `0x80` (or `0x9F`)

Table 3. Bit- and byte-level padding for RawSHAKE128/256, assuming no extra bit `0` at the end of inner nodes. The last row corresponds to SHAKE128/256 [19].

7 Conclusion

We showed that it is possible to define a fairly general coding for tree hashing, such that any tree hash mode compatible with this coding is sound and remains sound even when considered in a larger set of compatible tree hash modes. We proposed a concrete coding called SAKURA. In the current FIPS 202 draft, the SHA-3 extendable-output functions chosen by NIST adopt SAKURA for sequential hashing as a special case of tree hashing.

Acknowledgments

We would like to thank Stefan Lucks, Dan Bernstein and the members of the NIST hash team for useful discussions.

References

1. M. Bellare and P. Rogaway, *Random oracles are practical: A paradigm for designing efficient protocols*, ACM Conference on Computer and Communications Security 1993 (ACM, ed.), 1993, pp. 62–73.
2. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, `http://sponge.noekeon.org/`, pp. 181–197.
3. ———, *Sufficient conditions for sound tree hashing modes*, Symmetric Cryptography (Dagstuhl, Germany) (H. Handschuh, S. Lucks, B. Preneel, and P. Rogaway, eds.), Dagstuhl Seminar Proceedings, no. 09031, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.
4. ———, *The* KECCAK *reference*, January 2011, `http://keccak.noekeon.org/`.
5. ———, KECCAK *and the SHA3 standardization*, presentation at NIST, February 2013, `http://csrc.nist.gov/groups/ST/hash/sha-3/documents/Keccak-slides-at-NIST.pdf`.
6. ———, *Sufficient conditions for sound tree and sequential hashing modes*, International Journal of Information Security (2013), `http://dx.doi.org/10.1007/s10207-013-0220-y`.
7. J. Coron, Y. Dodis, C. Malinaud, and P. Puniya, *Merkle-Damgård revisited: How to construct a hash function*, Advances in Cryptology – Crypto 2005 (V. Shoup, ed.), LNCS, no. 3621, Springer-Verlag, 2005, pp. 430–448.
8. I. Damgård, *A design principle for hash functions*,, Advances in Cryptology – Crypto '89 (G. Brassard, ed.), LNCS, no. 435, Springer-Verlag, 1989, pp. 416–427.
9. Y. Dodis, L. Reyzin, R. Rivest, and E. Shen, *Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to MD6*, Fast Software Encryption (O. Dunkelman, ed.), Lecture Notes in Computer Science, vol. 5665, Springer, 2009, pp. 104–121.

10. 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, 2008, `http://skein-hash.info/`.
11. S. Gueron, *A j-lanes tree hashing mode and j-lanes SHA-256*, Journal of Information Security **4** (2013), 4–11.
12. RSA Laboratories, *PKCS # 1 v2.2 RSA Cryptography Standard*, 2012.
13. S. Lucks, *Tree hashing: A simple generic tree hashing mode designed for SHA-2 and SHA-3, applicable to other hash functions*, Early Symmetric Crypto (ESC), 2013.
14. S. Lucks, D. McGrew, and D. Whiting, *Batteries included: Features and modes for next generation hash functions*, The Third SHA-3 Candidate Conference, 2012.
15. 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.
16. R. C. Merkle, *Secrecy, authentication, and public key systems, PhD thesis*, UMI Research Press, 1982.
17. NIST, *Mailing list on NIST's cryptographic hash workshops and hash algorithm competition*, `http://csrc.nist.gov/groups/ST/hash/email_list.html`.
18. _____, *NIST special publication 800-57, recommendation for key management (revised)*, March 2007.
19. _____, *Federal information processing standard 202, SHA-3 standard: Permutation-based hash and extendable-output functions (draft)*, April 2014, `http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_standard_fips202.html`.
20. P. Overell, *Augmented BNF for syntax specifications: ABNF*, Internet Request for Comments, RFC 5234, January 2008.
21. R. Rivest, B. Agre, D. V. Bailey, S. Cheng, C. Crutchfield, Y. Dodis, K. E. Fleming, A. Khan, J. Krishnamurthy, Y. Lin, L. Reyzin, E. Shen, J. Sukha, D. Sutherland, E. Tromer, and Y. L. Yin, *The MD6 hash function – a proposal to NIST for SHA-3*, Submission to NIST, 2008, `http://groups.csail.mit.edu/cis/md6/`.
22. P. Sarkar and P. J. Schellenberg, *A parallelizable design principle for cryptographic hash functions*, Cryptology ePrint Archive, Report 2002/031, 2002, `http://eprint.iacr.org/`.
23. M. Torgerson, R. Schroeppel, T. Draelos, N. Dautenhahn, S. Malone, A. Walker, M. Collins, and H. Orman, *The SANDstorm hash*, Submission to NIST, 2008, `http://www.sandia.gov/scada/documents/SANDstorm_Submission_2008_10_30.pdf`.