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.

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 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* [4]. 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.

In [4] we have specified a set of conditions for a tree (or sequential) hashing mode to be *sound*, i.e., if 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.

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.

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 3.1 we specify Sakura, the coding we propose, and 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 make some concrete proposals in the context of making the SHA-3 standard tree-hashing ready.

This paper follows ideas presented in [3, Slides 54-59].

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. The section ends with figures illustrating the concepts.

2.1 Modeling tree hash modes

We refer to [4, 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 (i.e., constant-valued bits, e.g., for padding or domain separation).

The union of tree hash modes is defined in [4, Section 7.4]. 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. 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 takes as input 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*. 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 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.

For node indexing, we naturally adopt the convention that the index of a node equals the index of its hop of smallest height.

Relating hops to final and inner nodes, this gives:

Final node Result of applying f to the node is the output of F. The last hop in this node is the final hop.

Inner node Result of applying f to the 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.

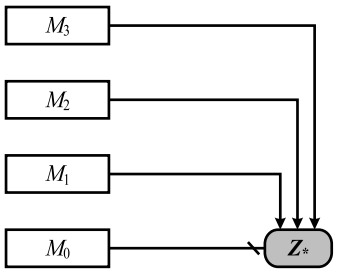

Fig. 1. Example of a hop tree with application of kangaroo hopping. M_0 and Z are in the same node.

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.

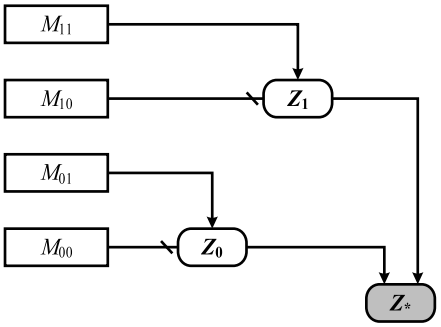

Fig. 2. Another example of a hop tree. M_{00} and Z_0 are in the same node, as well as M_{10} and Z_1.

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.

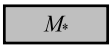

Fig. 3. Example of a hop tree with a single node

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, tree-decodability is one condition, as detailed in Section 4. In our scope, this means that the individual parts (e.g., message bits, chaining values) can be recovered. 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.

3.1 Formal description of Sakura

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 octets;
    - 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 capacity 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 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 [8]. (We refer to the Wikipedia entries for Backus-Naur Form and Augmented Backus-Naur Form for an introduction and further references.)

In short, a BNF 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 $\langle\rangle$. In our case, the *terminals* are the two bit values `'0'` and `'1'` and the empty string `''`. The expression $n\langle x\rangle$ denotes a sequence of n elements of type $\langle x\rangle$.

⟨*final node*⟩ ::= ⟨*node*⟩ `'1'`

⟨*inner node*⟩ ::= ⟨*node*⟩ `'0'`

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

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

⟨*message hop*⟩ ::= ⟨*bitstring*⟩ `'1'`

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

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

⟨*integer*⟩ ::= OCTET STRING

⟨*length of integer*⟩ ::= OCTET

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

⟨*mantissa*⟩ ::= OCTET

⟨*exponent*⟩ ::= OCTET

⟨*CV*⟩ ::= OCTET STRING

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

⟨*bitstring*⟩ ::= `''` | ⟨*bitstring*⟩ BIT

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 in bytes of the chaining values ⟨*CV*⟩ is given by $\lceil c/8 \rceil$, taking the capacity c of the inner function, or equivalently, its worst-case generic security strength level multiplied by two. For example, if the capacity is 256 bits, a ⟨*CV*⟩ consists of 32 octets. We assume that the capacity of the inner function is known from the context.

When interpreted as an integer, an octet has the value

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

where the first bit in an octet has index 0 and the last 7.

The ⟨*coded nrCVs*⟩ codes the number of chaining values and is a positive integer. It consists of two fields:

- ⟨*integer*⟩: an octet string that can be decoded to an integer using the function `OS2IP(X)` specified in the RSA Labs standard PKCS#1[6].
- ⟨*length of integer*⟩: codes the length in octets of the ⟨*integer*⟩ field in a single octet.

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.

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 $\mathtt{\{I}_\alpha\mathtt{\}}$ the encoding of its interleaving block size. Then, $\mathtt{CV}_\alpha$ is the chaining value resulting from the application of f to the node with index α. Finally, $\mathtt{0}^*$ indicates a 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 maps to ⟨*message hop*⟩.

The example corresponding to Figure 2 is given in Table 2. Two inner nodes expand to ⟨*message hop*⟩ ⟨*padSimple*⟩ ⟨*chaining hop*⟩ and two inner nodes map to ⟨*message hop*⟩. The final node maps to ⟨*chaining hop*⟩.

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

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

Node index	Encoding
3	$M_3$1 0
2	$M_2$1 0
1	$M_1$1 0
$*$	$M_0$1 10* CV_1 CV_2 CV_3 0x03 0x01 {I_*}0 1

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

Node index	Encoding
11	M_{11}1 0
1	M_{10}1 10* CV_{11} 0x01 0x01 {I_1}0 0
01	M_{01}1 0
0	M_{00}1 10* CV_{01} 0x01 0x01 {I_0}0 0
$*$	CV_0 CV_1 0x02 0x01 {I_*}0 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 1. *A tree hash mode is Sakura-compatible if its nodes are compliant with the coding specified in Figure 4 and the message can be reconstructed by applying* `GetMessage` *to the final hop according to following recursion:*

- `GetMessage(`*message hop*`)` *is the message hop's message string*
- `GetMessage(`*chaining hop*`)` = `DeInterleave`*(L, interleaving block size I), where*
  - *L is an ordered list obtained by calling* `GetMessage()` *on each child hop, 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 available bits.)*

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. It is similar to computing the inverse of a permutation that is used in the sponge construction [1].

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.

Lemma 1. *Given a tree instance with Sakura coding and the knowledge of the capacity of f, one can recover indices of all hops, the message strings of the message hops and the interleaving block size attributes of all chaining hops.*

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

Parsing a ⟨*node*⟩ 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. The last two octets 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 ⟨*node*⟩ or the end of the ⟨*message hop*⟩ in the beginning of the ⟨*node*⟩.

The index of the last ⟨*chaining hop*⟩ is that of the ⟨*node*⟩. The index of any other hop in a ⟨*node*⟩ is that of the hop following it with 0 concatenated to it. The interleaving block size of a chaining hop can be computed from ⟨*interleaving block size*⟩ at its end and the message string of the ⟨*message hop*⟩ (if any) can be obtained by removing the trailing bit 1. The indices of the nodes corresponding with the ⟨*CV*⟩s of a ⟨*chaining hop*⟩ can be obtained by appending to the hop index 0 for the first CV, 1 for the second and so on. If the ⟨*chaining hop*⟩ is not the first in the ⟨*node*⟩, one shall start with appending 1 for the first ⟨*CV*⟩ and so on. This proves the lemma.

Lemma 2. *A Sakura-compatible tree hash mode is tree-decodable.*

Proof. A sufficient condition for tree-decodability is that for every node with valid coding the chaining values and the indices of the corresponding child nodes can be identified. It follows from Lemma 1 that this is the case.

Lemma 3. *A Sakura-compatible tree hash mode is message-complete.*

Proof. Given the indices of the hops, the message strings of the message hops and the interleaving block sizes of the chaining hops, Definition 1 unambiguously specifies how to reconstruct the message for any Sakura-compatible hash mode. From Lemma 1, these hop attributes can all be extracted from the node tree. It follows that the message can be reconstructed from the node tree.

Lemma 4. *Sakura enforces domain separation between final and inner nodes.*

Proof. The trailing bit of a final node is `1` and that of an inner node is `0`.

Lemma 5. *The union of two or more Sakura-compatible tree hash modes is Sakura-compatible.*

Proof. If the conditions of Definition 1 are valid for any tree obtained by either of the modes, they are valid for any tree obtained by the union of these modes.

From the previous lemmas and [4, Theorem 1] follows our main theorem:

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

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 simply 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 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. 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 chaining hash function f.

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

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

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, $j = 0, \ldots, p-1$ 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.

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.

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, independently 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.

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 rate (or block size).

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 macroscropic 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.

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,

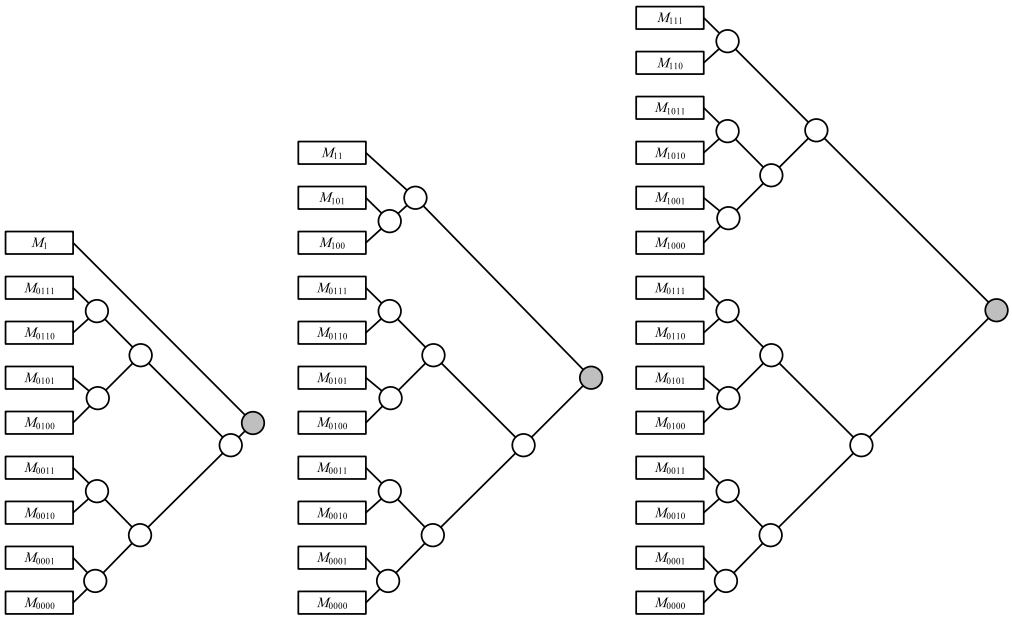

Fig. 5. Examples of binary trees with 9, 11 and 14 leaves

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, including for sequential hashing, any future Sakura-compatible tree hash mode using Keccak as inner function can be introduced while guaranteeing soundness of the union of the sequential mode and any tree hash mode one may define. 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.

We propose to adopt Sakura coding for all SHA-3 variants: the four SHA-2 drop-in replacements and the arbitrary output length instance(s) (called *sponge instances* in the remainder of this paper).

One may additionally require domain separation between SHA-3 and future uses of Keccak, or even between different instances of SHA-3. For this purpose, we propose to

apply domain separation at the level of the inner function f. For the SHA-3 instances, instead of taking for the inner function KECCAK, we propose to take variants where one or more bits are appended to the message for domain separation.

If this additional domain separation is realized by appending at most 4 bits, there is no performance penalty for messages that consist of byte sequences and rate values that are a multiple of 8. In particular, the multi-rate padding in KECCAK adds at least 2 bits and at most r bits. For byte sequences this becomes at least 1 byte and at most $r/8$ bytes. So up to 6 bits can be appended to the message without impacting these minimum and maximum values. In the case of sequential hashing, the SAKURA coding consumes 2 bits, leaving 4 bits for the additional domain separation. (See also Section 6.4 for an example.)

For domain separation between drop-in replacements and sponge instances there are multiple options. In the following subsections we make proposals for three cases, as considered by NIST [5], namely using one, two or four different capacity values.

Note that these proposals have in common that KECCAK instances with inputs that end in `0` are reserved for future use, e.g., for other input formatting conventions or additional domain separation methods [3].

In the sequel, we denote by $\triangle$ a SAKURA-compatible formatted input. In the particular case of sequential hashing, $\triangle = \texttt{M11}$ with `M` the outer hash function input.

6.1 Four capacities

With four capacities, namely $c \in \{224, 256, 384, 512\}$, one can match the security strength levels 112, 128, 192 and 256 of [7]. We propose to have four sponge instances, with matching capacities. The SHA-2 drop-in replacements simply consist of these instances with the output length fixed to the capacity value. The domains between these functions are separated by the multi-rate padding in KECCAK, so there is no need for additional domain separation. Hence, f is KECCAK where a bit `1` is appended to the input. Table 3 lists the four instances.

Combining this domain separation with SAKURA coding simply consists in appending `111` to the message before presenting to KECCAK in the case of sequential hashing.

Sponge instances	**SHA-2 drop-in replacements**
$\textsc{Keccak}[c = 224](\triangle \Vert 1)$	$\lfloor \textsc{Keccak}[c = 224](\triangle \Vert 1) \rfloor_{224}$
$\textsc{Keccak}[c = 256](\triangle \Vert 1)$	$\lfloor \textsc{Keccak}[c = 256](\triangle \Vert 1) \rfloor_{256}$
$\textsc{Keccak}[c = 384](\triangle \Vert 1)$	$\lfloor \textsc{Keccak}[c = 384](\triangle \Vert 1) \rfloor_{384}$
$\textsc{Keccak}[c = 512](\triangle \Vert 1)$	$\lfloor \textsc{Keccak}[c = 512](\triangle \Vert 1) \rfloor_{512}$

Table 3. Overview of the instances proposed in the case of four capacities. Instances on the same line are equal for the specified output length. Horizontal lines separate instances with different security strength levels.

6.2 Two capacities

Another option is to consider instances with capacities 256 and 512. We propose to have two sponge instances with each of these two capacities. The drop-in replacements for

output lengths 224 and 256 consist of truncated versions of the former, the drop-in replacements for output lengths 384 and 512 consist of truncated versions of the latter. The domains between the two sponge instances are separated by the multi-rate padding.

If there is a need for enforcing domain separation between the drop-in replacements with the same capacities and also between the drop-in replacements and the sponge instances, we propose the following domain separation method. For the sponge instances, f is Keccak, where `11` is appended to the input. For the drop-in replacements, f is Keccak, where `101` is appended for output lengths 256 and 512 and `001` is appended for output lengths 224 and 384. Table 4 lists the six instances.

Sponge instances	**SHA-2 drop-in replacements**
$\textsc{Keccak}[c = 256](\triangle\|\|\texttt{11})$	
	$\lfloor \textsc{Keccak}[c = 256](\triangle\|\|\texttt{001}) \rfloor_{224}$
	$\lfloor \textsc{Keccak}[c = 256](\triangle\|\|\texttt{101}) \rfloor_{256}$
$\textsc{Keccak}[c = 512](\triangle\|\|\texttt{11})$	
	$\lfloor \textsc{Keccak}[c = 512](\triangle\|\|\texttt{001}) \rfloor_{384}$
	$\lfloor \textsc{Keccak}[c = 512](\triangle\|\|\texttt{101}) \rfloor_{512}$

Table 4. Overview of the instances proposed in the case of two capacities. There is domain separation in the Keccak inputs between all six instances. Horizontal lines separate instances with different security strength levels.

6.3 Single capacity

In the case of a single capacity, namely $c = 512$ to match the highest security strength level of [7], there is a single sponge instance with that capacity. The drop-in replacements consist of truncated versions of this instance.

If there is a need for enforcing domain separation between all drop-in replacements and also between the drop-in replacements and the sponge instance, we propose the following domain separation method. For the sponge instance, f is Keccak, where `11` is appended to the input. For the drop-in replacements, f is Keccak, where `0001` is appended for output lengths 224, `1001` for 256, `0101` for 384 and `1101` for 512. Table 5 lists the five instances.

Sponge instances	**SHA-2 drop-in replacements**
$\textsc{Keccak}[c = 512](\triangle\|\|\texttt{11})$	
	$\lfloor \textsc{Keccak}[c = 512](\triangle\|\|\texttt{0001}) \rfloor_{224}$
	$\lfloor \textsc{Keccak}[c = 512](\triangle\|\|\texttt{1001}) \rfloor_{256}$
	$\lfloor \textsc{Keccak}[c = 512](\triangle\|\|\texttt{0101}) \rfloor_{384}$
	$\lfloor \textsc{Keccak}[c = 512](\triangle\|\|\texttt{1101}) \rfloor_{512}$

Table 5. Overview of the instances proposed in the case of a single capacity. There is domain separation in the Keccak inputs between all five instances.

6.4 Byte-level padding

When the rate r is a multiple of 8 bits and the message blocks and chaining values are all aligned on 8-bit boundaries, Sakura imposes two bits at the end of $\triangle$, after the last complete byte. These two bits depend on the type of node, i.e., if the node is final or not and if the last hop is a message or a chaining hop.

In practice, the implementer has to take into account three suffixes: the node type (the aforementioned two bits), the domain separation suffix (instance-dependent) and the multi-rate sponge padding (10^*1). Together, these suffixes fit in one byte. The combination of node type and domain-separated instances proposed above can be seen as different ways to do the padding.

Let us take as example Keccak$[c = 512](\triangle||101)$. With sequential hashing, $\triangle$ reduces to $\triangle$ = M11. The three suffixes simply combine into padding with 1110110^*1. In hexadecimal notation, using Keccak's bit numbering conventions [2], namely from the least to the most significant bit and consistently with Eq. (1), this becomes 0x37 0x00* 0x80 if $|M| \bmod r < r - 8$ or 0xB7 otherwise. For other node types, the different byte-level paddings are shown in Table 6. (Of course, the same exercise can also be done with other instances proposed in Tables 3, 4 and 5.)

Node type	Bit-level	Byte-level padding
inner node with chaining hop	0010110^*1	0x34 0x00* 0x80 (or 0xB4)
inner node with only message hop	1010110^*1	0x35 0x00* 0x80 (or 0xB5)
final node with chaining hop	0110110^*1	0x36 0x00* 0x80 (or 0xB6)
final node with only message hop	1110110^*1	0x37 0x00* 0x80 (or 0xB7)

Table 6. Bit- and byte-level padding for Keccak$[c = 512](\triangle||101)$

Acknowledgements

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

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Cryptographic sponge functions*, January 2011, http://sponge.noekeon.org/.
2. _____, *The* Keccak *reference*, January 2011, http://keccak.noekeon.org/.
3. _____, 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.
4. _____, *Sufficient conditions for sound tree and sequential hashing modes*, Cryptology ePrint Archive, Report 2009/210 (revised April 2013), 2013, http://eprint.iacr.org/.
5. J. Kelsey, *SHA3 and the future of hashing*, CT-RSA, February 2013, https://ae.rsaconference.com/US13/connect/fileDownload/session/397EA47B1FB103F0B3E87D6163C7129E/CRYP-W23.pdf.
6. RSA Laboratories, *PKCS # 1 v2.2 RSA Cryptography Standard*, 2012.
7. NIST, *NIST special publication 800-57, recommendation for key management (revised)*, March 2007.
8. P. Overell, *Augmented BNF for syntax specifications: ABNF*, Internet Request for Comments, RFC 5234, January 2008.