# Sufficient conditions for sound tree and sequential hashing modes

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

[1] STMicroelectronics
[2] NXP Semiconductors

**Abstract.** Hash functions are usually composed of a mode of operation on top of a concrete primitive with fixed input-length and fixed output-length, such as a block cipher or a permutation. In practice, the mode is often sequential, although parallel (or tree) hashing modes are also possible. The former requires less memory, while the latter has several advantages such as its inherent parallelism and a lower cost of hash value re-computation when only a small part of the input changes. In this paper, we consider the general case of (tree or sequential) hashing modes that make use of an underlying hash function, which may in turn be sequential. We formulate a set of three simple conditions for such a (tree or sequential) hashing mode to be *sound*. By sound, we mean that the advantage in differentiating a hash function obtained by applying a tree hashing mode to an ideal underlying hash function from an ideal monolithic hash function is upper bounded by $q^2/2^{n+1}$ with $q$ the number of queries to the underlying hash function and $n$ the length of the chaining values. We provide a proof of soundness in the indifferentiability framework. The conditions we formulate are easy to implement and to verify, and can be used by the practitioner to build a tree hashing mode on top of an existing hash function. We show how to apply tree hashing modes to sequential hash functions in an optimal way, demonstrate the applicability of our conditions with two efficient and simple tree hashing modes and provide a simple method to take the union of tree hashing modes that preserves soundness. It turns out that sequential hashing modes using a compression function (i.e., a hash function with fixed input-length) can be considered as particular cases and, as a by-product, our results also apply to them. We discuss the different techniques for satisfying the three conditions, thereby shedding a new light on several published modes.

**Keywords:** hash functions, tree hashing modes, sequential hashing modes, indifferentiability

## 1 Introduction

Most hash functions are iterated, that is, the message blocks are processed sequentially, and the processing of a block requires all previous blocks to be processed. This limits the efficient use of multi-processors and single-instruction multiple-data (SIMD) units, when hashing a single (long) message. By adopting tree hashing, several parts of the message can be processed simultaneously, and parallel architectures are used more efficiently in the hashing of single messages. Tree hashing has other advantages: on the condition that chaining values are kept, adapting the hash of a message after modifying only a small part of it can be done with much less effort than for a sequential hash function. On the other hand, tree hashing uses more memory and may be less efficient than sequential hashing for small messages.

Nevertheless, tree hashing can be implemented sequentially or only partially exploiting the parallelism available in the chosen tree structure. Except for the memory footprint and for short messages, it can be advantageous to use a tree enabling a high level of parallelism and let the target platform organize the computation to take advantage of this parallelism or less.

Tree hashing was already introduced in [18], and in [14] a tree hashing mode was proposed, which is provably collision-resistant if the underlying compression function (i.e., a hash function with fixed input-length(FIL) ) is collision-resistant. In this paper, we treat the general case of tree hashing modes that call a hash function with no restrictions on its input- or output-length. These modes can be used to construct tree hashing when a sequential hash function is available. Clearly, our treatment remains valid for compression functions.

Sequential hashing can be seen as a particular case of tree hashing, where all nodes of the tree except a single leaf node have degree one. The main goal of sequential hashing modes is to construct a variable input-length (VIL) hash function from a function with fixed input- and output-length such as a compression function, a permutation or a block cipher. Hence, the security of sequential and tree hashing modes can be analyzed using the same techniques, and we obtain the same conditions for both.

Our aim is to formulate a number of simple conditions for a tree or sequential hashing mode to be *sound*. In Section 3 we show that any hashing mode can be distinguished from a random oracle by generating a collision in the chaining value. We call a hashing mode sound if this is the most efficient generic distinguishing attack.

For proving the soundness, we base ourselves on the indifferentiability framework introduced by Maurer et al. in [17] and applied to hash functions by Coron et al. in [13].

### 1.1 The indifferentiability framework

The indifferentiability framework as introduced in [17] by Maurer et al. is an extension of the classical notion of indistinguishability. It deals with the interaction between systems where the objective is to show that two systems cannot be told apart by an adversary able to query both systems but not knowing a priori which system is which. It was applied by Coron et al. to iterated hash function constructions in [13]. The first system contains two subsystems: the hash function construction and the compression function. The second system contains as one of its subsystems an ideal function that has the same interface as the hash function construction in the first system. As both systems must have equal interfaces toward the distinguisher, the second system must have a subsystem offering the same interface as the compression function. This subsystem is called a *simulator*.

For hash function constructions, a random oracle usually serves as an ideal function. We use the definition of random oracle from [5]. A random oracle, denoted $\mathcal{RO}$, takes as input binary strings of any length and returns to each input a random infinite string, i.e., it is a map from $\mathbf{Z}_2^*$ to $\mathbf{Z}_2^\infty$, chosen by selecting each bit of $\mathcal{RO}(s)$ uniformly and independently, for every $s$.

In most important use cases, a construction with a proven upper bound on the differentiating advantage can replace the ideal primitive (here a random oracle) in any cryptosystem, up to a security loss bounded by that differentiating advantage. However, it has been shown in [22] that the indifferentiability framework does not cover all use cases: there are use cases where a random oracle offers security, whereas a sequential (or tree) hash function construction does not, even if proven indifferentiable. An example proposed by Ristenpart et al. is the hash-based storage auditing, in which a server has to prove to the client that it really stores the file uploaded by the client. In a nutshell, the server should return $Z = \mathrm{Hash}(\mathrm{File}||C)$ for a client-chosen challenge $C$. Using a hash function with finite state size, a malicious server could just store the last chaining value and not the complete file; using a random oracle, however, would force the server to actually store the

complete file. This protocol can easily be fixed by requesting $Z = \text{Hash}(C||\text{File})$ instead, but it nevertheless calls for caution when doing *composition*: plugging in a sequential (or tree) hash function in a protocol secure in the random oracle model. The authors define the concepts of single-stage games and multi-stage games to address these issues, where the hash-based storage auditing security implies a multi-stage game. Indifferentiability proofs following the framework of [17] only cover single-stage composition. We refer to [22] for an in-depth treatment. Fortunately, indifferentiability covers security against several generic attacks that include collision attacks, (second) preimage attacks and their variants and generic attacks against keyed modes [1, Section B].

### 1.2 Previous work

Provable security of tree hashing was already investigated in [24] and upper bounds on the differentiating advantage have been given, e.g., for the mode used in MD6 [23]. This present paper was inspired by the proofs in [23], which were unfortunately quite specific for the mode of use adopted in MD6. Instead, our goal was to be more general, and we wanted to formulate a set of simple conditions that should be easy to verify and implement, and sufficient for a tree hashing mode to be sound. We presented the ideas that lie at the basis of this paper for the first time in [8]. In the meanwhile, the authors of [23] independently set out to do the same thing, and the results of their work surfaced in the pre-proceedings of Fast Software Encryption Workshop 2009 and was later slightly refined in the proceedings version [15].

### 1.3 Our contributions

Despite the similarity between the conditions in this paper and those in [15], this present paper has a substantial added value with respect to prior work:

1. Our set of conditions allowing more freedom in the definition of the hashing mode than that of [15]. We provide a detailed comparison in Section 4.1.
2. While [15] only covers modes on top of a (FIL) compression function, our model allows inner hash functions with variable input-length. This allows a modular approach with standardized tree hashing modes calling standardized sequential hash function as inner functions.
3. Our model allows both inner hash function and outer hash functions have indefinite output-length, unlike [15].
4. Our bound on the differentiating advantage is as tight as theoretically possible (see Section 3). The bound in [15] is a factor $4r + 2$ higher, with $r$ is the number of chaining values that fit in the input of the inner hash function (e.g., $r = 4$ in MD6 [23] and $r = 3$ in SHA-2 [20]).
5. Our treatment also applies to the specific case of sequential modes. This sheds a new light on these modes and even yields improved bounds for some of them (see Sections 8.4 and 8.6).

### 1.4 Organization of the paper

The remainder of this paper is organized as follows. We first propose a general, flexible, definition of tree hashing modes in Section 2. After giving an upper bound on the achievable security level due to the birthday bound in Section 3, we introduce in Section 4 a set

of simple and easy-to-verify conditions for tree hashing modes that result in sound tree hashing and compare our conditions with the properties defined in [15]. After adapting the indifferentiability setting of [13] to tree hashing in Section 5, we provide in Section 6 the proof of an upper bound on the differentiating advantage valid for any tree hashing mode satisfying our conditions.

In Section 7, we discuss how a tree hashing mode can be built on top of a sequential hash function, provide two practical examples of sound tree hashing modes and give a simple method to take the union of tree hashing modes that preserves soundness. In Section 8, we show that the conditions we propose for tree hashing modes also make sense for sequential hashing modes. We take the point of view of these conditions to give a fresh look at techniques used in the definition of sequential modes.

Finally, Appendix A explains the difference between this version of the paper with the previous ones, Appendix B provides some illustrations related to Section 4 and Appendix C discusses the cost measure defined in Section 5.2.

## 2 Tree hashing modes

Most hash functions are constructed in a layered fashion. Traditionally, hash functions have variable input-length and fixed output-length. There is a *mode of use* that processes the input and in turn calls an underlying function $F$. Usually, this underlying function is a compression function.

In this section, we generalize this idea. We do not impose limits to the input- or output-length of the underlying function called the *inner hash function* and denoted by $\mathcal{F}$. Our generalization allows for dealing with hierarchical hash functions obtained by applying a tree hashing mode to an inner hash function that is itself sequential. Still, our treatment is generic enough to also cover the case of Merkle-Damgård style hashing with $\mathcal{F}$ a compression function (see Section 8).

The combination of a *tree hashing mode* $\mathcal{T}$ and an inner hash function $\mathcal{F}$ defines a hash function $\mathcal{T}[\mathcal{F}]$ that we call the *outer hash function*. In general, the outer hash function has variable input-length and arbitrary output-length.

A tree hashing mode and the resulting outer hash function may be parameterized. For example, one may put as parameter the height of the tree or the degree of the nodes (see, e.g., Section 7.2). So in general, a tree hashing function takes as input a message $M$ and the parameter values $A$ for a set of parameters that are specific to the tree hashing mode.

### 2.1 Hashing as a two-step process

The tree hashing mode specifies for any given combination of message length and parameter values the number of calls to $\mathcal{F}$ and how the inputs in these calls must be constructed from the message and the output of previous calls to $\mathcal{F}$. For a given input $(M, A)$, the result is the hash $h = \mathcal{T}[\mathcal{F}](M, A)$.

We express tree hashing as a two-step process:

**Template construction** The mode of use $\mathcal{T}$ generates a so-called *tree template* $Z$ that only depends on the length $|M|$ of the message and on the parameters $A$. We write $Z = \mathcal{T}(|M|, A)$. The tree template consists of a number of *virtual* strings called *node templates*. Each node template specifies for a call to $\mathcal{F}$ how the input must be constructed from message bits and from the output of previous calls to $\mathcal{F}$ (see Section 2.3).

**Template execution** The tree template $Z$ is *executed* by a generic template interpreter $\mathcal{Z}$ for a specific message $M$ and a particular $\mathcal{F}$ to obtain the output $h = \mathcal{T}[\mathcal{F}](M, A)$. The interpreter produces an intermediate result called a *tree instance* $S$ that is a tree consisting of *node instances*. Each node instance is a bitstring that is constructed according to the corresponding node template and presented to $\mathcal{F}$. We express this as $S = \mathcal{Z}[\mathcal{F}](M, Z)$. The hash result is finally obtained by $h = \mathcal{F}(S_*)$, where $S_*$ is a particular node of $S$, called the final node (see Section 2.2).

Hence, $h = \mathcal{T}[\mathcal{F}](M, A)$ is a shortcut notation to denote first $Z = \mathcal{T}(|M|, A)$ then $S = \mathcal{Z}[\mathcal{F}](M, Z)$ and finally $h = \mathcal{F}(S_*)$. See Figure 1 below for a toy example.

We now define the tree template set of a mode.

**Definition 1.** *The* tree template set *of a mode $\mathcal{T}$, denoted by $\mathcal{Z}_\mathcal{T}$, is the (possibly infinite) set of all tree templates that can be generated by $\mathcal{T}$. Formally:*

$$\mathcal{Z}_\mathcal{T} = \{Z \mid \exists(\ell, A) \text{ such that } Z = \mathcal{T}(\ell, A)\}.$$

In this paper, we only consider tree hashing modes that can be described in this way. However, this is without loss of generality. While we split the function's input in the parameters $A$ and the message content $M$, this is only a convention. If the tree template has to depend on the value of bits in $M$, and not only on its length, the parameters $A$ can be extended so as to contain a copy of such message bits. In other words, the definition of the parameters $A$ is just a way to cut the set of possible tree templates into equivalence classes identified by $(|M|, A)$. As far as we know, all hashing modes of use proposed in literature allow a straightforward identification of the parameters that influence the tree structure.

### 2.2 The tree structure

The node templates of a tree template $Z$ are denoted by $Z_\alpha$, where $\alpha$ denotes its index. Similarly, node instances are denoted by $S_\alpha$. As such, the nodes of tree templates and tree instances form a directed acyclic graph and hence make a tree.

We now introduce some terminology and concepts related to the tree topology. These are valid both for templates and instances, and we simply say "node" and "tree".

- A node may have a unique *parent node*. We denote the index of the parent of the node with index $\alpha$ by $\text{parent}(\alpha)$. (We assume that the node indexes $\alpha$ faithfully encode the tree structure, so that the function parent can work alone on the index given as input). In a tree, all nodes have a parent except one; we call this the *final node* and use the index $*$ to denote it. By contrast, we call the other nodes *inner nodes*.
- We say the node with index $\alpha$ is a *child* of the node with index $\text{parent}(\alpha)$. A node may have zero, one or more children. We call the number of children of a node its *degree* and a node without children a *leaf* node.
- We say that a node $Z_\alpha$ is an *ancestor* of a node $Z_\beta$ if $\alpha = \text{parent}(\beta)$ or if $Z_\alpha$ is an ancestor of the parent of $Z_\beta$. In other words, $Z_\alpha$ is a parent of $Z_\beta$ if there exists a sequence of indices $\alpha_0, \alpha_1, \alpha_{d-1}$ such that $\alpha = \alpha_0$, $\alpha_{i-1} = \text{parent}(\alpha_i)$ and $\alpha_{d-1} = \text{parent}(\beta)$. We say $Z_\beta$ is a *descendant* of $Z_\alpha$ and $d$ is the *distance* between $Z_\alpha$ and $Z_\beta$. Clearly, the final node has no ancestors and a leaf node has no descendants.
- Every node $Z_\alpha$ is a descendant of the final node and the distance between the two is called the *height* of $\alpha$. The final node has by convention height 0. The height of a tree is the maximum height over all its nodes.

– We denote the *restriction* of a tree $Z$ to a set of indices $J$ as the subset of its nodes with indices in $J$ and denote it as $Z_J$. The restriction is a *subtree* if it contains a node of which all other nodes in the restriction are descendant. We call a subtree a *final subtree* if it contains the final node. We call a subtree a *leaf subtree* if for each node it contains, it also contains all its descendants in $Z$. Note that a leaf subtree is fully determined by the index of the single node that is the ancestor of all the nodes it contains. We call a subtree a *proper* subtree of a tree if it does not contain all its nodes.

### 2.3 Structure of node templates

A node template $Z_\alpha$ is a sequence of *template bits* $Z_\alpha[x]$, $0 \leq x < |Z_\alpha|$, and instructs the forming of a bitstring called the node instance $S_\alpha$ in the following way. Each template bit has a type and the following attributes, depending on its type (and purpose):

**Frame bits** Represent bits fully determined by $A$ and $|M|$ and covers padding, IV values and coding of parameter value $A$. A frame bit only has a single attribute: its binary *value*. Upon execution, the template interpreter $\mathcal{Z}$ assigns the value of frame bit $Z_\alpha[x]$ to bit $S_\alpha[x]$.

**Message pointer bits** Represent bits to be taken from the message. A message pointer bit has a single attribute: its *position*. The position is an integer in the range $[0, |M| - 1]$ and points to a bit position in a message string $M$. Upon execution, $\mathcal{Z}$ assigns the message bit $M[y]$ to $S_\alpha[x]$, where $y$ is the position attribute of $Z_\alpha[x]$.

**Chaining pointer bits** Represent bits to be taken from the output of a previous call to $\mathcal{F}$. Chaining pointer bit have two attributes: a child index and a *position*. The child index $\beta$ identifies a node that is the child of this node and the position is an integer that points to a bit position in the output of $\mathcal{F}$. Upon execution, $\mathcal{Z}$ assigns chaining bit $\mathcal{F}(S_\beta)[y]$ to $S_\alpha[x]$, with $\beta$ the child index attribute of chaining pointer bit $Z_\alpha[x]$ and $y$ is its position attribute (A *chaining value* is the sequence of all chaining bits coming from the same child.).

Execution of a tree template for a specific message $M$ and function $\mathcal{F}$ now just consists of executing its node templates, where each template node $Z_\alpha$ is executed only after its children are processed. This results in a tree instance $S$ with nodes $S_\alpha$.

We illustrate the process in Figure 1 with a toy example taking a message of 28 bits and with 3-bit chaining values. In the nodes of the tree template, the dark gray blocks contain message pointer bits, the light gray chaining pointer bits and the white contain frame bits. The tree template is independent of the bits of $M$ and of $\mathcal{F}$. The tree instance is obtained by filling in the message bits and computing the chaining values by applying $\mathcal{F}$ to the nodes left-to-right.

We can define compliance of instances with templates.

**Definition 2.** *A tree instance $S$ is* compliant *with a tree template $Z$ iff it has the same tree topology, the corresponding nodes have the same length and the values of the frame bits in $Z$ match the corresponding bits in $S$. A tree instance $S$ is* final-subtree-compliant *with a tree template $Z$ iff the latter has a proper final subtree with which $S$ is compliant.*

In these definitions, we assume that the mode does not generate templates where a given message pointer bit or chaining pointer bit occurs twice. The case of repeating message pointer bits or chaining pointer bits could be covered but would complicate the definitions of compliance putting a burden on the readability of the paper.

We can define compliance of instances with modes.

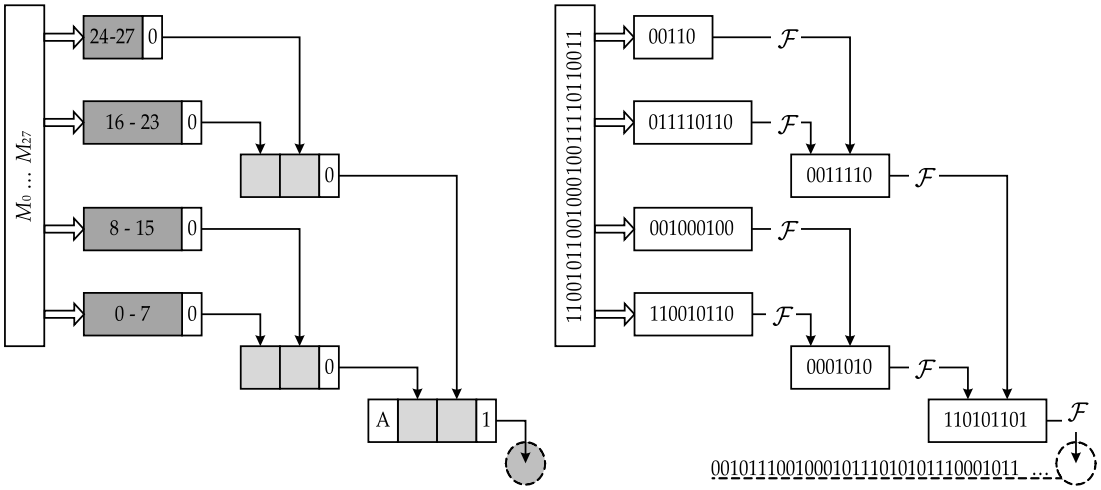


**Fig. 1.** Toy example tree template (left) and instance (right)

**Definition 3.** *A tree instance $S$ is* compliant with a mode $\mathcal{T}$ *iff $\mathcal{Z}_\mathcal{T}$ contains a tree template it is compliant with. A tree instance $S$ is* final-subtree-compliant with a mode $\mathcal{T}$ *iff $\mathcal{Z}_\mathcal{T}$ contains a tree template it is final-subtree-compliant with.*

## 3 The birthday bound and the size of chaining values

In this section, we show that collisions in chaining values result in behavior not observed in a random oracle and hence impose an upper bound on the strength of the tree hashing mode.

Let us try to produce a collision in the output of $\mathcal{T}[\mathcal{F}]$. Consider two inputs $(M, A)$ and $(M', A)$ with messages of equal length. As $|M| = |M'|$ they will have the same tree templates $Z = Z' = \mathcal{T}(|M|, A)$. For some fixed index $\alpha$, we construct pairs of messages that differ only in bits that are mapped to $Z_\alpha$ and its descendants (e.g., if $Z_\alpha$ is a leaf node, they only differ in a single node). This difference can only propagate to the final node via the chaining bits with child index $\alpha$ in its parent node. Let the number of these chaining bits be denoted by $n_\alpha$. For the two messages, these chaining bits will consist of a selection of output bits from $\mathcal{F}(S_\alpha)$ and $\mathcal{F}(S'_\alpha)$ respectively. Hence, a collision in the output of $\mathcal{F}$ restricted to these $n_\alpha$ bits implies an output collision in $\mathcal{T}[\mathcal{F}]$.

Assuming that $\mathcal{F}$ behaves like a random oracle, the success probability of having a collision in its output restricted to $n$ bits after trying $N$ inputs is

$$1 - \prod_{i=0}^{N-1} \left(1 - \frac{i}{2^n}\right) \approx 1 - \exp\left(-\frac{N(N-1)}{2^{n+1}}\right).$$

If $N < 2^{-n/2}$ this is upper bounded by:

$$\frac{N(N-1)}{2^{n+1}}.$$

This reasoning is independent of the value of index $\alpha$, so an upper bound to this success probability imposes a lower bound on the length of the shortest chaining value in the

tree. We can therefore logically expect a tree hashing mode to have the same length for all chaining values.

Our definition of templates allows for composing chaining values using bits of arbitrary positions of the output of $\mathcal{F}$. If we assume $\mathcal{F}$ generates its bits in a sequential fashion, the most efficient way is to take the first $n$ output bits of $\mathcal{F}$. We call the first $n$ bits of $\mathcal{F}(s)$ the *chaining value* of $s$ and denote the truncation of $\mathcal{F}$ to its first $n$ output bits by $\mathcal{F}_n$. Hence, in the following we will assume that $\mathcal{F}_n$ is used for computing the chaining values.

This birthday bound limits the achievable security one can expect from such a hashing mode. Theorem 1, proven in Section 6.2, actually achieves this bound and is thus as tight as theoretically possible.

## 4 Sufficient conditions for sound tree hashing

In this section, we formulate three conditions that a tree hashing mode $\mathcal{T}$ should satisfy. In Section 6 we will prove that the strength of a tree hashing mode that satisfies these three conditions coincides with the birthday bound. The first two conditions stem from the ability to generate collisions. The last condition prevents a generalization of length extension.

We start by defining the concept of inner collisions.

**Definition 4.** *An* inner collision *in $\mathcal{T}[\mathcal{F}]$ is a pair of inputs $(M, A)$ and $(M', A')$ such that their corresponding tree instances are different: $S \neq S'$ but have equal final-node instances $S_* = S'_*$.*

Clearly, an inner collision implies that $\mathcal{T}[\mathcal{F}](M, A) = \mathcal{T}[\mathcal{F}](M', A')$.

A collision of $\mathcal{F}_n$ can be used to generate an inner collision in $\mathcal{T}[\mathcal{F}]$. On the contrary, an inner collision does not necessarily imply an output collision of $\mathcal{F}_n$. For instance, let us try to produce an inner collision without a collision in $\mathcal{F}_n$. Consider two inputs $(M, A)$ and $(M', A')$ leading to tree templates $Z$ and $Z'$. We choose the values of $(|M|, A)$ and $(|M'|, A')$ in such a way that for all nodes $Z_\alpha$ we have $Z_\alpha = Z'_\alpha$ except for a particular node $Z_\beta$ and its descendants. Node $Z_\beta$ has one descendant $Z_\gamma$ and $Z'_\beta$ is a leaf node. Additionally, $Z_\beta$ and $Z'_\beta$ have the same length and in the positions where there are chaining pointer bits in $Z_\beta$, there are message pointer bits in $Z'_\beta$. For a given $M$, we can now compute all node instances; this includes the chaining bits in $S_\beta$ by instantiating its descendant $S_\gamma$ and evaluating $\mathcal{F}_n(S_\gamma)$. We can then construct $M'$ such that $S'_\beta = S_\beta$ and $S'_\alpha = S_\alpha$ for all other nodes in $Z'$. As $S$ has one more node than $S'$, the tree instances are not equal, and hence, we have an inner collision. This is illustrated with a simple example in Figure 4, Appendix B.

In this case, the inner collision is possible because the node templates $Z_\beta$ and $Z'_\beta$ are different. A simple way to avoid this situation is mandating that $\mathcal{T}$ is *tree-decodable*.

**Definition 5.** *A mode of use $\mathcal{T}$ is* tree-decodable *if there are no tree instances that are both compliant and final-subtree-compliant with that mode, and there exists a deterministic algorithm $A_{decode}$ that, given a tree instance $S$ with index set $J$, has the following behaviour:*

- *If $S$ is compliant with $\mathcal{T}$, $A_{decode}$ returns a flag* "compliant".
- *Else if $S$ is final-subtree-compliant with $\mathcal{T}$, $A_{decode}$ returns a flag* "final-subtree-compliant", *a node index $\beta \notin J$ such that* $\mathrm{parent}(\beta) \in J$ *and the list of positions in $S_{\mathrm{parent}(\beta)}$ of the corresponding chaining pointer bits 0 to $n-1$. We call the index $\beta$ an* expanding index *of $S$.*
- *Else $A_{decode}$ returns a flag* "incompliant".

*The running time of $A_{decode}$ shall be $O(m)$ with $m$ total number of bits in $S$.*

Note that $A_{\text{decode}}$ can be specific to the mode but can only use the information contained in the tree instance. Also, this definition includes the case where $A_{\text{decode}}$ can identify the chaining values and their attributes in a node from the sole information in that node instance, or the case where it does so from information in that node instance and all its ancestors.

We can now prove the following lemma, leading to our first condition.

**Lemma 1.** *When $\mathcal{T}$ is tree-decodable and $\mathcal{T}$ uses $\mathcal{F}_n$ to compute chaining values, an inner collision in $\mathcal{T}[\mathcal{F}]$ implies an output collision in $\mathcal{F}_n$.*

*Proof.* Let $S \neq S'$ produce an inner collision. Now, let $J$ define a final subtree $S_J$ and a final subtree $S'_J$ such that $S_J = S'_J$ and they have an expanding index $\beta$ with $S_\beta \neq S'_\beta$. We have that $S_{\text{parent}(\beta)} = S'_{\text{parent}(\beta)}$ and the chaining values with child index $\beta$ are fully determined by $S_J$. It follows that $\mathcal{F}_n(S_\beta) = \mathcal{F}_n(S'_\beta)$ and hence we have an output collision in $\mathcal{F}_n$.

We must now prove that there exists a set $J$ such that $S_J = S'_J$ and it has an expanding index $\beta$ such that $S_\beta \neq S'_\beta$. We do this in a recursive way. We have per definition $S_* = S'_*$ and hence we can take initially $J = \{*\}$, clearly defining a final subtree.

We can now repeat the following procedure until a set $J$ is found that satisfies the conditions above. If $J$ defines a final subtree, tree-decodability guarantees that there exists an expanding index $\beta$. If we can find an expanding index $\beta$ such that $S_\beta \neq S'_\beta$ we have found our $J$. Otherwise, we expand $J$ by adding $\beta$: $J \leftarrow J \cup \{\beta\}$. Clearly, this $J$ again defines a final subtree with $S_J = S'_J$. This process can only stop in three ways. First, for the current set $J$ an expanding index $\beta$ is found with $S_\beta \neq S'_\beta$. Second, $J$ covers all node indices of either $S$ or $S'$ but not in both. In this case, one tree is a proper final subtree of the other, which contradicts our assumption of tree-decodability. Third, $J$ covers all node indices of both $S$ or $S'$. This implies $S = S'$, which contradicts our initial assumption $S \neq S'$.

□

**Condition 1** *$\mathcal{T}$ is tree-decodable.*

Naturally, we can have an output collision in $\mathcal{T}[\mathcal{F}]$ without an inner collision if there are message bits that are not mapped to any template node or if two template trees resulting from two different messages of the same length and different parameters are equal in all frame bits and chaining pointer bits, but not in message pointer bits. For that reason, we introduce the concept of *message-completeness*.

**Definition 6.** *A mode of use $\mathcal{T}$ is* message-complete *if for all combinations $(|M|, A)$ the resulting tree template contains all $|M|$message pointer bits at least once and there is a deterministic algorithm $A_{message}$ that, given a tree instance $S$ compliant with $\mathcal{T}$, provides the list of positions in $S$ of message pointer bits 0 to $|M| - 1$. The running time of $A_{message}$ shall be $O(m)$ with $m$ the total number of bits in $S$.*

Message-completeness implies that the message can be fully reconstructed from the tree instance.

**Condition 2** *$\mathcal{T}$ is message-complete.*

The third condition is related to a property that generalizes length extension to tree hashing. Assume we have two trees $S$ and $S'$ corresponding with inputs $(M, A)$ and $(M', A')$ with a particular property. First of all, $S'$ has the same topology as a leaf subtree $S_J$ of $S$ containing a node $S_\alpha$ and all its descendants. Second, there is a one-to-one mapping $\psi$ between the indices of $S'$ and the elements of $J$ that preserves the parent-child relation: $\text{parent}(\psi(\beta)) = \psi(\text{parent}(\beta))$ and for which $\psi(\alpha) = *$. Finally, we have $S_{\psi(\beta)} = S'_\beta$.

As $S_\alpha = S'_*$, we have $\mathcal{T}[\mathcal{F}](M', A') = \mathcal{F}(S'_*) = \mathcal{F}(S_\alpha)$. Hence, one can compute $\mathcal{T}[\mathcal{F}](M, A)$ without knowing the message bits of $M$ mapped to the subtree $S_J$ and just knowing $\mathcal{T}[\mathcal{F}](M', A')$. This is illustrated with a simple example in Figure 5, Appendix B.

This property is not present in a random oracle and could be easily checked by a distinguisher. It can be avoided in several ways, such as fixing the topology of the trees. However, imposing the condition that there is a way to distinguish final nodes from inner nodes is a way to solve this problem that allows more flexibility. This can be expressed as follows:

**Definition 7.** *A mode of use $\mathcal{T}$ is* final-node separable *if any node instance that is an inner node in a tree instance compliant with $\mathcal{T}$ is, as a single-node tree instance, neither compliant nor final-subtree-compliant with that mode.*

For a mode that is both final-node-separable and tree-decodable, $A_{\text{decode}}$ will return "incompliant" to any node instance that is an inner node in a compliant tree instance.

Now we can formulate our third condition:

**Condition 3** *$\mathcal{T}$ is final-node separable.*

A simple way to implement final-node separability is by domain separation between final and inner nodes, e.g., by starting (or ending) each node with a frame bit indicating whether it is a final or inner node. Note that if $\mathcal{F}$ is a random oracle, this is equivalent to saying that the function applied to the final node is a different one than the function applied to the inner nodes and hence is similar to an output transformation (with arbitrary output-length though).

Verifying whether a given hash mode satisfies these conditions is typically straightforward. For some concrete examples, we refer to Sections 7.2, 7.5 and 8.

### 4.1 Comparison with tree based modes of Dodis et al.

The five "required properties of Mode of Operation" listed in [15] correspond to a large extent with our three conditions, but not quite.

First, the condition *unique parsing* is similar to our condition tree-decodability. However, according to the definition in [15], unique parsing of any node instance that may occur in a tree instance implies that it must be possible to identify the chaining pointer bits, frame bits and message pointer bits with just access to the node instance itself. This is a restricted case of our tree-decodability. While for unique parsing the node instance coding is either fixed or must contain some frame bits fully specifying its composition, in tree-decodability only the chaining pointer bits of a single expanding index $\beta$ must be found. For doing this, information may be derived from the part of the tree that has already been decoded. In general, our condition requires less frame bits to be inserted and thus allows for a better trade-off between flexibility and efficiency.

Second, the property *root predicate* fully coincides with our condition on final-node separability.

Third, in our conditions, there is no equivalent for the so-called *straight-line program structure* property. Rather, it corresponds to our definition of a tree hashing mode in Section 2. At first sight, the two definitions of tree hashing modes are rather different. This is, however, just a difference in presentation. In this paper, we clearly distinguish two parts in the input: a message part $M$ of which only the length has an impact on the tree template, and a parameter part that determines the tree template. In [15] they are presented as a single input called "message" and no distinction is made between the two types of input. As discussed in Section 2, our definition allows a large amount of flexibility and can actually implement any "straight-line program structure".

Fourth, there is no equivalent either for the property *final output processing*. It says that the output of applying the inner hash to the final node undergoes a function $\zeta$ that must be an "efficiently computable, regular function" for which "the set of all preimages $\zeta^{-1}(h)$ must be efficiently sampleable given $h$". An example of such a function is truncation (chopping) of the output and it seems that this function is introduced as a generalization of truncation to accommodate an outer hash function output-length different from the inner hash function output-length. In our approach, we study the differentiating advantage from a random oracle with variable output-length and the need for such a complication does not appear. Clearly, truncating the output does not harm the differentiating advantage as it gives an adversary less information. Moreover, the application of any balanced function (as $\zeta$ above) to the output also preserves it.

Finally, the property *message reconstruction* maps to message-completeness.

### 4.2 Property preserving aspects

Independently of the bounds proved below, some properties hold when the three conditions of Section 4 are satisfied.

First, given message-completeness, producing a collision in the first $m$ output bits of $\mathcal{T}[\mathcal{F}](M, A)$ implies either an $m$-bit collision in $\mathcal{F}(S_*)$ or an inner collision. In the latter case, Lemma 1 implies that one produces a collision in $\mathcal{F}_n$. Therefore, the tree hashing mode preserves the collision resistance of the inner hash function.

Then, a similar idea applies to the preimage attack. Given an $m$-bit output value $s$, an adversary who can find an input $(M, A)$ such that the first $m$ bits of $\mathcal{T}[\mathcal{F}](M, A)$ are $s$ can reconstruct the tree node instances and compute the final node $S_*$. She is thus able to find a preimage on the inner hash function $\mathcal{F}$ (For second preimage, this reasoning does not apply as a second preimage for $\mathcal{T}[\mathcal{F}]$ can be constructed by finding a second preimage of $\mathcal{F}$ for any inner node, and hence, it is not a second preimage of some designated output).

## 5 Adoption of the indifferentiability framework

In this section we specify and motivate the distinguisher's setting, how we compute the cost of queries and the resulting definition of indifferentiability of a tree hashing mode.

### 5.1 The distinguisher's setting

We study the differentiating advantage of a tree hashing mode $\mathcal{T}$, calling an ideal inner hash function $\mathcal{F}$, from an ideal outer hash function $\mathcal{G}$. This leads to the setting illustrated

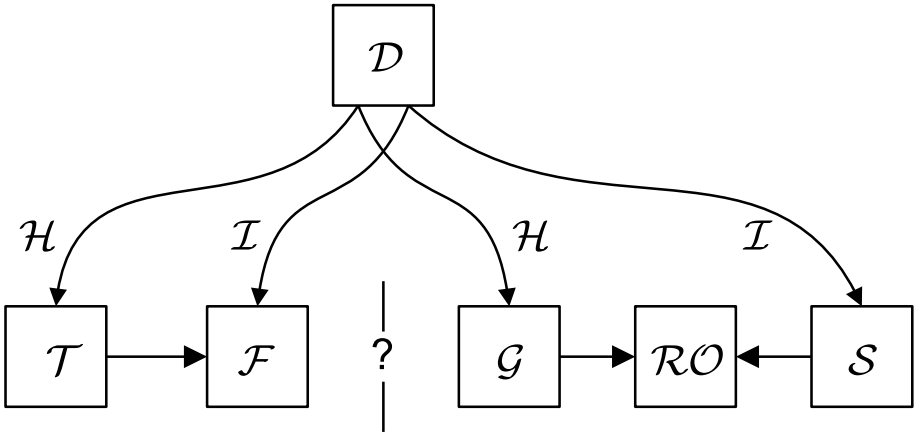


**Fig. 2.** The differentiability setup

in Figure 2. The system at the left is $\mathcal{T}[\mathcal{F}]$ and $\mathcal{F}$, and the adversary can make queries to both subsystems separately, where the former in turn calls the latter to construct its responses. The distinguisher has the following interfaces to this system:

- $\mathcal{H}$ taking as input $(M, A, \ell)$ with $M$ a binary string, i.e., $M \in \mathbf{Z}_2^*$, $A$ the value of the mode parameters and $\ell$ the requested output-length, and returning a binary string $y \in \mathbf{Z}_2^\ell$;
- $\mathcal{I}$ taking as input $(s, \ell)$ with $s$ a binary string $s \in \mathbf{Z}_2^*$ and $\ell$ the requested output-length, and returning a binary string $t \in \mathbf{Z}_2^\ell$.

When queried at the interface $\mathcal{H}$ with a query $(M, A, \ell)$, the left system returns $y = \mathcal{T}[\mathcal{F}](M, A)$ truncated to $\ell$ bits. When queried at the interface $\mathcal{I}$ with a query $(s, \ell)$, it returns $t = \mathcal{F}(s)$ truncated to $\ell$ bits.

The system at the right consists of an ideal hash function $\mathcal{G}$ implementing the interface $\mathcal{H}$ and a simulator $\mathcal{S}$ implementing the interface $\mathcal{I}$. We define the ideal hash function as $\mathcal{G}[\mathcal{RO}]$, where $\mathcal{RO}$ is a random oracle. Upon receiving a query $(M, A, \ell)$, $\mathcal{G}$ constructs the tree template $Z = \mathcal{T}(|M|, A)$ and encodes it together with the message $M$ in an injective way, denoted by $\mathrm{enc}(M, Z, \text{flag "success"})$ (the last argument is for domain separation as detailed in Section 6.1). It queries $\mathcal{RO}$ with the latter as argument returns its response to the distinguisher. $\mathcal{G}[\mathcal{RO}]$ returns independent outputs for different messages. It also returns independent outputs for equal messages but parameter values leading to different tree templates. However, for equal messages and different parameter values leading to equal tree templates it returns equal outputs.

The output of $\mathcal{S}$ should look *consistent* with what the distinguisher can obtain from the ideal hash function $\mathcal{G}[\mathcal{RO}]$, like if $\mathcal{S}$ was $\mathcal{F}$ and $\mathcal{G}[\mathcal{RO}]$ was $\mathcal{T}[\mathcal{F}]$. To achieve that, the simulator can query $\mathcal{RO}$, denoted by $\mathcal{S}[\mathcal{RO}]$. Note that the simulator does not see the distinguisher's queries to $\mathcal{G}[\mathcal{RO}]$. Summarizing, $\mathcal{S}$ implements the interface $\mathcal{I}$ and when queried with $(s, \ell)$, it responds with $\mathcal{S}[\mathcal{RO}](s, \ell)$.

Indifferentiability of $\mathcal{T}[\mathcal{F}]$ from the ideal function $\mathcal{G}[\mathcal{RO}]$ is now satisfied if there exists a simulator $\mathcal{S}$ such that no distinguisher can tell the two systems apart with non-negligible probability, based on their responses to queries it may send.

In this setting, the distinguisher can send queries $Q$ to both interfaces. Let $\mathcal{X}$ be either $(\mathcal{T}[\mathcal{F}], \mathcal{F})$ or $(\mathcal{G}[\mathcal{RO}], \mathcal{S}[\mathcal{RO}])$. The sequence of queries $Q$ to $\mathcal{X}$ consists of a sequence of queries to the interface $\mathcal{H}$, denoted $Q_{\mathcal{H}}$ and a sequence of queries to the interface $\mathcal{I}$, denoted $Q_{\mathcal{I}}$. $Q_{\mathcal{H}}$ is a sequence of triplets $Q_{\mathcal{H},i} = (M_i, A_i, \ell_i)$, while $Q_{\mathcal{I}}$ is a sequence of couples $Q_{\mathcal{I},i} = (s_i, \ell_i)$.

We can now define $\mathcal{T}$-consistency.

**Definition 8.** *For a given set of queries $Q$ and their responses $\mathcal{X}(Q)$,* $\mathcal{T}$-consistency *is the property that the responses from the $\mathcal{H}$ interface are equal to those that one would obtain by applying the tree hashing mode $\mathcal{T}$ to the responses from the $\mathcal{I}$ interface (when the queries $Q_{\mathcal{I}}$ suffice to perform this calculation), i.e., that $\mathcal{X}(Q_{\mathcal{H}}) = \mathcal{T}[\mathcal{X}(Q_{\mathcal{I}})](Q_{\mathcal{H}})$.*

Note that $\mathcal{T}$-consistency is per definition always satisfied by the system on the left but not necessarily by the system on the right.

### 5.2 The cost of queries

The differentiability bounds in [13] are expressed as a function of the total number $q$ of queries and their maximum input-lengths. In [7] a bound is expressed as a function of a *cost*, that is proportional to the total length of the queries and their responses. In this paper, we use a third approach: we quantify the contribution of the queries to $\mathcal{H}$ and to $\mathcal{I}$ using a common unit, which is a query to the interface $\mathcal{I}$. This is motivated by the fact that queries to $\mathcal{H}$ and queries to $\mathcal{I}$ behave very differently when addressing $(\mathcal{T}[\mathcal{F}], \mathcal{F})$: a query to $\mathcal{H}$ may require many calls to $\mathcal{F}$, while a query to $\mathcal{I}$, when applied to $\mathcal{F}$, requires only a single call.

The *cost* $q$ of queries to a system $\mathcal{X}$ is the total number of calls to $\mathcal{F}$ it would yield if $\mathcal{X} = (\mathcal{T}[\mathcal{F}], \mathcal{F})$, either directly due to queries $Q_{\mathcal{I}}$, or indirectly via queries $Q_{\mathcal{H}}$ to $\mathcal{T}[\mathcal{F}]$. The cost of a sequence of queries is fully determined by their number and their input.

- Each query $Q_{\mathcal{I},i}$ to $\mathcal{I}$ contributes 1 to the cost.
- Each query $Q_{\mathcal{H},i} = (M_i, A_i, \ell_i)$ to $\mathcal{H}$ costs a number $f_{\mathcal{T}}(|M_i|, A_i)$, depending on the tree hashing mode $\mathcal{T}$, the mode parameters $A_i$ and the length of the input message $|M_i|$. The function $f_{\mathcal{T}}(|M|, A)$ counts the number of calls $\mathcal{T}[F]$ needs to make to $\mathcal{F}$ from the template produced for parameters $A$ and message length $|M|$. Note that $f_{\mathcal{T}}(|M|, A)$ is also the number of nodes produced by $\mathcal{T}(|M|, A)$.

In addition, we define the cost not to take into account duplicate queries. Two queries $Q_{\mathcal{I},i} = (s_i, \ell_i)$ and $Q_{\mathcal{I},j} = (s_j, \ell_j)$ with $s_i = s_j$ are counted as one, and cost as much as a single query $(s_i, \max(\ell_i, \ell_j))$. Similarly, two queries $Q_{\mathcal{H},i} = (M_i, A_i, \ell_i)$ and $Q_{\mathcal{H},j} = (M_j, A_j, \ell_j)$ with $M_i = M_j$ and $A_i = A_j$ are counted as one, and cost as much as a single query $(M_i, A_i, \max(\ell_i, \ell_j))$. Note that this is only an a posteriori accounting convention rather than a suggestion to replace overlapping queries by a single one. This convention only benefits to the adversary and is thus on the safe side regarding security; see also Appendix C for some additional discussion.

### 5.3 Definition

We can now adapt the definition as given in [13] to our setting.

**Definition 9 ([13]).** *A tree hashing mode $\mathcal{T}$ with oracle access to an ideal hash function $\mathcal{F}$ is said to be $(t_D, t_S, q, \epsilon)$-indifferentiable from an ideal hash function $\mathcal{G}[\mathcal{RO}]$ if there exists a simulator $\mathcal{S}[\mathcal{RO}]$, such that for any distinguisher $\mathcal{D}$ it holds that:*

$$|\Pr[\mathcal{D}[\mathcal{T}[\mathcal{F}], \mathcal{F}] = 1] - \Pr[\mathcal{D}[\mathcal{G}[\mathcal{RO}], \mathcal{S}[\mathcal{RO}]] = 1]| < \epsilon.$$

*The simulator has oracle access to $\mathcal{RO}$ and runs in time at most $t_S$. The distinguisher runs in time at most $t_D$ and has a cost of at most $q$. The value $\epsilon$ is an upper bound on the differentiating advantage. We speak about indifferentiability when the differentiating advantage is a negligible function of the security parameter $n$.*

## 6 Upper bound on the differentiating advantage

In this section, we always assume that the conditions presented in Section 4 are fulfilled by the tree hashing mode $\mathcal{T}$. We first describe the simulator and its general goal. We then prove an upper bound on the differentiating advantage by means of a series of lemmas and a final theorem. We follow a proof technique very similar to the one we introduced in [7].

### 6.1 The simulator and $\mathcal{T}$-decoding

The adversary can verify $\mathcal{T}$-consistency in the following way. She takes any couple $(M, A)$ and the constructs from $(|M|, A)$ the template $Z = \mathcal{T}(|M|, A)$. Subsequently, s executes the template $Z$. In this process, she produces all node instances of the tree instance $S = \mathcal{Z}[\mathcal{F}](M, Z)$ and makes for each of them a query to $\mathcal{I}$ requesting $n$ bits. For the final node $S_*$ of $S$, she queries $\mathcal{I}$ with $(S_*, \ell)$. She then makes the query $(M, A, \ell)$ to $\mathcal{H}$ and compares the responses. For $\mathcal{T}$-consistency, these must be the same. Note that in case the distinguisher is addressing the system $(\mathcal{G}[\mathcal{RO}], \mathcal{S}[\mathcal{RO}])$, the simulator has received queries for all node instances in $S$. This reasoning remains valid if the distinguisher queries $\mathcal{H}$ before or between queries to $\mathcal{I}$.

Our simulator has a set $T$ in which it keeps for each inner-node query a couple with the node $s$ and the chaining value $c$ it returned. We say that a final-node instance $s$ is *message-bound* if the set $T$ allows reconstructing the corresponding input message $M$ and tree template $Z$, denoted by the couple $(M, Z)$. This is an important concept in our proof. Algorithm 1 attempts to extract the couple $(M, Z)$ from $T$ and a given final node, by first reconstructing the complete tree instance using the known pairs $(s, t)$ in $T$. If it succeeds in obtaining $(s, t)$ using $T$, Conditions 1 and 2 imply that it can then reconstruct $(M, Z)$ from this tree instance. In this case it will return a flag "success" and the couple $(M, Z)$. If it can only form a tree that is final-subtree-compliant with $\mathcal{T}$, it returns the flag "dead end" and the chaining value $c$ whose preimage could not be found in the set $T$. If at some point, the tree it is generating becomes incompliant with $\mathcal{T}$, it returns a flag "incompliant coding". Note that $\mathcal{T}$-decoding as specified in Algorithm 1 is a deterministic algorithm as the algorithms $A_{\text{decode}}$ and $A_{\text{message}}$ are deterministic.

In our $\mathcal{T}$-decoding algorithm, we make use of a *working* tree template. This is the same as a tree template, except that it can have a fourth type of bit called *undetermined*, denoted as $\square$. A string of $\ell$ undetermined bits is denoted as $\square^\ell$.

Algorithm 2 specifies the simulator $\mathcal{S}[\mathcal{RO}]$. It has two sets for the purpose of behaving deterministically and $\mathcal{T}$-consistently. First, as already explained, it keeps track of the

**Algorithm 1** $\mathcal{T}$-decoding

1: **input:** $s$ and set $T$
2: **output:** flag and pair $(M,Z)$ or chaining value $c$
3: Initialization: $J = \{*\}$, $S_* = s$ and $Z_* = \square^{|s|}$
4: **while** $A_{\text{decode}}(S_J)$ returns flag "final-subtree-compliant" **do**
5: Fill in the chaining pointer bits of the expanding index in $Z$
6: Let $c$ be the chaining value corresponding to expanding index $\beta$ extracted from $S_J$
7: **if** there is exactly one entry $(s', t) \in T$ with $t = c$ **then**
8: Let $J = J \cup \{\beta\}$, $S_\beta = s'$ and $Z_\beta = \square^{|s'|}$
9: **else**
10: **return** flag "dead end" and $c$
11: **end if**
12: **end while**
13: **if** $A_{\text{decode}}(S_J)$ returns flag "incompliant" **then**
14: **return** flag "incompliant coding"
15: **else**
16: Reconstruct the message pointer bits in $Z_J$ by calling $A_{\text{message}}(S_J)$
17: Determine the remaining undetermined bits (all frame bits) in $Z_J$ by copying them from $S_J$
18: Reconstruct $M$ from $S_J$ using the message pointer bits in $Z_J$.
19: **return** flag "success" and $(M, Z)$
20: **end if**

chaining values it generated for queries that may be inner nodes in a set $T$ of couples $(s, c)$ with $s \in \mathbf{Z}_2^*$ and $c \in \mathbf{Z}^n$. We denote the set of first members of $(s, c) \in T$ by $T_{\text{in}}$ and the set of second members by $T_{\text{out}}$. Second, it keeps track of all dead ends it has generated in a set $P \subseteq \mathbf{Z}_2^n$. Both $T$ and $P$ are initialized to the empty set.

Upon receipt of a query $(s, \ell)$, it subjects it to $\mathcal{T}$-decoding and subsequently generates its response and updates its sets $T$ and $P$ based on its outcome.

- If $\mathcal{T}$-decoding returns "incompliant coding" and there is not yet a couple in $T$ with $s$ as its first member, the simulator randomly generates a chaining value $c$ and adds the couple $(s, c)$ to $T$ for the purpose of acting deterministically. In generating $c$, it avoids collisions in $T_{\text{out}}$ and therefore line 7 of Algorithm 1 does not have to explicitly treat the case of multiple entries. Moreover, it also avoids values in $P$ to make sure that nodes that have returned a flag "dead end" upon their first $\mathcal{T}$-decoding will also do so for all later ones. It then returns as response the first $\ell$ bits of the chaining value $c$ of the pair $(s, c) \in T$ appended with the result of calling $\mathcal{RO}$ with $c$ as input for the purpose of acting in a deterministic way.
- If $\mathcal{T}$-decoding returns a flag "dead end," the simulator adds the chaining value $c$ to $P$ and calls $\mathcal{RO}$ with $s$ as input for the purpose of acting in a deterministic way.
- If $\mathcal{T}$-decoding is successful this means $s$ is message-bound. The simulator calls $\mathcal{RO}$ with the resulting couple $(M, Z)$ for guaranteeing $\mathcal{T}$-consistency.

In each of the cases, the simulator makes a call to the random oracle, but every time for a different purpose. The calls to the random oracle can be seen as calls to three different random oracles. This is realized by applying domain separation by means of the flag that is the last argument of the encoding function.

**Algorithm 2** The simulator $\mathcal{S}[\mathcal{RO}]$

1: **input:** $(s, \ell)$ (interface $\mathcal{I}$)
2: **output:** string in $\mathbf{Z}_2^\ell$
3: Call $\mathcal{T}$-decode$(s, T)$ resulting in flag and a result
4: **if** flag is "incompliant coding" **then**
5:  **if** $s \notin T_{\text{in}}$ **then**
6:  Select $c$ randomly from $\mathbf{Z}_2^n \setminus (P \cup T_{\text{out}})$
7:  Add $(s, c)$ to $T$
8:  **end if**
9:  Let $c$ be the chaining value corresponding to $s$ in $T$
10:  **return** the first $\ell$ bits of $(c||\mathcal{RO}(\text{enc}(s, \text{flag "incompliant coding"})))$
11: **else if** flag is "dead end" and result is $c$ **then**
12:  Set $P \leftarrow P \cup \{c\}$
13:  **return** the first $\ell$ bits of $\mathcal{RO}(\text{enc}(s, \text{flag "dead end"}))$
14: **else if** flag is "success" and result is $(M, Z)$ **then**
15:  **return** the first $\ell$ bits of $\mathcal{RO}(\text{enc}(M, Z, \text{flag "success"}))$
16: **end if**

### 6.2 The proof

We prove a bound on the differentiating advantage in a series of lemmas. Our simulator behaves deterministically and $\mathcal{T}$-consistently as long as $P \cup T_{\text{out}}$ is not equal to $\mathbf{Z}_2^n$. We call this saturation:

**Definition 10.** *The simulator is* saturated *when $P \cup T_{out} = \mathbf{Z}_2^n$ in Algorithm 2 .*

Note that for the simulator to be saturated it must have received at least $2^n$ queries.

**Lemma 2.** *The simulator acts as a deterministic function as long as it is not saturated.*

*Proof.* We must prove that when the simulator receives a query $(s, \ell)$, the response $t$ will be consistent to the response $t'$ of any previous query $(s, \ell')$. This implies that if $\ell < \ell'$, $t$ must be equal to $t'$ truncated to $\ell$ bits, if $\ell > \ell'$, the first $\ell$ bits of $t$ must be equal to $t'$ and if $\ell = \ell'$, $t$ must be equal to $t'$.

The simulator constructs the response depending on the outcome of $\mathcal{T}$-decoding. We will first show that $\mathcal{T}$-decode$(s, T)$ has the same result for all queries $(s, \ell)$ with common $s$. Or equivalently, the way the simulator adds entries $(s, c)$ cannot change the result of $\mathcal{T}$-decode$(s, T)$.

$\mathcal{T}$-decode$(s, T)$ is a deterministic algorithm, and the only step that depends on the set $T$ is the finding of an entry $(s', c)$ with $c$ the chaining value corresponding to the expanding index. Note that set $T$ only contains couples $(s, c)$ for which $\mathcal{T}$-decoding of $s$ has resulted in the flag "incompliant coding." It follows that adding entries to $T$ could only affect $\mathcal{T}$-decode$(s, T)$ in two cases:

- It would cause $T$ to have multiple entries $(s, c)$ with the same $c$-value where in a prior call to $\mathcal{T}$-decoding there was exactly one. The simulator avoids this by choosing the chaining values $c$ added to $T_{\text{out}}$ different from all chaining values already in $T_{\text{out}}$. (So the case that in line 7 of Algorithm 1 there are multiple entries cannot occur. Similarly, line 9 of Algorithm 2 unambiguously retrieves one value $c$.)
- It would cause $T$ to have an entry $(s', c)$ where in a prior call to $\mathcal{T}$-decoding there was none with second member $c$ (so a dead end). The simulator avoids this by choosing the chaining values $c$ added to $T_{\text{out}}$ different from all chaining values in the set $P$, containing all dead-end chaining values.

The simulator returns a response consisting of the first $\ell$ bits of a string $V$ that it constructs differently depending on the outcome of $\mathcal{T}$-decode$(s, T)$. It does this in a deterministic way, depending on the flag that $\mathcal{T}$-decode$(s, T)$ returns:

- "incompliant coding": $V$ consists of the second member of $(s, c)$ stored in $T$ followed by a random string fully determined by $s$. The simulator adds the couple $(s, c)$ to $T$ upon the first query with $s$ as first member.
- "dead end": $V$ is a random string fully determined by $s$.
- "success": $V$ consists of the response of $\mathcal{RO}$ to a query with the encoding of the couple $(M, Z)$ that is the result of the $\mathcal{T}$-decoding. We have shown above that adding entries to $T$ as done by the simulator cannot change the result of $\mathcal{T}$-decoding.

□

**Lemma 3.** *The responses of $(\mathcal{G}[\mathcal{RO}], \mathcal{S}[\mathcal{RO}])$ are $\mathcal{T}$-consistent as long as the simulator $\mathcal{S}$ is not saturated.*

*Proof.* First of all, from Lemma 2 it follows that the simulator will behave as a deterministic function for any query as long as it is not saturated.

According to Definition 8, $\mathcal{T}$-consistency implies that there is a set of queries to the simulator corresponding to the execution of a template $Z$ for a message $M$ and a query for the corresponding final-node instance $S_*$ and such that its response $\mathcal{S}[\mathcal{RO}](S_*)$ is different from $\mathcal{RO}(\text{enc}(M, Z, \text{flag "success"}))$.

Thanks to final-node separability (Condition 3) and tree-decodability (Condition 1), $\mathcal{T}$-decoding can distinguish between queries with message-bound final nodes $S_*$ and others. We will show that the simulator avoids giving $\mathcal{T}$-inconsistent responses in both cases.

In the case of queries with nodes that are not message-bound final nodes, this is immediate since there is no information in the $Q_{\mathcal{I}}$ queries and their responses to show inconsistency.

In the case of message-bound final nodes, final-node separability (Condition 3) guarantees that the inner-node queries will be recorded into the set $T$ and so $S_*$ is message-bound to $(M, Z)$. The $\mathcal{T}$-decoding of the final node will then lead to reconstruction of the corresponding tree instance $S$ thanks to tree-decodability (Condition 1) and the fact that $T_{\text{out}}$ has no collisions while it is not saturated. From message-completeness (Condition 2) it follows that the message $M$ can be reconstructed from the found tree instance $S$. The combination of tree-decodability and message-completeness implies that $\mathcal{T}$-decoding can successfully reconstruct the tree template $Z$. Finally, the simulator queries $\mathcal{RO}(\text{enc}(M, Z, \text{flag "success"}))$ to make its response $\mathcal{T}$-consistent per construction. □

**Lemma 4.** *As long as the simulator is not saturated, any sequence of queries $Q_{\mathcal{H}}$ can be converted to a sequence of queries $Q_{\mathcal{I}}$, where $Q_{\mathcal{I}}$ gives at least the same amount of information to the adversary and has no higher cost than that of $Q_{\mathcal{H}}$.*

*Proof.* For each query $Q_{\mathcal{H},i} = (M_i, A_i, \ell_i)$, we can produce the template from $A_i$ and $|M_i|$. This template determines exactly how the query $Q_{\mathcal{H},i}$ can be converted into a set $Q_{\mathcal{I}}$ of $f_{\mathcal{T}}(A_i, |M_i|)$ queries to interface $\mathcal{I}$. From the definition of the cost, it follows that the cost of $Q_{\mathcal{I}}$ cannot be higher than that of $Q_{\mathcal{H}}$; the cost can be lower if there are redundant queries in $Q_{\mathcal{I}}$. □

**Lemma 5.** *The advantage of an adversary in distinguishing between $\mathcal{F}$ and $\mathcal{S}[\mathcal{RO}]$ with the responses to a sequence of $q < 2^n$ queries $Q_{\mathcal{I}}$ is upper bounded by:*

$$\epsilon_n(q) = 1 - \prod_{i=0}^{q-1} \left(1 - \frac{i}{2^n}\right) .$$

*Proof.* The distinguishing advantage is defined as

$$\mathrm{Adv}(\mathcal{A}) = |\Pr[\mathcal{A}[\mathcal{F}] = 1] - \Pr[\mathcal{A}[\mathcal{S}[\mathcal{RO}]] = 1]|.$$

We provide an upper bound of the advantage by computing the total variation distance between the two statistical distributions.

Since $\mathcal{F}$ is a random oracle, the responses to $q$ different queries are independent and uniformly distributed over $\mathbf{Z}_n^2$. For the simulator $\mathcal{S}$, the distributions depend on the outcome of $\mathcal{T}$-decoding. The responses to queries for which $\mathcal{T}$-decoding returns "success" or "dead end" are uniformly and independently generated as they are the responses of calls to the random oracle with different inputs thanks to including the flags in the encoding for domain separation. The responses to queries for which $\mathcal{T}$-decoding returns "incompliant coding" avoid inner collisions in their first $n$ bits. The remaining bits are again uniformly and independently generated by yet other calls to the random oracle. Hence, we are only interested in the first $n$ output bits of the responses to queries for which $\mathcal{T}$-decoding returns "incompliant coding".

The simulator chooses them from the set $\mathbf{Z}_2^n \setminus (P \cup T_{\mathrm{out}})$. Each query can add at most one element $T_{\mathrm{out}}$. The response to the $i$-th query is chosen from at least $2^n - i + 1$ values. After $q$ queries, there are at least $(2^n)_{(q)}$ (where $a_{(n)}$ denotes $a!/(a-n)!$) possible responses, each with equal probability $1/(2^n)_{(q)}$. This gives $\mathrm{Adv}(\mathcal{A}) \leq 1 - \frac{(2^n)_{(q)}}{2^{nq}} = \epsilon_n(q)$. □

We have now all the ingredients to prove our main theorem.

**Theorem 1.** *A tree hashing mode $\mathcal{T}[\mathcal{F}]$ that uses $\mathcal{F}_n$ for the chaining values and satisfies Conditions 1, 2 and 3, is $(t_D, t_S, q, \epsilon)$-indifferentiable from an ideal hash function, for any $t_D$, $t_S = O(q^3 + q^2 m)$, $q < 2^n$ and any $\epsilon$ with $\epsilon > \epsilon_n(q)$, with $q$ the cost of queries as defined in Section 5.2 and $m$ the maximum size in bits of the trees processed by the simulator $\mathcal{S}$.*

*Proof.* We consider an adversary that is more powerful than required; the bound we prove is also valid for the actual adversary who cannot do better. For a given cost, the adversary can issue the queries $Q_{\mathcal{I}}$ and $Q_{\mathcal{H}}$ in any order she wishes. After she is done, we give her for free additional queries $Q'_{\mathcal{I}}$ derived from the queries $Q_{\mathcal{H}}$ as in Lemma 4 and their responses. Since the queries $Q'_{\mathcal{I}}$ are issued at the end of the process, they have no impact on the state of the simulator $\mathcal{S}$ when issuing the original queries $Q_{\mathcal{I}}$.

From Lemma 4, the queries $Q_{\mathcal{I}} \cup Q'_{\mathcal{I}}$ do not have a cost higher than that of $Q_{\mathcal{I}} \cup Q_{\mathcal{H}}$. Since $q < 2^n$, we are sure that $P \neq \mathbf{Z}_2^n$ in the simulator even after issuing the free extra queries $Q'_{\mathcal{I}}$. As a consequence, Lemma 2 and Lemma 3 guarantee $\mathcal{T}$-consistency of all the queries $Q_{\mathcal{I}} \cup Q_{\mathcal{H}} \cup Q'_{\mathcal{I}}$. This means that the responses to the queries $Q_{\mathcal{I}} \cup Q'_{\mathcal{I}}$ give the same information as those to $Q_{\mathcal{I}} \cup Q_{\mathcal{H}} \cup Q'_{\mathcal{I}}$. We can therefore concentrate on the distinguishing probability using only the queries $\overline{Q}_{\mathcal{I}} = Q_{\mathcal{I}} \cup Q'_{\mathcal{I}}$ and their response $\mathcal{X}(\overline{Q}_{\mathcal{I}})$.

For any fixed sequence of queries $\overline{Q}_{\mathcal{I}}$, we look at the problem of distinguishing the random variable $\mathcal{F}(\overline{Q}_{\mathcal{I}})$ from the random variable $\mathcal{S}[\mathcal{RO}](\overline{Q}_{\mathcal{I}})$. Lemma 5, upper bounds the advantage to $\epsilon_n(q)$.

For the complexity of the simulator, we have $t_S = O(q^3 + q^2 m)$. Namely, for each of the $q$ queries, the simulator performs $\mathcal{T}$-decoding. The latter builds a tree instance with at most $q$ nodes, and for each of these nodes, an entry must be found in the set $T$ containing at most $q$ entries. This accounts for $O(q^3)$. Additionally, $\mathcal{T}$-decoding executes $A_{\text{decode}}$ at most $q$ times and $A_{\text{message}}$ a single time. This accounts for $O(q^2 m)$. □

If $q$ is significantly smaller than $2^n$, we can use the approximation $1 - x \approx \mathrm{e}^{-x}$ for $x \ll 1$ to simplify the expression for $\epsilon_n(q)$:

$$\epsilon_n(q) \approx 1 - \mathrm{e}^{-\frac{q(q-1)}{2^{n+1}}} < \frac{q(q-1)}{2^{n+1}} \approx \frac{q^2}{2^{n+1}}.$$

## 7 Application to tree hashing

In the following subsection, we discuss the approach of using a tree hashing mode calling a sequential hash function. This is followed by two simple examples of tree hashing modes and a method to combine different hashing modes into one. We also discuss tree hash codings that satisfy the three conditions and on which multiple modes can be built and discuss some real-world hashing modes in the light of our conditions.

One can build a tree hashing mode calling a compression function, where it is assumed to behave as a fixed input-length (FIL) random oracle. The typical block cipher based compression function constructions, such as the Davies-Meyer mode, are trivially differentiable from a random oracle and are therefore not covered by our proof. On the contrary, it has been shown in [23,15,7] that a random permutation can be converted to a FIL random oracle simply by fixing part of its input and truncating its output.

### 7.1 Tree mode calling a sequential hash function

Sequential hashing modes typically come either with a security claim or an upper bound on the differentiating advantage of the form $N^2/2^{c+1}$, where $N$ relates to the number of queries to the underlying function $f$ and $c$ is a security parameter (e.g., the length of the inner chaining values or the capacity).

If we use a tree hashing mode (outer mode) calling a sequential hash mode (inner mode) calling an underlying function $f$, the total differentiating advantage is upper bound by the sum of the outer advantage $q^2/2^{n+1}$ and of the inner advantage $N^2/2^{c+1}$. To measure the cost of an adversary, we choose as unit the evaluation of the function $f$ since in practice it bears the bulk of the computational workload. In this context, the best an adversary can do is to choose messages in the outer mode that result in short node instances (e.g., $r$ blocks). We assume that a call to the sequential hash function results in only a small constant number $r$ of calls to $f$, leading to $q = rN$.

For an underlying function of given dimensions, one can now determine the optimal values of the chaining value size $n$ and of the security parameter $c$ for providing a given security level. We do the exercise for a sponge function [6,7]. Assume we have a permutation $f$ and we want to limit the total differentiating advantage to $N^2/2^{c'+1}$ for some target value $c'$. We further assume that $r = 1$, i.e., the tree hashing mode allows the adversary to query small node sizes at the cost of only one evaluation of the permutation $f$. The optimal choice of parameters in this case is to use the sponge construction with capacity equal to $c = c' + 1$ and a tree hashing mode with chaining values of length $n = c' + 1$.

### 7.2 Two simple examples

We now present two simple examples of tree hashing modes. These two modes are also discussed in [10], where they are instantiated with the KECCAK sponge function. In both modes, the tree parameters $A = (H, D, B)$ are composed of the height $H$ of the tree, the degree $D$ of the nodes and the leaf block size $B$.

All nodes end with a frame bit indicating whether it is a final or an inner node. Also, the tree parameters $A$ are encoded in the final node (e.g., as frame bits before the last one). For a node at height $h$, its index $\alpha = \alpha_0 || \dots || \alpha_{h-1}$ is in $(\mathbb{Z}^+ \cup \{0\}) \times \mathbb{Z}_D^{h-1}$ for the first mode, or in $\mathbb{Z}_D^h$ for the second mode.

In the first mode, the degree of the final node grows as a function of the message length, while the leaves have a fixed number of message pointer bits (i.e., $\alpha_0 \in \mathbb{Z}^+ \cup \{0\}$). The final node is connected to $\left\lceil \frac{|M|}{BD^{H-1}} \right\rceil$ balanced trees, each of height $H-1$ and degree $D$. The leaf nodes $Z_\alpha$ consist of $B$ message pointer bits covering the $B$ positions (or less if not enough bits in the message) starting from $B \sum_{i=0}^{H-1} \alpha_i D^{H-1-i}$. The (non-leaf) inner nodes have $D$ chaining blocks of length $n$.

In the second mode, the tree has a fixed size but the leaves input a variable number of message pointer bits (i.e., $\alpha_0 \in \mathbb{Z}_D$). The tree is a balanced tree of height $H$ and all (non-leaf) nodes have degree $D$. The leaf nodes $Z_\alpha$ consist of sequences of $B$-bit message blocks where the $j$-th block covers the $B$ positions (or less if not enough bits in the message) starting from $B(jD^H + \sum_{i=0}^{H-1} \alpha_i D^{H-1-i})$. The (non-leaf) nodes have $D$ chaining blocks of length $n$. This mode is easy to use when an upper bound on the number of parallel processes is known in advance. The inner hashes of each of the $D^H$ leaves can be fetched with $B$-bit blocks in parallel (Exploiting the available parallelism on a platform capable of less than $D^H$ independent computations still yields efficient modes in the case of long messages). In practice, restricting to $H = 1$ gives a simple tree that still allows to choose the number of leaves $D$ that can be computed in parallel.

It is clear that both modes are message-complete. Moreover, they implement final-node domain separation. Additionally, they are tree-decodable as the tree structure can be fully determined from $A$ encoded in the final node and from the length of the node instances.

### 7.3 Taking the union of tree hashing modes

We can create a new tree hashing mode $\mathcal{T}_{\text{union}}$ by taking the union of $n$ tree hashing modes $\mathcal{T}_i$ in the following way. $A_{\text{union}}$ is given by a choice parameter indicating the mode $i$ composed with the tree parameters $A_i$ for the particular mode. For instance, taking the union of the two modes presented in Section 7.2 simply just requires adding a binary choice parameter.

Taking the union of modes that use different coding may lead to loss of tree-decodability if no special precautions are taken. For instance, this problem can be fixed by coding in the final node for each of the modes the choice parameter $i$ such that it can be uniquely decoded (domain separation). A coding-oriented approach is given in the next section.

Conversely, restricting the range of tree parameters of a given tree hashing mode does not impact its soundness. In particular, when fixing the value of the tree parameters of a sound tree hashing mode to a single value, the tree template $Z$ is fully determined by the message length $|M|$.

### 7.4 Coding for tree hash modes satisfying the conditions

One can specify a tree hash coding that allows many different tree hashing modes and that satisfies our three conditions. Such a coding can in principle be seen as a super tree hash mode that is the union of all possible tree hash modes that can be realized by this tree hash coding. If the coding satisfies the three conditions, this super tree hash mode is sound, and hence, so are all tree hash modes supported by it. An example of such a tree hash coding is the Sakura coding that we published in [9]. In that paper, we prove that any Sakura-compatible mode is sound.

### 7.5 Checking of some real-world tree hash modes

Bitcoin [19] is a peer-to-peer electronic cash system that make use of tree hashing based on Merkle trees [18] and is specified at [21]. Remarkably, the employed tree hashing mode satisfies none of the three conditions, and it is easy to generate collisions or perform length-extension attacks. However, abusing these properties seems to be made infeasible by the higher-level layers of the protocol.

The Tree Hash Exchange (THEX) format is proposed for assisting in checking the integrity of exchanged files, allowing arbitrary subranges of bytes to be verified before the entire file has been received [12]. It satisfies tree-decodability by means of domain separation between leaf nodes and inner nodes and message-completeness. It is not final-node separable, so it is vulnerable to length extension. This may, however, not be a requirement for the typical use cases.

MD6 [23] was a SHA-3 candidate that applies a hash tree mode to a compression function based on a permutation. Its tree hash mode satisfies all three conditions. This is done at the cost of 73 frame bits per node call: 64-bit word $U$ for the location of the node in the tree, and 9 bits in metadata word $V$: the $z$-bit indicating whether it is the final node and the maximum tree height coded on 8 bits. For soundness, just 2 frame bits per node would have been sufficient.

## 8 Implications for sequential hashing

Sequential hash function modes can be seen as a special case of tree hashing modes, where the tree reduces to a single linear sequence, the inner hash function has fixed input-length (i.e., is a compression function) and the parameters are *empty*. Therefore, the conditions for tree hashing modes introduced in Section 4 can be applied to sequential hash functions.

In this section, we present the techniques for satisfying the three conditions and discuss a number of published modes in this context.

Clearly, a sequential mode has a single leaf node and the size of all nodes is equal to the input-length of the compression function. We limit ourselves to modes in which all nodes but the leaf and final nodes (and possibly the one before the final one, called *pre-final*) contain a chaining value, a message block and some frame bits in a fixed layout. Clearly, the leaf node has no chaining value and the message block in the final node (or the pre-final one) may be shorter.

Most techniques we discuss introduce frame bits, which cause an overhead as they either require a larger compression function or take the place of message or chaining bits. In the following, we measure the overhead by the number of frame bits added.

Note that in all these cases our conditions are sufficient if $\mathcal{F}$ behaves as a (FIL) random oracle. The case of $\mathcal{F}$ being an ideal cipher in Davies-Meyer mode is not covered as it can trivially be differentiated from a FIL random oracle. In [23,15] a construction is provided to construct a FIL random oracle from a random permutation.

Our analysis in Sections 8.4 and 8.5 is closely related to the notion of free-IV hashing and its indifferentiability analysis in [2].

### 8.1 Satisfying tree-decodability

Tree-decodability is satisfied if any given node instance can be identified as the leaf node, the final node, the pre-final node or another inner node. This is trivial for the final and pre-final nodes thanks to their position in the tree. For the leaf node, this is, however, not the case. We distinguish three techniques:

**Domain separation** We include in each node a frame bit that codes whether it is the leaf node or not. The overhead is a single bit per node and is proportional to the message length.

**Length coding** We use frame bits in the final node to code the height of the leaf node. A variant, coding of the message length, is often called *Merkle-Damgård strengthening*, and allows computing the height of the leaf node. Length coding implies the adoption of a coding convention for integers. Typically, this integer is coded in a fixed-length field imposing an upper limit to the message length that may be supported by a mode. The overhead is independent of the message length and is $\log_2(X/m)$ bits with $X$ the maximum message length and $m$ the length of message blocks.

**Initial Value (IV)** In bit positions where other nodes have a chaining value, we put in the leaf node a block of frame bits with a value specified in the mode. When $\mathcal{T}$-decoding a node, one can now check for the presence of the IV to determine whether it is the leaf node or not. This resolves tree-decodability in all cases except a non-leaf node in which a chaining value occurs with value IV. To cover this case, we need to slightly modify our simulator and this results in a marginally different bound. We discuss this in Section 8.4. The overhead is independent of the message length: $n$ bits.

In the basic mode of [14], non-leaf nodes consist of the concatenation of a chaining value, a single frame bit with value 1 and a message block $X_i$. In the leaf node, an all-zero IV takes the place of the chaining value and the single frame bit. Hence, tree-decodability is guaranteed by domain separation between leaf node and the other nodes. Additionally, [14] proposes a variant where the single frame bit equal to 1 is not present and tree-decodability fully relies on the presence of the IV in the leaf node.

### 8.2 Satisfying message-completeness

In a sequential mode, the message bits are mapped sequentially to the message blocks of the individual nodes. Satisfying message-completeness thus comes down to determining the message length. If it does not have an IV, the leaf node may have a larger message block than the other nodes. As any message length shall be supported, the final (or pre-final) node may have a shorter message block and the remaining bit positions are filled with frame bits (padding). To uniquely determine the message length, we distinguish two techniques:

**Reversible padding** We perform padding to the message to result in a multiple of the message block length. Typically a single frame bit with value 1 is appended and a minimum number of frame bits with value 0 to have a multiple of the message block length. The overhead is in the range $[1, m]$ bits and does not scale with the message length.

**Length coding** We code the length of the message in the final node, or append it to the message. To fit the node lengths, some additional padding must be performed. The overhead does not scale with the message length and is in the following range:

$$[\log_2(X/m), \log_2(X/m) + m - 1].$$

### 8.3 Satisfying final-node domain separation

For final-node domain separation we distinguish the following techniques:

**Frame bit** We include in the nodes a single frame bit that codes whether the node is the final one or not. The overhead is 1 bit per message block. This method was proposed in [13] as a method to implement input prefix-free coding.

**IV** In the bit positions where other nodes have a chaining value, we put in the final node a block of frame bits with a value specified in the mode. This implies that the chaining value must be put elsewhere in the node and this goes at the cost of the message block. We discuss this case in Section 8.4. The overhead is independent of the message length: $n$ bits.

### 8.4 Relying on IV values for indifferentiability

For tree-decodability let us assume that recognizing the leaf node relies on the presence of an IV. Then, our simulator may generate an inner collision without a collision in the compression function. We give a simple example. Assume that the simulator upon receipt of a query with a node with message block $\mu$ and containing the IV has by chance generated the IV as response. The distinguisher can then query $\mathcal{G}[\mathcal{RO}]$ with two messages: a message $M$ and a message $\mu|M$ and if it returns different responses, she knows it is $\mathcal{G}[\mathcal{RO}]$ and not $\mathcal{T}$. The probability that the responses are different is $1 - 2^{-\ell}$, with $\ell$ the number of requested bits, and hence, the mode of use is differentiated. Actually Lemmas 2 and 3 do not hold due to the fact that they rely on tree-decodability via Lemma 1.

However, it is easy to fix it by slightly adapting the simulator. It suffices to initialize the set $P_n$ to $\{\text{IV}\}$ rather than the empty set. Then the simulator avoids $\{\text{IV}\}$ as a chaining value and tree-decodability is repaired.

Similarly, if final-node recognition is based on the presence of a value IV2, the simulator can in principle erroneously recognize an inner node as a final node when the chaining value it contains happens to be IV2. This can be avoided by including IV2 in $P_n$ from the start. This guarantees that the simulator will never return IV2 as a chaining value.

So, the initialization of $P_n$ to the set of IV values fixes Lemma 2 and Lemma 3 and has no impact on Lemma 4. However, it does have an impact on the bound in Lemma 5. So Theorem 1 remains valid but with a different bound. Let us denote the number of IV values defined in the mode by $z$ and denote the bound by $\epsilon_n(q, z)$. Note that we define $\epsilon_n(q) = \epsilon_n(q, 0)$.

If the set $P_n$ is initialized with $z$ IV instances, the response to the $i$-th query is chosen from at least $2^n - i + z + 1$ values rather than $2^n - i + 1$ values. This yields the following expression for the bound:

$$\begin{aligned}\epsilon_n(q,z) &= 1 - \prod_{i=z}^{q+z-1}\left(1 - \tfrac{i}{2^n}\right) \\ &< 1 - \exp\left(\tfrac{(q+z)(q+z-1)}{2^{n+1}}\right) \\ &\approx \tfrac{(q+z)^2}{2^{n+1}}.\end{aligned}$$

Typically $z$ is small (1 or 2) and, for large values of $q$, it holds that $\epsilon_n(q,z)/\epsilon_n(q)$ is very close to 1. We conclude that the price paid for counting on IV values for satisfying tree-decodability and final-node domain separation is a negligible deterioration of the bound.

Mind that relying on a particular IV for tree-decodability introduces the possibility to have inner collisions without a collision in the inner function and hence collision resistance is no longer preserved.

In [4], the enveloped Merkle-Damgård (EMD) transform was presented that makes use of IVs and that preserves collision resistance. It has a particular IV in the leaf node and another IV in the final node. However, it does not require the IV in the leaf node for tree-decodability as it also appends the message length to the padded message. The upper bound in our proof is $\epsilon_n(q,2)$, which is better than the one given in [4].

The EMD transform can be seen as an improved version of two modes previously proposed in [13]. These modes are called NMAC and HMAC respectively and are inspired by the MAC function constructions with the same name published in [3]. In the NMAC mode, tree-decodability is realized with an IV in the leaf node. Final-node domain separation is avoided by applying a so-called independent function to the hash output of the final node. In practice this would typically be realized with the same compression function, but having domain separation. In HMAC, leaf and final nodes can be recognized by the presence of an IV. One can distinguish between the two by the presence of an all-zero block in the leaf node. In the final node there is a chaining value in that place. The upper bound in our proof is $\epsilon_n(q,1)$ for NMAC and $\epsilon_n(q,3)$ for HMAC.

### 8.5 IVs as a public resource

The value of the IVs can either be part of the definition of the mode $\mathcal{T}$, or a public resource like the compression function $\mathcal{F}$. So far, we have considered the former approach, and IVs are implemented as frame bits. In the latter approach, the IVs are not known in advance, but they have to be queried, either by the mode $\mathcal{T}$ or by an adversary.

Concretely, we can consider the IVs as part of the definition of the compression function $\mathcal{F}$. If a mode uses $z$ IVs, we can extend the domain of the compression function with $z$ artificial elements $\{\diamondsuit_1, \ldots, \diamondsuit_z\}$ and consider the IV values as the images through $\mathcal{F}_n$ of these new elements, i.e., $\mathrm{IV}_i = \mathcal{F}_n(\diamondsuit_i)$. In the mode, putting $\mathrm{IV}_i$ in a leaf node or in the final node is then modeled as putting chaining bits pointing to a new node whose input is $\diamondsuit_i$. (A node with an IV is now no longer a leaf node but rather the parent of a leaf node containing $\diamondsuit_i$ as frame "bits".)

The difference between the two approaches is rather philosophical, and we see this simply as a different way to model the introduction of IVs. Choosing between the two is rather a matter of taste.

With IVs as a public resource, the original three conditions apply directly, without the need to adapt the simulator, but at the price of an artificial extension of the domain of $\mathcal{F}$. Here, the bound is again $\epsilon_n(q, 0)$, as no IV has to be put in $P_n$, but instead the burden of learning about the IVs goes to the adversary, who does not know them in advance and has to make $z$ more queries. Another difference is that a collision of a chaining value with an IV implies a collision in the compression function, applying Lemma 1 directly, with the extended domain of $\mathcal{F}$.

### 8.6 Techniques for avoiding final-node domain separation

Reserving a frame bit for domain separation between final and inner nodes is sometimes perceived as too costly. Techniques are proposed in literature to prevent length extension attacks without final-node domain separation. Remarkably, the techniques we have seen so far appear to cost more than final-node domain separation. Three proposed techniques are:

**Chopping** By chopping $s$ bits from the output, i.e., reducing the output to $n - s$ bits, length extension requires guessing $s$ bits. In [11] the following differentiability bound is proven for certain sequential modes calling a compression function and chopping $s$ bits:

$$\frac{(3(n-s)+1)\,Q}{2^s} + \frac{Q}{2^{n-s-1}} + \frac{q^2}{2^{n+1}}\,,$$

with $q$ the total number of calls to the compression function and $Q$ the number of calls to the outer hash function. When chopping half of the bits, i.e., $s = n/2$, this yields:

$$\frac{3(n+2)Q}{2^{(n/2)+1}} + \frac{q^2}{2^{n+1}}\,.$$

For $Q < 2^{n/2}$ this differentiability bound is very close to the optimum. However, chopping reduces the output-length to $n - s$, increasing the success probability of finding an output collision after $q$ queries by a factor $2^s$ and leading to an expected workload of $2^{(n-s)/2}$ rather than $2^{n/2}$. If a resistance level $2^{c/2}$ is desired against all generic attacks including generating collisions, the best one can achieve with chopping is taking $n \approx 3c/2$ and $s \approx n/3$. So for the same security level $2^{c/2}$, this method results in an overhead of about $c/2$ bits per node as compared to a mode that does final-node domain separation.

**Tweaking** This method consists of tweaking the chaining value in the final node by a simple function. A typical tweak is the addition of a nonzero constant $X$. This method was proven indifferentiable in [16]. When looking at it from the perspective of our proof, in this construction the simulator cannot distinguish between inner and final nodes. However, we can adapt our simulator to avoid inner collisions and guarantee $\mathcal{T}$-consistency also for this case; upon receipt of a query $s$ to which it returns $t$, it stores in $P_n$ both $t$ and $t \oplus X$ and the chaining value present in $s$. This adds three values to $P_n$ for each query and leads to a bound that is a factor 3 larger than the optimum one (but still smaller than the one proven in [16]). Hence, this suggests that this method is also less efficient than final-node domain separation (i.e., $\log_2 3 > 1$ extra chaining value bits would be necessary to compensate for this extra factor 3 in the bound).

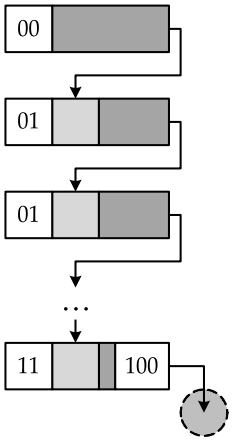


**Fig. 3.** Tree template for a simple sequential mode

**Pre-pending the message length** By coding the length of the message in the leaf node, length extension is prevented. Note that independently leaf node identification must be guaranteed with a dedicated frame bit or an IV. This method is not covered by our conditions. It was proposed in [13] as a form of prefix-free input coding and proven indifferentiable. The overhead of this method is limited. However, this method has an important drawback that makes it impractical for many applications: the length of the message must be known in advance.

The simplest sequential hashing mode, that is sound, is the following. All but the leaf and final nodes consist of an $m$-bit message block, an $n$-bit chaining value and two frame bits (coding final/inner node and leaf/non-leaf node). The leaf node has no chaining value but an $n + m$ bit message block. The final node has an incomplete message block of length in $[0, m - 1]$ followed by a single frame bit with value 1 and up to $m - 1$ frame bits with value 0 (for padding). This is illustrated in Figure 3.

## 9 Conclusions

In this paper, we have given a set of sufficient conditions for both tree and sequential hashing modes to be sound. If these conditions are satisfied, the differentiability bound is as tight as theoretically possible: it is only limited by the length of chaining values and independent of the output-length. While the conditions were mainly aimed at tree hashing, they shed a different light to most published sequential hashing modes and allowed for improved bounds in some cases.

## References

1. E. Andreeva, B. Mennink, and B. Preneel, *Security reductions of the second round SHA-3 candidates*, Cryptology ePrint Archive, Report 2010/381, 2010, `http://eprint.iacr.org/`.
2. N. Bagheri, P. Gauravaram, L. R. Knudsen, and E. Zenner, *The suffix-free-prefix-free hash function construction and its indifferentiability security analysis*, Int. J. Inf. Sec. **11** (2012), no. 6, 419–434.
3. M. Bellare, R. Canetti, and H. Krawczyk, *Keying hash functions for message authentication*, Advances in Cryptology – Crypto '96 (N. Koblitz, ed.), LNCS, no. 1109, Springer-Verlag, 1996, pp. 1–15.

4. M. Bellare and T. Ristenpart, *Multi-property-preserving hash domain extension and the EMD transform*, Advances in Cryptology – Asiacrypt 2006 (X. Lai and K. Chen, eds.), LNCS, no. 4284, Springer-Verlag, 2006, pp. 299–314.
5. 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.
6. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Sponge functions*, Ecrypt Hash Workshop 2007, May 2007, also available as public comment to NIST from `http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html`.
7. ———, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, `http://sponge.noekeon.org/`, pp. 181–197.
8. ———, *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.
9. ———, *Sakura: a flexible coding for tree hashing*, Cryptology ePrint Archive, Report 2013/231, 2013, `http://eprint.iacr.org/`.
10. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, Keccak *implementation overview*, May 2012, `http://keccak.noekeon.org/`.
11. D. Chang and M. Nandi, *Improved indifferentiability security analysis of chopMD hash function*, Fast Software Encryption (K. Nyberg, ed.), Lecture Notes in Computer Science, vol. 5086, Springer, 2008, pp. 429–443.
12. J. Chapweske and G. Mohr, *Tree Hash EXchange format (THEX)*, 2003, `http://adc.sourceforge.net/draft-jchapweske-thex-02.html`.
13. 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.
14. 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.
15. 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.
16. S. Hirose, J. Park, and A. Yun, *A simple variant of the Merkle-Damgård scheme with a permutation*, Asiacrypt, 2007, pp. 113–129.
17. 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.
18. R. C. Merkle, *Secrecy, authentication, and public key systems, PhD thesis*, UMI Research Press, 1982.
19. S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2008, `http://bitcoin.org/bitcoin.pdf`.
20. NIST, *Federal information processing standard 180-2, secure hash standard*, August 2002.
21. Bitcoin Portal, *Bitcoin protocol specification*, 2013, `https://en.bitcoin.it/wiki/Protocol_specification`.
22. T. Ristenpart, H. Shacham, and T. Shrimpton, *Careful with composition: Limitations of the indifferentiability framework*, Eurocrypt 2011 (K. G. Paterson, ed.), Lecture Notes in Computer Science, vol. 6632, Springer, 2011, pp. 487–506.
23. 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/`.
24. 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/`.

## A On this version of the paper

Compared to the earlier versions of this paper we have removed one of the four conditions for soundness, reducing them to three. This removed condition, *parameter-completeness*, applied to parametrized hashing modes and was necessary for guaranteeing that a message $M$, hashed with two different parameter values $A$ and $A'$ would not lead systematically to the same digest. More specifically, we considered two different inputs $(M, A)$ and $(M, A')$ leading to the same digest to be a collision. For some message lengths, the tree

template may be independent of a parameter value and hence, the digest is independent too. Avoiding a collision for such cases requires to explicitly code this parameter value in frame bits. In this version we no longer consider this to be a collision. This reflects the situation in real-world applications where one speaks of a collision if two different messages lead to the same digest. Actually, the need for our fourth condition was an artefact of the ideal hash function definition that we adopted in our security proof. In this and the previous version of the paper we have adapted this definition, removing the need for parameter-completeness.

In this version of the paper we have made the reasoning more formal by adding definitions and making the $\mathcal{T}$-decoding and simulator algorithms more rigorous. However, the general set-up and philosophy remains unchanged. Furthermore, the applicability of the three conditions stays the same.

Additionally, in previous versions of this paper we claimed that satisfying the four conditions implies preservation of second preimage resistance. We acknowledge Stefan Lucks for pointing out to us this is not the case and have removed that claim.

## B Illustrations

In this section, we illustrate two undesired properties of tree hashing modes explained in Section 4 to introduce two of the three conditions for sound tree hashing. We give some figures of templates generated by some mode of use. The way these templates have been generated by the mode of use are out of scope of this section. Note also that these templates illustrate undesired properties, and hence, the modes of use that would produce them are per definition not sound.

We use the following conventions. Instead of depicting individual bits, we depict message/chaining/frame blocks, where a block is just a sequence of consecutive bits. Frame blocks are depicted by white rectangles with its value indicated, message blocks by light gray rectangles and their position in the message indicated, and chaining blocks by dark gray rectangles with an indication of their child. An output is depicted by a rounded rectangle. The nodes are identified with their indices and the relation between the nodes is additionally indicated by arrows, symbolizing the application of $\mathcal{F}$ during template execution for a concrete input $M$.

The first property is related to the existence of inner collisions in the absence of collisions in the output of $\mathcal{F}$ and is illustrated in Figure 4. The figure depicts two templates that are generated by a mode of use $\mathcal{T}$ for two different message lengths. All nodes have as first two bits frame bits with value 01. The template on the left has four nodes: three leaf nodes of height 1 and a final node that takes an input block and the chaining values corresponding to the three leaf nodes. The template on the right has three nodes: two leaf nodes of height 1 and a final node that takes an input block and the chaining values corresponding to the two leaf nodes. Note that the final node of the right template has a message block (indicated by $M_0'$) in the place where the final node of the left template has the concatenation of a message block $M_0$ and a chaining block $CV_2$. We can exploit this fact to construct an inner collision from any message $M$ with length matching the left template. As can be seen in the figure, it suffices to form $M'$ by replacing in $M$ the block $M_1$ by $\mathcal{F}(01|M_1)$.

The second property, a generalization of length extension to tree hashing, is illustrated in Figure 5. Given the output of $h = \mathcal{T}[\mathcal{F}](M)$ of some message $M$, length extension is

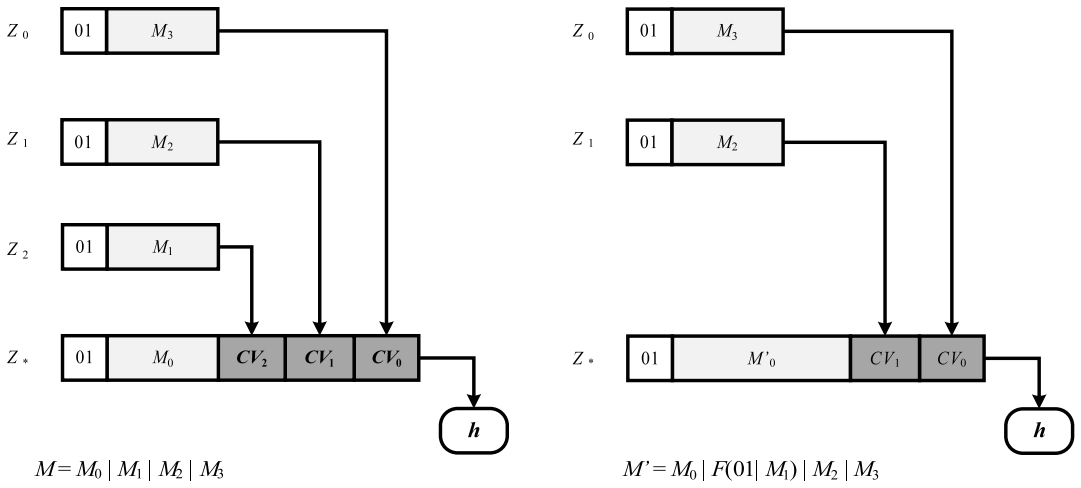


**Fig. 4.** Example of an inner collision without a collision in $\mathcal{F}$

the possibility to compute the output of $\mathcal{T}[\mathcal{F}](M')$ with $M$ a substring of $M'$, only knowing the output $h$ and not $M$ itself. Figure 5 depicts two templates corresponding with two different message lengths. The templates have a binary tree structure. The template at the left has three nodes: two leaf nodes and a final node containing the chaining values corresponding to the two leaf nodes. The template at the right has seven nodes: four leaf nodes, two intermediate nodes each containing the chaining values corresponding to two leaf nodes and a final node containing the chaining values of the intermediate nodes. Note that the chaining block $CV_0$ in the final node of the right template corresponds with the hashing output of the left template. As can be seen in the figure, given the hash output $h$ of a message $M$ with length matching the left template, one can compute the hash output of any message $M' = M|M_2|M_3$ with length matching the right template without knowledge of $M$.

## C Remarks on the cost

The cost measure introduced in Section 5.2 aims at counting on an equal footing both queries to $\mathcal{H}$ and queries to $\mathcal{I}$. We wish to illustrate this by comparing two examples of distinguisher.

The first distinguisher uses only the $\mathcal{I}$ interface to produce a collision in $\mathcal{F}_n$ (or in the simulator). Assuming a collision is produced, two messages can be built, so as to turn this collision into an inner collision in $\mathcal{T}$ but not in $\mathcal{G}$. This attack takes about $2^{n/2}$ queries. (If after $2^{n/2}$ attempts no collision has been found, the distinguisher may suspect it is not querying $\mathcal{F}$ but a simulator.)

The second distinguisher uses only the $\mathcal{H}$ interface and attempts to exhibit an inner collision directly. When talking to $\mathcal{T}$, such an inner collision can occur, but when talking to $\mathcal{G}$, an inner collision does not even exist (with the requested output-length sufficiently large to detect such an inner collision with arbitrary certainty). More specifically, the dis-

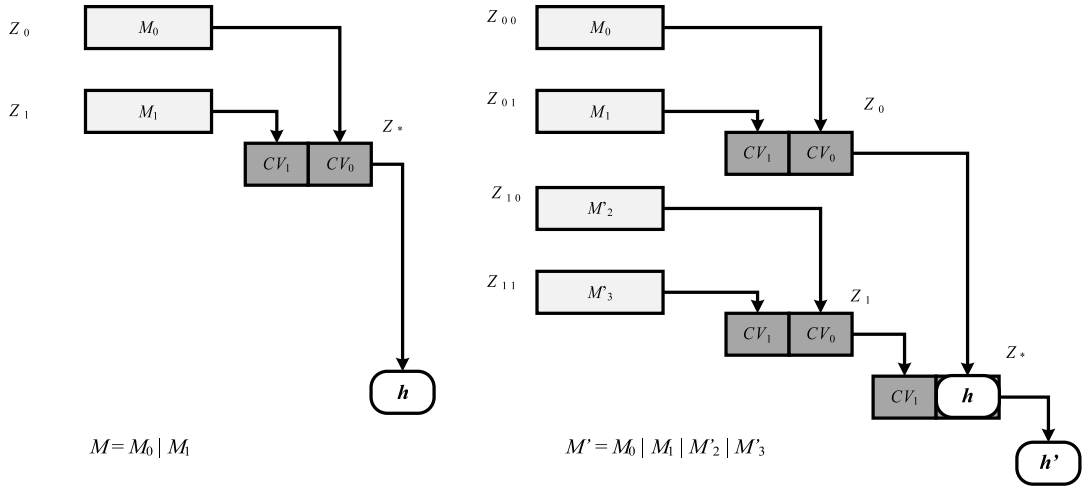


**Fig. 5.** Example of the generalization of length extension to tree hashing

tinguisher queries the $\mathcal{H}$ interface with equal tree parameters $A$ and messages $M_i$ that vary only in one leaf, which is chosen to have the maximum height $H$ in the tree. To obtain an inner collision, it is sufficient to get a collision at any of the $H$ nodes on the way from the leaf to the final node. The distinguisher needs about $2^{n/2}/H$ queries to hit an inner collision. Hence, in this context a query to $\mathcal{H}$ appears to be a factor $H$ more powerful than a query to $\mathcal{I}$.

The cost function that counts calls to $\mathcal{F}$ and discards duplicate queries as one, brings the two distinguishers to a more equal footing. The first distinguisher succeeds at a cost of about $2^{n/2}$. The queries of the second distinguisher could be performed at the level of the $\mathcal{I}$ interface, the tree mode $\mathcal{T}$ being simulated by the distinguisher. In this case, each query to $\mathcal{H}$ translates into $f_{\mathcal{T}}(|M|, A)$ queries to $\mathcal{I}$. However, the strategy of the second distinguisher is such that only $H$ $Q_{\mathcal{I}}$ queries differ for each of the $2^{n/2}/H$ $Q_{\mathcal{H}}$ queries. Hence, the cost of $Q_{\mathcal{I}}$ for the second distinguisher is also about $2^{n/2}$.