Sufficient conditions for sound tree hashing modes

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

[1] STMicroelectronics
[2] NXP Semiconductors

Abstract. Tree hashing has several advantages over sequential hashing such as parallelism and a lower cost of hash value recomputation when only a small part of the input changes. In this paper we consider the general case of tree hashing modes that make use of an underlying (sequential) hash function. We formulate a set of four simple conditions, which are easy to implement and to verify, for such a tree hashing mode to be *sound*. We provide a proof that for any tree hashing mode satisfying these four conditions, the advantage in differentiating it from an ideal monolithic hash function is upper bound by $q^2/2^n$ with q the number of queries to the underlying hash function and n the length of the chaining values. We apply our results to sequential hashing modes calling a fixed-input-length compression function and show for two examples of simple tree hashing modes that they are sound. Finally we show a simple method to take the union of tree hashing modes that preserves soundness.

Keywords: tree hashing, indifferentiability

1 Introduction

Most hash functions are iterated where the message blocks are processed sequentially and the processing of a block requires all previous blocks to be processed. This severely limits the efficient use of multi-processors when hashing a single (long) message. By adopting tree hashing, several parts of the message may be processed simultaneously and parallel architectures can be used more efficiently in the hashing of a single message. 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.

Tree hashing was already introduced in [11]. Recently, in the SHA-3 competition, a number of hash functions were proposed that have tree hashing built into their mode of use, i.e., the mode of use is a tree hashing mode calling a fixed-input-length (FIL) compression function [9,12,14,8].

In this paper, we discuss tree hashing modes that call a hash function without 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 FIL compression functions.

Our aim is to formulate a number of simple conditions for such a tree hashing mode to be *sound*. For the soundness, we base ourselves on the indifferentiability framework introduced by Maurer et al. in [10] and applied to hash functions by Coron et al. in [7]. To do that, we follow a proof technique very similar to the one introduced in [4]. The remainder of this paper is organized as follows. After providing a rigorous definition of tree hashing modes in Section 2, we introduce in Section 3 a set of simple and easy-to-verify conditions for tree hashing modes that result in sound tree hashing. After adapting the indifferentiability setting of [7] to tree hashing in Section 4, we provide in Section 5 a clear indifferentiability proof valid for any tree hashing mode satisfying our criteria. In Section 6, we discuss the choice of the underlying compression function and its implications. Independently of the indifferentiability framework, we show in Section 7 that some properties of the underlying compression function are preserved in the construction. We show in Section 8 the applicability of our results by applying them to classical Merkle-Damgård style hashing in Section 8.1, proving the

soundness of two examples in Section 8.2, and showing a simple method to take the union of tree hashing modes that preserves soundness in Section 8.3.

Provable security of tree hashing was already investigated in [13] and indifferentiability proofs have been given, e.g., for the mode used in MD6 [12]. Our paper expands the concepts presented in [6] and was actually inspired by the proofs in [12] that were quite specific for the mode of use adopted in MD6. Our goal was to formulate a set of simple conditions that should be easy to verify and implement, sufficient for a tree hashing mode to be sound. In the meanwhile the authors of [12] 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 [15]. Not unsurprisingly, the conditions in this paper and those in [15] turn out to be very similar.

Despite this similarity, we believe this paper has a substantial added value with respect to prior work, including [15]. First, the bound we achieve on the success probability of differentiating the tree hashing mode from an ideal hash function is tighter (and simpler to express) than the one in [15]. As a matter of fact, it only depends on the length of the chaining values and is as tight as theoretically possible. Second, the set of conditions and the corresponding indifferentiability proof in this paper is short and easier to verify than those in many papers on provable security. Third, we treat a more general case than prior work in several aspects. Our mode of use is *parameterized*, with parameters specifying the way to build the tree. Also, our tree hashing modes can generate indefinite-length outputs, allowing us to prove indifferentiability from an random oracle with indefinite output length, rather than a truncated random oracle. Finally, in our construction the inner hash function is not necessarily a compression function but can have variable output length. This is useful when building a tree hash function using a variable output-length sequential hash function as component (see Section 6).

2 Tree hashing mode

Most hash functions are constructed in a layered fashion. Typically, a hash function has a variable input length and a 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, i.e., a fixed-input-length (FIL) and fixed-output-length hash function with input length larger than output length. The task of the mode is to call F with inputs composed from message bits and output bits of previous calls to F.

In this section we generalize this idea. We do not impose limits to the input or output length of the underlying function and consider a wide range of tree hashing modes. We call the underlying hash function the *inner hash function* and denote it 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 classical compression function (see Section 8.1).

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 and output lengths.

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.

It is reasonable to assume that the way the message is cut into blocks and processed is independent of the actual bit values of the message, although it can certainly depend on its length. In this paper, we thus assume that the tree hashing mode can be formally expressed as a two-step process. First, the *tree template generator* $\mathcal{Z}$ is a deterministic algorithm that constructs a *tree template* $Z = \mathcal{Z}(|M|, A)$, which depends only on the length of the message $|M|$ and on the tree parameters

A. The tree template Z is a recipe for $\mathcal{T}[\mathcal{F}]$, which tells, e.g., how to cut M into blocks, how to format the input to $\mathcal{F}$ and how to use the output of $\mathcal{F}$ to generate another input to $\mathcal{F}$ down the tree. Second, $\mathcal{T}[\mathcal{F}]$ follows the template Z by instantiating it with the bits of the actual message M and the intermediate chaining values.

A tree template Z is composed of *node templates* Z_α, where α denotes the index of a node template. The nodes form a directed acyclic graph and make a tree. The structure of the tree is reflected in the way the nodes are indexed. The index α of a node template consists of a string of integers and has the following properties:

- The node template Z_*, where $*$ denotes the empty string, is called the *final node template*. (By contrast, the other nodes are called *inner nodes*.)
- In general, a node template Z_α is connected to zero, one or more other nodes, called its *sons*. The number of sons of a node is called its *degree* d. The sons of a node template Z_α are the nodes $Z_{\alpha||0}, Z_{\alpha||1} \ldots Z_{\alpha||d-1}$. We call a node template with no sons a *leaf node template*.
- The *height* of a node template is the number of integers in its index; it is also the distance to the final node template, which has height 0. The height of a tree template is the maximum height over all its nodes.

When instantiating the tree for a given value of M, the node templates become bitstrings called *node instances* forming a *tree instance*. The definitions above can be translated to node instances, replacing the word "template" by the word "instance". We simply say "node" when it is clear from the context if we refer to a node template or a node instance.

A node template Z_α is a sequence of *template bits* $Z_\alpha[x]$, $0 \leq x < |Z_\alpha|$, and specifies the way a node template Z_α becomes a node instance S_α when the value of M is fixed. There are three types of template bits, which can be identified in a node template:

- Frame bits: a frame bit has a single attribute: its binary *value*. The value is fully determined by A and $|M|$. When instantiating, each frame bit $S_\alpha[x]$ takes the value of $Z_\alpha[x]$.
- Message bits: a message 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. When instantiating, we assign the message bit $M[y]$ to $S_\alpha[x]$, where y is the position attribute of $Z_\alpha[x]$.
- Chaining bits: a chaining bit has two attributes: a *son number* and a *position*. The son number is an integer in the range $[0, d - 1]$ with d the degree of Z_α and identifies a son node. The position is an integer that points to a bit position in the output of $\mathcal{F}$. When instantiating, we set $S_\alpha[x] \leftarrow \mathcal{F}(S_{\alpha||j})[y]$, where j is the son number attribute of $Z_\alpha[x]$ and y is its position attribute. (A *chaining value* is the set of all chaining bits coming from the same son.)

Computing the outer hash function $\mathcal{T}[\mathcal{F}]$ for a given input (M, A) can now be specified as follows. First, the tree template is built $Z = \mathcal{Z}(|M|, A)$. Then, the node templates are instantiated and subject to $\mathcal{F}$ starting with the leaves and ending with the final node, each node being processed only after its sons are. Finally, the output of the outer hash function is obtained by applying $\mathcal{F}$ on the final node instance, namely,

$$\mathcal{T}[\mathcal{F}](M, A) = \mathcal{F}(S_*).$$

3 Sufficient conditions for sound tree hashing

In this section we first 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. Then we introduce four conditions that a tree hashing mode should satisfy. In Section 5 we will prove that

the strength of a tree hashing mode that satisfies these four conditions equals this upper bound, and hence is as tight as theoretically possible. The differentiability upper bound and the first three conditions stem from the ability to generate collisions. The last condition prevents a generalization of length extension.

Throughout the text, we assume that $\mathcal{F}$ is a random oracle, with infinite output, and that the tree hashing mode can provide an output with indefinite length.

First, we 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 $Z = Z' = \mathcal{Z}(|M|, A)$. For some fixed index α, we construct pairs of messages that differ only in bits that are mapped to nodes with indices restricted to $\alpha||\beta, \forall\beta$ (e.g., the two messages differ only in one leaf). This difference can only propagate to the final node via the chaining bits referring to Z_α in its parent node. Let the number of these chaining bits be denoted by n_α. 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_α 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 for $N < 2^{-n/2}$ is

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

This reasoning is independent of the value of α, 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 denote by $\mathcal{F}_n$ the truncation of $\mathcal{F}$ to its first n output bits. In the following we will assume that $\mathcal{F}_n$ is used for the inner nodes.

We will now further elaborate on collisions. For this, we introduce the concept of inner collision.

Definition 1. *An* inner collision *in $\mathcal{T}[\mathcal{F}]$ is a pair of different tree instances $S \neq S'$ with equal final node instances $S_* = S'_*$.*

With an inner collision, the output of $\mathcal{T}[\mathcal{F}]$ is equal for all output bits, not just the first n bits.

A collision of $\mathcal{F}_n$ can be used to generate an inner collision in $\mathcal{T}[\mathcal{F}]$. However, 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'. The values of $(|M|, A)$ and $(|M'|, A')$ are chosen in such a way that for all indices $\alpha \notin \{\beta, \beta||0\}$ we have $Z_\alpha = Z'_\alpha$. Nodes Z'_β and $Z_{\beta||0}$ are both leaf nodes, and Z_β has degree 1. Additionally, Z_β and Z'_β have the same length and in the positions where there are chaining bits in Z_β, there are message bits in Z'_β. For a given M, we can now compute all node instances; this includes the chaining bits in S_β by instantiating $S_{\beta||0}$ and evaluating $\mathcal{F}_n(S_{\beta||0})$. We can then construct M' such that $S'_\beta = S_\beta$ and $S'_\alpha = S_\alpha$ for all the indices α in Z'. As S has one more node than S', the tree instances are not equal and hence we have an inner collision.

In this case, the inner collision is only possible because the node templates Z_β and Z'_β are different. A simple way to avoid this situation is mandating that $\mathcal{Z}$ is *template-decodable*.

Definition 2. *A tree template generator $\mathcal{Z}$ is* template-decodable *if for any tree instance S generated with $\mathcal{Z}$ the following holds. For any index α, the knowledge of the node instances (S_β), for β running through all prefixes of α (including S_* and S_α itself), is sufficient to determine the node template Z_α.*

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

Lemma 1. *When $\mathcal{Z}$ is template-decodable and $\mathcal{T}$ uses the first n bits of $\mathcal{F}$ as 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 with $S_* = S'_*$. Let α be any index such that $S_\beta = S'_\beta$ for all β prefixes of α. From the decodability property, one can unambiguously determine the node templates $Z_\alpha = Z'_\alpha$. The number of sons is thus identical for both node templates.

Now let α be as above and j an integer such that $S_{\alpha||j} \neq S'_{\alpha||j}$. From the node template Z_α, the positions of the bits of $\mathcal{F}_n(S_{\alpha||j})$ or $\mathcal{F}_n(S'_{\alpha||j})$ in S_α are known. Hence, $S_\alpha = S'_\alpha$ implies $\mathcal{F}_n(S_{\alpha||j}) = \mathcal{F}_n(S'_{\alpha||j})$. □

Condition 1 *$\mathcal{Z}$ is template-decodable.*

This condition, as well as the next three ones, are easy to verify and to implement. This is discussed and illustrated on some examples in Section 8.

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. This leads to the condition for $\mathcal{Z}$ that for all inputs $(|M|, A)$, the resulting template tree $Z = \mathcal{Z}(|M|, A)$ has at least one message bit for every position x in $[0, |M| - 1]$. We call this property *M-completeness*. (Basically, all the message bits enter the function $\mathcal{F}$.)

Condition 2 *$\mathcal{Z}$ is M-complete.*

Similarly, we can have an output collision in $\mathcal{T}[\mathcal{F}]$ without an inner collision if the template does not allow to fully determine the tree parameters A. This leads to the condition for $\mathcal{Z}$ that for all inputs $(|M|, A)$, the resulting template tree $Z = \mathcal{Z}(|M|, A)$ allows fully determining A. We call this property *A-completeness*.

Condition 3 *$\mathcal{Z}$ is A-complete.*

The final condition is related to a property that generalizes length extension to tree hashing and requires the definition of *subtrees*.

Definition 3. *We call a tree instance S' a* subtree *of tree instance S if for all nodes S'_β of S' and for some index α in S, it holds that $S'_\beta = S_{\alpha||\beta}$.*

Finally, let us assume that the trees S and S' correspond with inputs (M, A) and (M', A'), respectively, and that S' is a subtree of S. 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 nodes $Z_{\alpha||\beta}$.

This property is not present in a random oracle. It can be avoided in several ways, such as fixing the topology of the trees. However, the simplest method is to have domain separation between final and inner nodes: this makes $S'_* = S_\alpha$ impossible as they are in different domains. This leads to the following condition:

Condition 4 *$\mathcal{Z}$ enforces domain separation between final and inner nodes.*

A simple way to implement domain separation is to start (or end) 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. In our indifferentiability proof there is however no restriction to the output length of this function.

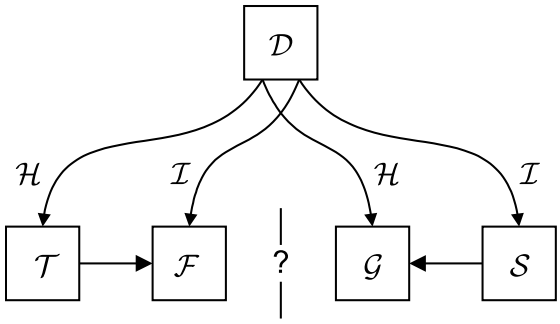

Fig. 1. The differentiability setup

4 The indifferentiability framework

The indifferentiability framework was introduced by Maurer et al. in [10] and 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 [7]. 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 towards 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 [2]. A random oracle, denoted $\mathcal{RO}$, takes as input binary strings of any length and returns for 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.

4.1 The distinguisher's setting

We study the indifferentiability 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 in Figure 1. 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, ℓ) with M a binary string, i.e., $M \in \mathbf{Z}_2^*$, A the value of the mode parameters and ℓ the requested output length, and returning a binary string $y \in \mathbf{Z}_2^\ell$;
- $\mathcal{I}$ taking as input (s, ℓ) with s a binary string $s \in \mathbf{Z}_2^*$ and ℓ 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, ℓ), the left system returns $y = \mathcal{T}[\mathcal{F}](M, A)$ truncated to ℓ bits. When queried at the interface $\mathcal{I}$ with a query (s, ℓ), it returns $t = \mathcal{F}(s)$ truncated to ℓ bits.

The system at the right consists of an ideal hash function $\mathcal{G}$ implementing the interface $\mathcal{H}$ and of a simulator $\mathcal{S}$ implementing the interface $\mathcal{I}$. When queried with a query (M, A, ℓ), $\mathcal{G}$ returns $y = \mathcal{G}(M, A)$ truncated to ℓ bits. From the requirement that ideally the outputs corresponding to

two different inputs (M, A) and (M', A') are independent, we construct this ideal hash function as $\mathcal{G} = \mathcal{RO} \circ \mathcal{E}$, where $\mathcal{E}$ is a (deterministic) *encoder* and $\mathcal{RO}$ a random oracle. The encoder $\mathcal{E}$ converts the input (M, A) to a binary string s in an injective way, which is then transmitted to the random oracle.

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

Indifferentiability of $\mathcal{T}[\mathcal{F}]$ from the ideal function $\mathcal{G}$ 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 $(\mathcal{G}, \mathcal{S}[\mathcal{G}])$. 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 $Q_{\mathcal{I},i} = (s_i, \ell_i)$.

For a given set of queries Q and their responses $\mathcal{X}(Q)$, we define the *$\mathcal{T}$-consistency* as the property that the responses to the $\mathcal{H}$ interface are equal to those that one would obtain by applying the tree hashing mode $\mathcal{T}$ to the responses to 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.

4.2 The cost of queries

The indifferentiability bounds provided in [7] are expressed as a function of the total number of queries q and their maximum input lengths. In [4] 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 $\mathcal{Z}(|M|, A)$.

In addition, the cost does not 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 can be replaced by 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 can be replaced by a single query $(M_i, A_i, \max(\ell_i, \ell_j))$.

4.3 Definition

We can now adapt the definition as given in [7] to our setting.

Definition 4 ([7]). *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, ϵ)-indifferentiable from an ideal hash function $\mathcal{G}$ if there exists a simulator $\mathcal{S}[\mathcal{G}]$, such that for any distinguisher $\mathcal{D}$ it holds that:*

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

The simulator has oracle access to $\mathcal{G}$ 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. Similarly, $\mathcal{T}[\mathcal{F}]$ is said to be indifferentiable from $\mathcal{G}$ if ϵ is a negligible function of the security parameter n.

5 Indifferentiability proof

In this section, we always assume that the conditions presented in Section 3 are fulfilled by the tree hashing mode $\mathcal{T}$. We first describe the simulator and its general goal. We then prove the indifferentiability results by means of a series of lemmas and a final theorem.

5.1 The simulator

The simulator $\mathcal{S}[\mathcal{G}]$ has two tables. First, it keeps track of the queries in a list T of couples (s, c) with $s, c \in \mathbf{Z}_2^*$. Second, it memorizes a set $P_n \subseteq \mathbf{Z}_2^n$ of n-bit values that the simulator cannot use as an output to $Q_{\mathcal{I}}$ queries for inner nodes. Both T and P_n are initialized to the empty set.

We say that a final node instance s is *message-bound* if the table T allows to reconstruct the corresponding input (M, A).

Algorithm 1 $\mathcal{T}$-decoding

input: S_* and table T
output: message M and tree parameters A, or status code
for α in any fixed order such that a node S_α is always before its sons **do**
  if S_α has formatting compliant with $\mathcal{T}$ and Z_β for all β prefix of α **then**
    Decode S_α into Z_α
  else
    return "invalid coding"
  end if
  Use the message bits in S_α to build M
  for each block c_j of n chaining bits from the son number j **do**
    if $(s, t) \in T$ and the first n bits of t are equal to c_j **then**
      Set $S_{\alpha||j} \leftarrow s$
    else
      return "dead end at c_j"
    end if
  end for
end for
Build A given the complete tree
return (M, A)

Algorithm 1 implements the reconstruction on the complete tree instance from the final node instance and using the known pairs (s, c) in T. When s is message-bound, it returns the corresponding M and A; in this case, Conditions 1, 2 and 3 imply that this algorithm always succeeds (see also Lemma 3). If not, it returns a chaining value whose preimage could not be found in the table T. With a (partial) tree instance that could not have been produced by the tree hashing mode, an error is returned.

Algorithm 2 implements the simulator. It stores in T the queries and their responses as (s, c) couples, and use the c part (possibly truncated) to build the response for a subsequent query with the same s. As a general rule, if two queries (s, ℓ_1) and (s, ℓ_2) are sent with $\ell_2 > \ell_1$, the second query only extends the c part to ℓ_2 bits.

Depending on the type of queries, the simulator takes the following actions:

- For inner node instances, the simulator avoids collisions in the first n output bits using the set P_n. This set is built from this type of queries, and also using the final nodes that are not message-bound (see below).
- For final node instances:
  - When s is message-bound, it calls $\mathcal{G}$ for guaranteeing $\mathcal{T}$-consistency.
  - When s is not message-bound, it returns random bits. Furthermore, the simulator makes certain that such an s not message-bound does not become message-bound later on; it would otherwise break $\mathcal{T}$-consistency. This is achieved by adding to P_n a chaining value not known during the $\mathcal{T}$-decoding of s; such a chaining value will therefore never be returned again for an inner node instance and s can never become message-bound (see also Lemma 4).

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

```
 1: input: (s, ℓ) (interface I)
 2: output: string in Z_2^ℓ
 3: Let t such that (s, t) ∈ T, or t = ∗, the empty string, if no such (s, t) exists in T
 4: if |t| < ℓ then
 5:    if s is a final node instance then
 6:       T-decode s using T
 7:       if decoding returned (M, A) then
 8:          Set t to the first ℓ bits of G(M, A)
 9:       else if decoding returned "dead end at e" then
10:          Set P_n ← P_n ∪ {e}
11:       end if
12:    else {s is an inner node instance}
13:       if |t| < n then
14:          Choose t randomly and uniformly from Z_2^n \ P_n
15:          Set P_n ← P_n ∪ {t}
16:       end if
17:    end if
18:    Append max(ℓ − |t|, 0) uniformly and independently drawn random bits to t
19:    Add (s, t) to the table T, replacing any previous entry whose first component is s
20: end if
21: return t truncated to its first ℓ bits
```

5.2 The proof

Lemma 2. *If $P_n \neq \mathbf{Z}_2^n$, the simulator avoids collisions in the first n output bits when queried with inner node instances.*

Proof. As it can be seen in lines 13–16 of Algorithm 2, the simulator stores the first n bits of the output in P_n and avoids returning any value in P_n for any other inner node instance. □

Lemma 3. *If $P_n \neq \mathbf{Z}_2^n$, the reconstruction of (M, A) from a message-bound final node is possible and there is only one possible (M, A).*

Proof. First, note that when queried with the same inner node instance but different lengths, (s, ℓ_1) and (s, ℓ_2), the simulator returns consistent values in the first $\min(\ell_1, \ell_2)$ bits. In line 13 of Algorithm 2, the condition $|t| < n$ is actually equivalent to $t = *$, as line 14 always generates n bits, so these n bits are never regenerated for the same s. And line 19 keeps in memory the longest response given so far; consequently, any pair (s, c) in T can only grow by appending bits to the c component.

The $\mathcal{T}$-decoding algorithm depends only on answers to inner node instances, whose consistence is shown above. Furthermore, together with Lemma 2, the pair (s, t) at line 11 of Algorithm 1 is either not found or can be retrieved without ambiguity. □

Lemma 4. *Given queries $Q_{\mathcal{I}}$ to the simulator $\mathcal{S}[\mathcal{G}]$ described in Algorithm 2 and $Q_{\mathcal{H}}$ to $\mathcal{G}$, it returns $\mathcal{T}$-consistent responses, unless $P_n = \mathbf{Z}_2^n$.*

Proof. The proof is by induction. We assume that the simulator has received a sequence of queries and that up to now it has returned $\mathcal{T}$-consistent responses. Initially this is the case when the simulator has not received any queries at all. We will now prove that if $P_n \neq \mathbf{Z}_2^n$, the simulator will return a response such that the whole set of queries and responses remain $\mathcal{T}$-consistent.

Thanks to the domain separation between final and inner nodes, the simulator can distinguish between these two kinds of queries and we can consider these two cases separately. To rephrase the definition, $\mathcal{T}$-inconsistency would imply that there is a query to the simulator for a final node instance S_* that is message-bound to (M, A) and such that its response $\mathcal{S}[\mathcal{G}](S_*)$ is different from $\mathcal{G}(M, A)$.

When querying a final node S_* and it is message-bound, Lemma 3 shows that (M, A) is unambiguously retrieved and $\mathcal{S}$ queries $\mathcal{G}(M, A)$ to make its response $\mathcal{T}$-consistent per construction (line 8 of Algorithm 2). When querying a final node and it is not message-bound, there is no information in the $Q_{\mathcal{I}}$ queries and their responses to show the inconsistency.

When querying an inner node, $\mathcal{T}$-inconsistency cannot be shown using the response to this very query, as it cannot be compared to a query to the $\mathcal{H}$ interface. However, the simulator could result in $\mathcal{T}$-inconsistency if this new response would make the response t to (s, ℓ), a previously queried final node, $\mathcal{T}$-inconsistent. Now there are two possibilities:

- s was message-bound to some input (M, A) when the query was sent. In this case, $\mathcal{T}$-inconsistency could only come from an inner collision that would allow to $\mathcal{T}$-decode the final node s to another input $(M', A') \neq (M, A)$. But Lemma 1 shows that this would imply a collision in the first n output bits for inner nodes, and this is not possible thanks to Lemma 2.
- s was not message-bound when the query was sent. In this case, $\mathcal{T}$-inconsistency could only come from the final node s becoming message-bound due to the new query. However, for every such final node s, a chaining value c was added to P_n, preventing s from becoming message-bound in later queries.

It follows that the simulator guarantees $\mathcal{T}$-consistency for all queries Q unless $P_n = \mathbf{Z}_2^n$. □

Lemma 5. *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 6. *The advantage of an adversary in distinguishing between $\mathcal{F}$ and $\mathcal{S}[\mathcal{G}]$ 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 advantage is defined as $\mathrm{Adv}(\mathcal{A}) = |\Pr[\mathcal{A}[\mathcal{F}] = 1] - \Pr[\mathcal{A}[\mathcal{S}[\mathcal{G}]] = 1]|$. We provide an upper bound of the advantage by computing the variational distance between the two statistical distributions. Actually, we are interested only in the first n output bits of the responses; the other bits are uniformly and independently generated by $\mathcal{S}$ and $\mathcal{F}$ in all cases.

Since $\mathcal{F}$ is a random oracle, the responses to q different queries are independent and uniformally distributed over $\mathbf{Z}_n^2$. By inspecting Algorithm 2, the simulator always returns uniform values for final node instances. For inner node instances, the simulator chooses it from the set $\mathbf{Z}_2^n \setminus P_n$. Each query can add at most one element to P_n. 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, 3 and 4, is (t_D, t_S, q, ϵ)-indifferentiable from an ideal hash function, for any t_D, $t_S = O(q^3)$, $q < 2^n$ and any ϵ with $\epsilon > \epsilon_n(q)$.*

Proof. We consider a more powerful adversary 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 5 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 5, 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 $P_n \neq \mathbf{Z}_2^n$ in the simulator even after issuing the free extra queries $Q'_{\mathcal{I}}$. As a consequence, Lemma 4 guarantees $\mathcal{T}$-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 query $\overline{Q}_{\mathcal{I}}$, we look at the problem of distinguishing the random variable $\mathcal{F}(\overline{Q}_{\mathcal{I}})$ from the random variable $\mathcal{S}[\mathcal{G}](\overline{Q}_{\mathcal{I}})$. From Lemma 6, the advantage is upper bounded by $\epsilon_n(q)$.

We have $t_S = O(q^3)$ as for each of the q queries, the simulator may have to $\mathcal{T}$-decode q node instances, each requiring to look up in a table of at most q entries. □

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 $\epsilon_n(q)$:

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

6 Choice of $\mathcal{F}$ and its implications

The proof given in Section 5 assumes that $\mathcal{F}$ is a random oracle. In any concrete design $\mathcal{F}$ must naturally be a concrete hash function. The main purpose of this paper is to provide a means to

introduce tree hashing modes based on a sequential hash function $\mathcal{F}$. There are several constructions for building a sequential hash function using a random compression function [7] or a random permutation [4] that come with an indifferentiability proof. The differentiating advantage for these bounds are typically 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 (usually related to the internal state size).

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 just the sum of the outer advantage $q^2/2^{n+1}$ and the inner advantage $N^2/2^{c+1}$. To measure the cost of an adversary, we choose as unit the evaluation of the f function (typically a compression function or a permutation) 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 [3,4]. Assume we have a permutation f and we want to limit the total differentiating advantage to $N^2/2^{c'+1}$. 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 $n = c' + 1$.

With a tree hashing mode, a sequential hash function mode with an indefinite output length, such as the sponge construction, may come in handy. For a hash function with digest size m, (second) preimage resistance of 2^m requires that $c \geq 2m$. On the other hand, an optimal choice of parameters as above requires that $n = c \geq 2m$ output bits are used as chaining values, more than the available digest size m.

Another approach is to use the tree hashing mode calling a FIL hash function, where the FIL hash function is assumed to behave as a 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 other hand, it has been shown in [12,15] that a random permutation can be converted to a FIL random oracle simply by fixing part of its input and truncating its output.

7 Property preserving aspects

Independently of the indifferentiability result, some properties hold when the four conditions of Section 3 are satisfied.

First, 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 collision resistance of the outer hash function preserves that of the inner one.

Then, a similar idea applies to the (second) 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_*. He is thus able to find a (second) preimage on the inner hash function $\mathcal{F}$.

8 Applicability

We first show that, given a tree hashing scheme, it is easy to verify that it satisfies the four conditions of Section 3.

There are a few cases where Condition 1 can be directly checked. For instance, the structure of the tree can be fixed, or it is determined by the parameters A encoded as frame bits in the final node. In these cases, the knowledge of S_* is enough to decode the whole tree and the condition is immediately verified.

It is also possible that the structure of the tree is determined by S_* up to the degree of the nodes. In Section 8.2, we will see an example where the degree of the final node grows with the message size. In another example, the number of message bits in a leaf is variable. In both cases, Condition 1 comes down to checking that the size of the node instances allows to fully determine the tree structure.

In general, one has to start verifying Condition 1 by checking that the final node can be decoded, for all (M, A). Then, one follows the definition, by checking that the knowledge of a node instance and its parents allows to decode its sons. Finally, the leaves must be identified as such and the decoding must terminate.

Verifying Condition 2 is a matter of checking that all message bits are processed in some node, for all $(|M|, A)$. Condition 3 usually means that A can be recovered from the tree topology or from its encoding in frame bits, and checking Condition 4 follows directly from the tree hashing scheme definition.

8.1 Sequential hashing

Sequential hash functions can be seen as a special case of tree hashing modes, where the tree reduces to a single linear sequence. Therefore, the conditions for tree hashing modes introduced in Section 3 can be applied to sequential hash functions. Consider the following sequential hash mode that calls an $n + m$-bit to n-bit hash function, which can be seen as strengthened Merkle-Damgård hashing with final node domain separation added to it.

- Append to the message M its length coded on 64 bits.
- Pad it with a bit 1 and at most $m - 2$ bits 0 to form a sequence of y $m - 1$-bit blocks M'_i.
- Append a single 0 to each of the blocks except the last one, for which 1 is appended.
- Let $H_{i+1} = \mathcal{F}(H_i, M'_i)$ with $H_0 = 0^n$, and the hash result is defined as H_y.

This hashing mode obviously satisfies M-completeness, A-completeness (as there are not tree parameters) and final node domain separation. Template-decodability finally can be easily shown. All inner nodes have a similar template, except of the first node, which has frame bits in its first n bits instead of chaining bits. The length coded at the end of the message determines the height of the tree and thereby allows to identify the first node. It follows that differentiating this mode has advantage at most $q(q-1)/2^{n+1}$, where q is the number of queries to the compression function.

As a variant, consider this mode where M'_0 has length $n + m$ rather than m and where $H_1 = \mathcal{F}(M_0)$. In the first node, the n frame bits are replaced by n message bits. This satisfies again all four criteria and hence satisfies the same indifferentiability bound and results in n more bits hashed with the same effort.

In another variant, we omit the length coding at the end of the message and extend the domain of the compression function with the empty string $*$. We define $\mathrm{IV} = \mathcal{F}(*)$ and let $H_0 = \mathrm{IV}$. All the inner nodes have n chaining bits and m message bits, except the (single) leaf, which has an empty template, and $\mathcal{F}(*)$ is given as chaining value to its parent, together with the first message

block. Since the leaf is clearly identifiable by its length, the template is decodable. Also, it is easy to show that this mode satisfies the other three conditions.

Note that in this last variant, the IV is not part of the mode definition but is put as a public parameter together with the compression function. In other words, the distinguisher does not know the value of the IV in advance but has to query it to the $\mathcal{I}$ interface. In this extended model of the underlying compression function, the length coding can simply be omitted.

To avoid extending the domain of the compression function, it is also possible to cover this case by adapting the simulator. It suffices to initialize the set P_n to {IV}. The proof remains valid except that the bound becomes $q(q+1)/2^{n+1}$ instead of $q(q-1)/2^{n+1}$. In this case, the IV is encoded as frame bits instead of chaining bits.

A similar adaptation to our simulator suffices to extend our proof to also cover the enveloped Merkle-Damgård (EMD) transform [1]. The EMD transform is obviously A-complete (as there are no parameters) and M-complete. Additionally, it is template-decodable as the coding of the message length in the final node allows distinguishing the first call to f from the other ones. Domain separation between final and inner nodes is achieved on the condition that the chaining value never collides with IV2. This can simply be achieved in the simulator by initializing the set P_n to {IV2} resulting in the bound $q(q+1)/2^{n+1}$. If we further modify the simulator by initializing the P_n to $\{\text{IV1}, \text{IV2}\}$, template-decodability can be based on the presence of IV1 and no longer requires the length-coding in the input to the final node. This goes only at the expense of a slight deterioration of the bound to $(q+1)(q+2)/2^{n+1}$.

Note that in all these cases our proof requires $\mathcal{F}$ to be a (FIL) random oracle and it does not cover the case of $\mathcal{F}$ being an ideal cipher in Davies-Meyer mode. In [12,15] an indifferentiable construction is provided to construct a FIL random oracle from a random permutation.

8.2 Two simple examples

We now present two simple examples of tree hashing modes. These two modes are also discussed in [5], 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).

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 bits. 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_α consist of B message 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 bits. The tree is a balanced tree of height H and all (non-leaf) nodes have degree D. The leaf nodes Z_α, $\alpha = \alpha_0||\alpha_1||\ldots||\alpha_{H-1}$, 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 the number of parallel processes is know in advance. The compression functions on each of the D^H leaves can be fetched with B-bit blocks in parallel.

Both modes are clearly M-complete and A-complete (as A is encoded in the final node). Moreover, they implement final node separation. Additionally, they are template-decodable as the tree structure can be fully determined from A encoded in the final node and from the length of the node instances.

8.3 Taking the union of tree hashing modes

We can take the union $\mathcal{T}_{\text{union}}$ of n tree hashing modes $\mathcal{T}_i$ in the following way. A_{union} is given by a choice parameter indicating the mode i composed with the tree parameters A_i for the particular mode. In the union mode it is sufficient to additionally code in the final node for each of the modes the choice parameter i such that it can be uniquely decoded in order to preserve soundness.

For instance, taking the union of the two modes presented in Section 8.2 simply takes the addition of a binary choice parameter and coding it as a frame bit in the final node right before the final bit.

Conversely, restricting the range of tree parameters of a given tree hashing mode does not impact its soundness. When fixing the value of the tree parameters of a sound tree hashing mode to a single value, it becomes indifferentiable from a random oracle (i.e., there is no need for the encoder $\mathcal{E}$ anymore).

References

1. 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.
2. 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.
3. 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`.
4. ———, *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.
5. ———, Keccak *sponge function family main document*, NIST SHA-3 Submission (updated), January 2009, `http://keccak.noekeon.org/`.
6. ———, *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.
7. J. Coron, Y. Dodis, C. Malinaud, and P. Puniya, *Merkle-Damgård revisited: How to construct a hash function*, Advances in Cryptology – Crypto 2005 (V. Shoup, ed.), LNCS, no. 3621, Springer-Verlag, 2005, pp. 430–448.
8. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, *The Skein hash function family*, Submission to NIST, 2008, `http://skein-hash.info/`.
9. J. W. Martin, *ESSENCE: A candidate hashing algorithm for the NIST competition*, Submission to NIST, 2008, `http://www.math.jmu.edu/~martin/essence/Supporting_Documentation/essence_NIST.pdf`.
10. U. Maurer, R. Renner, and C. Holenstein, *Inidifferentiability, 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.
11. R. C. Merkle, *Secrecy, authentication, and public key systems, PhD thesis*, UMI Research Press, 1982.
12. 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/`.
13. 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/`.
14. M. Torgerson, R. Schroeppel, T. Draelos, N. Dautenhahn, S. Malone, A. Walker, M. Collins, and H. Orman, *The SANDstorm hash*, Submission to NIST, 2008, `http://www.sandia.gov/scada/documents/SANDstorm_Submission_2008_10_30.pdf`.
15. R. Rivest Y. Dodis, L. Reyzin and E. Shen, *Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to MD6*, Fast Software Encryption 2009, 2009.