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 and output lengths, 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 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 three simple conditions for such a (either sequential or tree) hashing mode to be *sound*. These conditions are easy to implement and to verify, and can for instance be used by the practitioner to build a tree hashing mode on top of an existing hash function. We provide a proof that for any hashing mode satisfying these three conditions, the advantage in differentiating it 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 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 function, hierarchic 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 [16], and in [12] 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) 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.

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 [15] and applied to hash functions by Coron et al. in [10]. 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 [18] that the indifferentiability framework covers only so-called single-stage games. So there are use cases (i.e., multi-stage games) 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 he 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 $Z = \mathrm{Hash}(C||\mathrm{File})$ instead, but it nevertheless calls for caution when plugging in a sequential (or tree) hash function in a protocol secure in the random oracle model. Fortunately, indifferentiability still covers the security with respect to the classical criteria of (second) preimage resistance, collision resistance, the numerous variants of these, and even security in keyed modes.

Clearly, 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 the simple conditions we formulate also apply to sequential hashing modes.

1.1 Previous work

Provable security of tree hashing was already investigated in [20] and upper bounds on the differentiating advantage have been given, e.g., for the mode used in MD6 [19]. This present paper was inspired by the proofs in [19], 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 [7]. In the meanwhile the authors of [19] 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 [13].

1.2 Our contributions

Despite the similarity between the conditions in this paper and those in [13], this present paper has a substantial added value with respect to prior work. First of all, we provide a set of conditions that are strictly more general than those of [13], hence allowing more freedom in the definition of a sound tree or sequential hashing mode. We provide a detailed comparison in Section 4.2.

Second, we cover the case of inner hash functions with variable input length, while [13] only covers modes on top of a compression function. Our goal is to allow taking an existing hash function as inner hash function and building a tree mode on top of it. This modularity can play an important role in practice, e.g., in standardization of tree hashing modes working on top of existing hash functions.

Third, we allow both inner hash function and outer hash functions have indefinite output length, unlike [13]. Compared to fixed output length cases—and where obviously the output of the outer hash function is limited in length by that of the inner hash function—this allows us to prove an upper bound on the differentiating advantage from an random oracle with indefinite output length, rather than a truncated random oracle. This also finds itself useful in some use cases such as full domain hashing [4,11]. Moreover, the variable output length of the inner hash function allows a better trade-off between security and efficiency, as we will discuss in Section 7.1.

Fourth, the bound we achieve on the success probability of differentiating tree hashing modes from an ideal hash function is as tight as theoretically possible (see Section 3). The bound in [13] 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 [19] and $r = 3$ in SHA-2 [17]).

Finally, we can also apply our fairly general treatment of tree hashing modes to the specific case of sequential modes. In addition to shedding a new light on these modes, our general theorem yields improved bounds for some of them (see Sections 8.4 and 8.6).

1.3 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 soundness 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 [13]. After adapting the indifferentiability setting of [10] 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 mode

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 and arbitrary 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 (see, e.g., Section 7.3). 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 for the tree hashing mode.

2.1 Hashing as a two-step process

The tree hashing mode specifies for any given parameter choice and message length 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 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 specifies for a call to $\mathcal{F}$ how the input must be constructed from message bits and 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 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 $h = \mathcal{F}(S_*)$.

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_α, where α denotes its index. Similarly, node instances are denoted by S_α. 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 α by $\text{parent}(\alpha)$. (We assume that the node indexes α 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 α 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_α is an *ancestor* of a node Z_β if $\alpha = \text{parent}(\beta)$ or if Z_α is an ancestor of the parent of Z_β. In other words, Z_α is a parent of Z_β 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_β is a *descendent* of Z_α and d is the *distance* between Z_α and Z_β. Clearly, the final node has no ancestors and a leaf node has no descendents.
- Every node Z_α is a descendent of the final node and the distance between the two is called the *height* of α. 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 descendent. 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 descendents. 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_α 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_α 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 β 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 β 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.)

We refer to Section 7.3 for two simple examples illustrating these concepts. 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_α is executed only after its children are. This results in a tree instance S with nodes S_α.

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 α, we construct pairs of messages that differ only in bits that are mapped to Z_α and its descendents (e.g., if Z_α 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 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 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 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 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 equals 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 1. *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 $S_* = S'_*$.*

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 $\mathcal{T}[\mathcal{F}]$. On the other hand, 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 nodes Z_α we have $Z_\alpha = Z'_\alpha$, except for a particular node Z_β and its descendents. Node Z_β has one descendent Z_γ and Z'_β is a leaf node. Additionally, Z_β and Z'_β have the same length and in the positions where there are chaining pointer bits in Z_β, there are message pointer 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 its descendent S_γ 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 2, Appendix B.

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

Definition 2. *A mode of use $\mathcal{T}$ is* tree-decodable *if for all tree instances S generated with $\mathcal{T}$ (i.e., there exists an input (M, A) and a function $\mathcal{F}$ such that $S = \mathcal{Z}[\mathcal{F}](M, Z)$ with $Z = \mathcal{T}(|M|, A)$), given any proper final subtree S_J, one can identify at least one node index $\beta \notin J$, the chaining pointer bits in S_J with child index β and their position attribute. We call the index β an* expanding index *of S_J.*

This definition includes the case where, given just a node instance, one can identify the chaining values and their attributes in that node, or the case where this is possible given the 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 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. 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 β with $S_\beta \neq S'_\beta$. We have that $S_{\text{parent}(\beta)} = S'_{\text{parent}(\beta)}$ and the chaining values with child index β are fully determined by S_J. It follows that $\mathcal{F}(S_\beta) = \mathcal{F}(S'_\beta)$ and hence we have an inner collision.

We must now prove that there exists a set J such that $S_J = S'_J$ and it has an expanding index β 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 β. If there exists an expanding index β such that $S_\beta \neq S'_\beta$ we have found our J. Otherwise, we expand J by adding β: $J = J \cup \{\beta\}$. Clearly, this J again defines a final subtree with $S_J = S'_J$. This process can only stop in two ways: either for the current set J an expanding index β is found with $S_\beta \neq S'_\beta$, or J covers all nodes indices of S and S'. In the latter case $S = S'$ 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 3. *A mode of use $\mathcal{T}$ is* message-complete *if for any tree instance S generated with $\mathcal{T}$, one can fully determine the input message M.*

Message-completeness implies that for every message length $|M|$ and every parameter value A, one can determine from S the position attribute of all message pointer bits in $Z = \mathcal{T}(|M|, A)$ and every position i in $[0, |M| - 1]$ occurs at least once in Z.

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_α and all its descendents. Second, there is a one-to-one mapping ψ 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 3, Appendix B.

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 3 *$\mathcal{T}$ enforces domain separation between final and inner nodes. In other words, $\mathcal{T}$ is such that for any (M, A) and (M', A') and for any node index $\alpha \neq *$ in $\mathcal{T}(|M|, A)$ we have $S_* \neq S'_\alpha$, where S and S' correspond with inputs (M, A) and (M', A'), respectively.*

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 proof there is however no restriction to the output length of this function.

4.1 Applicability

We show that, given a tree hashing scheme, it is easy to verify that it satisfies the three conditions.

There are a few cases where tree-decodability 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 tree-decode the whole tree instance 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 7.3, we will see an example where the degree of the final node grows with the message size. In another example, the number of message pointer bits in a leaf is variable. In both cases, tree-decodability comes down to checking that the size of the node instances allows to fully determine the tree structure.

Verifying tree-decodability starts with checking that the final node can be decoded, for all (M, A). Then, one follows the definition, by checking that for each final subtree one can find an expanding index. Finally, the leaves must be identified as such and the decoding must terminate.

Verifying message-completeness consists of checking that all message bits are processed in some node, for all $(|M|, A)$ and that the size of nodes and their frame bits allow to determine the position attribute of all message pointer bits.

For some concrete examples, we refer to Sections 7.3 and 8.

4.2 Comparison with tree based modes of Dodis et al.

The five "required properties of Mode of Operation" listed in [13] correspond to a large extent with our three condition, but not quite.

First, the condition *unique parsing* is similar to our condition tree-decodability. However, according to the definition in [13], 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 β 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 domain separation between final node and inner nodes.

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 [13] there 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 ζ 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 ζ above) to the output also preserves it.

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

4.3 Property preserving aspects

Independently of the bounds proved below, some properties hold when the three conditions of Section 4 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 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 The indifferentiability framework

The indifferentiability framework was introduced in [15] by Maurer et al. 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 [10]. 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 [3]. 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.

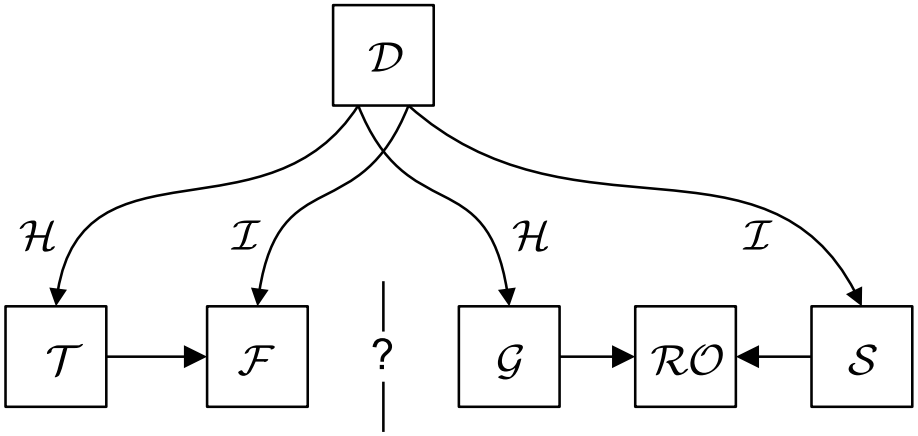

Fig. 1. The differentiability setup

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 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 $\mathcal{I}$. We define the ideal hash function as $\mathcal{G}[\mathcal{RO}]$, where $\mathcal{RO}$ is a random oracle. Upon receiving a query (M, A, ℓ), $\mathcal{G}$ constructs the tree template from the parameter value and message length $Z = \mathcal{T}(|M|, A)$ and encodes it together with the message M in an injective way, denoted by $\mathrm{enc}(Z, M)$. 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, ℓ), 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 $Q_{\mathcal{I},i} = (s_i, \ell_i)$.

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.

5.2 The cost of queries

The differentiability bounds in [10] are expressed as a function of the total number q of queries and their maximum input lengths. In [6] 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 [10] to our setting.

Definition 4 ([10]). *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 $\mathcal{G}[\mathcal{RO}]$ 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 ϵ 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 [6].

6.1 The simulator and input extraction

The simulator $\mathcal{S}[\mathcal{RO}]$ 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 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 reconstructing the complete tree instance using the known pairs (s, c) in T. When s is message-bound, it returns the corresponding M and Z; in this case, Conditions 1 and 2 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 (line 9). With a tree instance that could not have been produced by the tree hashing mode, an error is returned (line 13).

Algorithm 2 implements the simulator. It stores in T the queries and their responses as (s, c) couples, and uses 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, it avoids collisions in the first n output bits using the set P_n (lines 14-15). This set is built from inner node queries and final node queries that are not message-bound (see below).
- For final node instances:
  - When s is message-bound, it calls $\mathcal{RO}$ to guarantee $\mathcal{T}$-consistency (line 8).
  - When s is not message-bound, it returns random bits. Furthermore, the simulator assures that s does not become message-bound later on; this would break $\mathcal{T}$-consistency. This is achieved by adding to P_n a chaining value not known during the $\mathcal{T}$-decoding of s (line 10); 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 1 $\mathcal{T}$-decoding

input: s and table T
output: message M and tree template Z, or status code
Initialization: $J = \{*\}$ and $S_* = s$
while S_J has an expanding index β **do**
  Let c be the chaining value corresponding to β extracted from S_J
  if there is an entry $(s', t) \in T$ with the first n bits of t equal to c **then**
    Let $J = J \cup \{\beta\}$ and $S_\beta = s'$
  else
    return "dead end at c"
  end if
end while
if S_J does not have valid coding to be a tree instance generated with $\mathcal{T}$ **then**
  return "invalid coding"
else
  Extract the tree template Z and the message M from the tree instance S_J
  return (M, Z)
end if

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

input: (s, ℓ) (interface $\mathcal{I}$)
output: string in $\mathbf{Z}_2^\ell$
Let t such that $(s, t) \in T$, or $t = *$, the empty string, if no such (s, t) exists in T
if $|t| < \ell$ **then**
  if s is a final node instance **then**
    $\mathcal{T}$-decode s using T
    if decoding returned (M, Z) **then**
      Set t to the first ℓ bits of $\mathcal{RO}(\mathrm{enc}(M, Z))$
    else if decoding returned "dead end at e" **then**
      Set $P_n \leftarrow P_n \cup \{e\}$
    end if
  else {s is an inner node instance}
    if $|t| < n$ **then**
      Choose t randomly and uniformly from $\mathbf{Z}_2^n \setminus P_n$
      Set $P_n \leftarrow P_n \cup \{t\}$
    end if
  end if
  Append $\max(\ell - |t|, 0)$ uniformly and independently drawn random bits to t
  Add (s, t) to the table T, replacing any previous entry whose first component is s
end if
return t truncated to its first ℓ bits

6.2 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, Z) from a message-bound final node is possible and there is only one possible (M, Z).*

Proof. First, 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 6 of Algorithm 1 is either not found or can be retrieved without ambiguity. □

Lemma 4. *Given queries $Q_\mathcal{I}$ to the simulator $\mathcal{S}[\mathcal{RO}]$ described in Algorithm 2 and $Q_\mathcal{H}$ to $\mathcal{G}[\mathcal{RO}]$, 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 the 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, Z) and such that its response $\mathcal{S}[\mathcal{RO}](S_*)$ is different from $\mathcal{RO}(\text{enc}(M, Z))$.

When querying a final node S_* and it is message-bound, Lemma 3 shows that (M, Z) is unambiguously retrieved and $\mathcal{S}$ queries $\mathcal{RO}(\text{enc}(M, Z))$ 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 couple (M, Z) 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 couple $(M', Z') \neq (M, Z)$. 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. Here $\mathcal{T}$-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 $\mathcal{S}[\mathcal{RO}]$ 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 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 and 3, 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 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 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 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 6, upper bounds the advantage to $\epsilon_n(q)$.

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 $\epsilon_n(q)$:

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

7 Application to tree hashing

One can build a tree hashing mode calling a compression function, where it is assumed to behave as a finite-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 other hand, it has been shown in [19,13,6] that a random permutation can be converted to a FIL random oracle simply by fixing part of its input and truncating its output.

In the following subsection we discuss the alternative 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.

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 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 [5,6]. 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 Advantage of variable output length

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.

When using an inner hash function with output length fixed to m bits, extra steps must be taken to output $2m$ bits in the inner nodes, hence making the construction globally more complex and less efficient.

7.3 Two simple examples

We now present two simple examples of tree hashing modes. These two modes are also discussed in [8], 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 $\alpha = \alpha_0 || \ldots || \alpha_{h-1}$ 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_α 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_α 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 know in advance. The compression functions on 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 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.

7.4 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 (domain separation) in order to preserve soundness. Note that tree decodability does not necessarily imply explicit coding of all parameter values in the final (or any other) node.

For instance, taking the union of the two modes presented in Section 7.3 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 more need for the encoder $\mathcal{E}$, see Section 5.1).

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 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, and the case of $\mathcal{F}$ being an ideal cipher in Davies-Meyer mode is not covered. In [19,13] a construction is provided to construct a FIL random oracle from a random permutation.

8.1 Satisfying tree-decodability

Tree-decodability implies classifying any given node instance as the leaf node, the final node or the pre-final 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 template-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 an 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 [12], 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, [12] 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. The problem 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 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 [10] 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

Assume for tree-decodability 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 μ 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 ℓ the number of requested bits, and hence the mode of use is differentiated. Actually Lemma 4 does not hold due to the fact that it relies 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 is problematic for Lemma 2 as this may result in an inner collision. One can see why by inspecting the simulator in Algorithm 2. Upon receipt of a query with an inner node instance, the set P_n is used to avoid inner collisions in line 14. Upon receipt of a query with a final node instance (or erroneously identified as such) there are two possibilities: either the node is message-bound and it queries $\mathcal{RO}$ for the response, or it generates a random response (line 18). In either case, elements in P_n are not excluded as possible responses and hence if the query is an inner node erroneously identified as a final node, inner collisions may be generated. It is also problematic for Lemma 4 as there it is essential to distinguish between inner nodes and final nodes. However, these problems disappear when IV2 is included initially in P_n: the presence of IV2 now guarantees that in line 5 of Algorithm 2 an inner node instance is always correctly recognized as such.

So, the initialization of P_n to the set of IV values fixes Lemma 2, Lemma 3 and Lemma 4 and has no impact on Lemma 5. However, it does have an impact on the bound in Lemma 6. 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:

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

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.

On the other hand, 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 [2], 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 [2].

The EMD transform can be seen as an improved version of two modes previously proposed in [10]. These modes are called NMAC and HMAC respectively and are inspired by the MAC function constructions with the same name published in [1]. 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 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 [9] 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 non-zero constant X. This method was proven indifferentiable in [14]. 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 [14]). 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).

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 [10] 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).

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. 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.
2. 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.
3. 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.
4. ———, *The exact security of digital signatures - how to sign with RSA and Rabin*, Advances in Cryptology – Eurocrypt '96 (U. M. Maurer, ed.), Lecture Notes in Computer Science, vol. 1070, Springer, 1996, pp. 399–416.
5. 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`.
6. ———, *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.
7. ———, *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.
8. ———, Keccak *sponge function family main document*, NIST SHA-3 Submission (updated), June 2010, `http://keccak.noekeon.org/`.
9. 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.
10. 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.
11. J.-S. Coron, *On the exact security of full domain hash*, Crypto (M. Bellare, ed.), Lecture Notes in Computer Science, vol. 1880, Springer, 2000, pp. 229–235.
12. 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.
13. 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.
14. S. Hirose, J. Park, and A. Yun, *A simple variant of the Merkle-Damgård scheme with a permutation*, Asiacrypt, 2007, pp. 113–129.
15. U. Maurer, R. Renner, and C. Holenstein, *Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology*, Theory of Cryptography - TCC 2004 (M. Naor, ed.), Lecture Notes in Computer Science, no. 2951, Springer-Verlag, 2004, pp. 21–39.
16. R. C. Merkle, *Secrecy, authentication, and public key systems, PhD thesis*, UMI Research Press, 1982.

17. NIST, *Federal information processing standard 180-2, secure hash standard*, August 2002.
18. 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.
19. 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/`.
20. 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 previous versions of this paper we have removed one of the four conditions for soundness, reducing them to three. This condition, *parameter-completeness*, applies to parameterized hashing modes and is necessary for guaranteeing that a message M, hashed with two different parameter values A and A' will not lead systematically to the same digest. More specifically, we considered two different inputs (M, A) and (M, A') leading to the same digest a collision. For some message lengths, the tree template may be independent of a parameter value and hence the digest is too. Avoiding a collision for such cases requires explicitly coding 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 paper we have adapted this definition, removing the need for parameter-completeness.

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 grey rectangles and their position in the message indicated, and chaining blocks by dark grey 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 2. 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

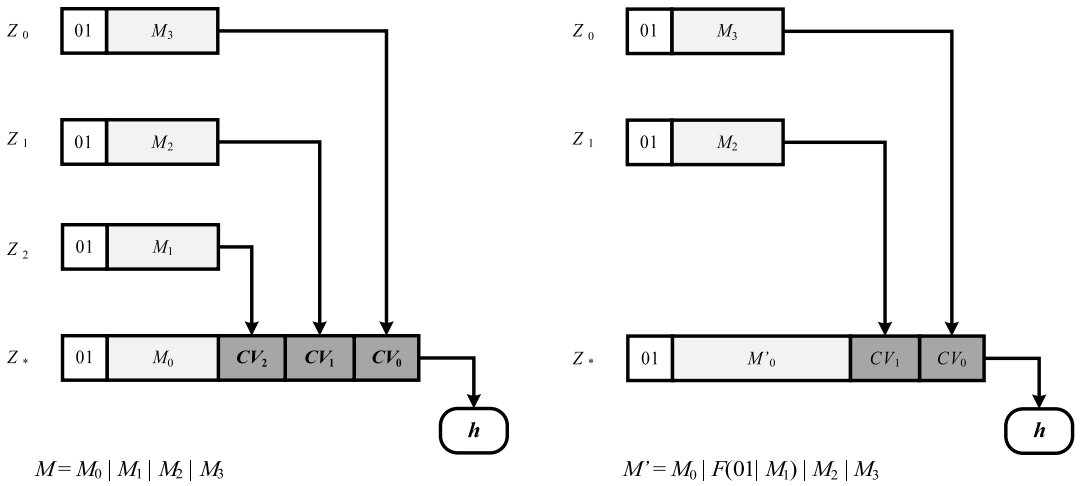

Fig. 2. Example of an inner collision without a collision in $\mathcal{F}$

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 3. Given the output of $h = \mathcal{T}[\mathcal{F}](M)$ of some message M, length-extension is the possibility to compute the output of $\mathcal{T}[\mathcal{F}](M')$ with M a substring of M', only knowing h and not M itself. Figure 3 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.

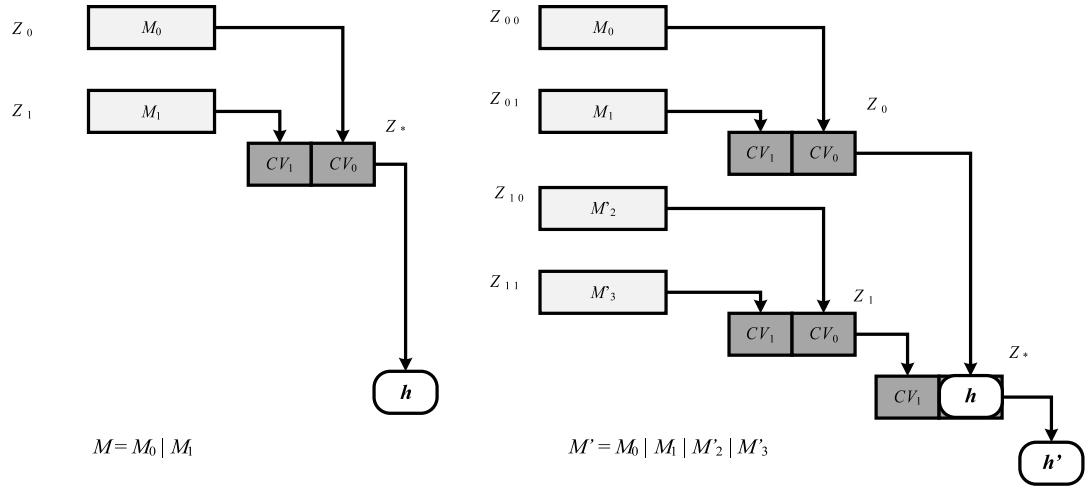

Fig. 3. Example of the generalization of length extension to tree hashing

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 distinguisher 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 $2^{n/2}$.