Sufficient conditions for sound hashing using a truncated permutation

Joan Daemen[1], Tony Dusenge[2], and Gilles Van Assche[1]

[1] STMicroelectronics
[2] Université Libre de Bruxelles

Abstract. In this paper we give a generic security proof for hashing modes that make use of an underlying fixed-length permutation. We formulate a set of five simple conditions, which are easy to implement and to verify, for such a hashing mode to be *sound*. These hashing modes include tree hashing modes and sequential hashing modes. We provide a proof that for any hashing mode satisfying the five 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 permutation and n the length of the chaining values.

Keywords: permutation-based hashing, indifferentiability, tree hashing

1 Introduction

In this paper, we give a generic security proof for tree and sequential hashing modes calling a fixed-length permutation. We formulate a number of simple conditions for such a hashing mode to be *sound*. For the soundness, we base ourselves on the indifferentiability framework introduced by Maurer et al. in [10] and applied to hash functions by Coron et al. in [7]. In particular, we prove an upper bound of the advantage of an adversary in differentiating a hashing mode calling a random permutation from an ideal hash function based on a random oracle as a function of the attack cost and the length of the chaining value. As already stated in [10] and formalized in [1, Theorem 2], the success probability of any generic attack (e.g., collision, pre-image, length-extension, ...) on such a hashing mode has success probability at most the proven bound plus the success probability of this attack on a random oracle. It suffices to take the chaining value long enough to make the difference between the success probability for the hashing mode and a random oracle negligible.

After the indifferentiability paradigm was applied to hash functions in [7], indifferentiability was proven for several other modes such as enveloped Merkle-Damgård (EMD) transform in [2] and chopped Merkle-Damgård in [6]. These modes are sequential and call an ideal compression function or an ideal block cipher that is used in Davies-Meyer mode. In contrast, the modes treated in this paper call a random permutation and do not require a feedforward. Note that the first hashing mode calling a random permutation that was proven indifferentiable was the sponge construction [4]. However, as opposed to the modes treated in this paper, the sponge mode is strictly sequential.

Provable security of tree hashing was already investigated in [12] and indifferentiability of permutation-based tree hashing modes was treated in [8], covering the mode used in the SHA-3 candidate MD6 [11]. However, as opposed to the modes treated in this paper, they used a two-layer approach. First, the tree hashing mode was proven indifferentiable assuming an underlying ideal compression function. Second, an ideal compression function construction was proven indifferentiable assuming an underlying random permutation. As another example, the paper [5] proves the indifferentiability of tree hashing modes calling an ideal compression function, which can in turn be instantiated by using either a permutation-based or block cipher-based construction.

In contrast, this paper addresses modes composed of only one layer. This allows a more efficient construction. For instance, the compression function in [8] is built by fixing part of the input of the permutation and truncating its output. Our construction is more efficient in that in typical

scenario's for most calls to the permutation its full input can be used. This is especially relevant in sequential hashing where the full input can be used for all but one permutation call.

Despite similarity with some of the mentioned prior work, we believe this paper has a substantial added value as it is to our knowledge the first time that indifferentiability is proven for tree hashing modes (with sequential hashing as a special case) calling a random permutation. Moreover, the bound we achieve on the success probability of differentiating the mode from an ideal hash function is as tight as theoretically possible. Finally, we treat a more general case than prior work in several aspects. Our mode of use is *parameterized*, with parameters specifying the way to build the tree.

The remainder of this paper is organized as follows. After providing a rigorous definition of tree hashing modes in Section 2, we introduce in Section 3 a set of simple and easy-to-verify conditions for permutation-based tree hashing modes that result in sound tree hashing. After adapting the indifferentiability setting of [7] to permutation-based tree hashing in Section 4, we provide in Section 5 the indifferentiability proof as a series of lemmas and a final theorem. In Appendix A we illustrate our formalism to describe tree hashing.

2 Tree hashing mode

In this paper, we consider tree hashing modes such as those treated in [5] by Bertoni et al. and cover sequential hashing modes as a special case. In [5], tree hashing modes $\mathcal{T}$ were considered that call an underlying function $\mathcal{F}$, called *inner function*, with variable input length and indefinite output length. In this paper we consider the case where the inner function is a permutation $\mathcal{P}$ operating on b-bit values. The tree hashing modes $\mathcal{T}$ applied to a permutation $\mathcal{P}$ defines a concrete hash function $\mathcal{T}[\mathcal{P}]$, called *outer hash function*.

In our modes, we use a permutation $\mathcal{P}$ with the output truncated to its first n bits, denoted by $\mathcal{P}_n$, to compute chaining values. Therefore, our outer hash function $\mathcal{T}[\mathcal{P}]$ produces outputs with fixed length n. In the rest of this section, we give the tree hashing mode description taken from [5], keeping in mind that $\mathcal{F}$ is $\mathcal{P}_n$ in our case.

We consider the general case of parameterized hash functions. Next to the input message M, such a function takes as input a set of parameter values A that determine the topology of the hashing tree such a node degree or total depth. In the simplest case, this set may be empty and the tree topology may be fully determined by the message length $|M|$.

2.1 Hashing as a two-step process

A tree hashing mode specifies for any given parameter choice A 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 bit string constructed according to the corresponding node template and presented to $\mathcal{F}$. We write $S = \mathcal{Z}[\mathcal{F}](M, Z)$. 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 $h = \mathcal{F}(S_*)$. This two-step process is illustrated in Appendix A.

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

2.2 The tree structure

The node templates of a tree template Z are denoted by Z_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.

Related to the tree topology, we now introduce some terminology and concepts. 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 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 the *final node*. We use the index $*$ to denote the final node. 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 child nodes. We call the number of child nodes of a node its *degree* and a node without child nodes 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 for all the nodes it contains, except one, the parents are also in the restriction. We call a subtree a *final subtree* if it contains the final node. 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* $Z_\alpha[x]$, $0 \le x < |Z_\alpha|$, and instructs the forming of a bit string called the node instance S_α in the following way. Each template bit has a type and the following attributes (and purpose), depending on its type:

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

Message pointer bits Represent bits 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 taken from the output of a previous call to $\mathcal{F}$. A chaining pointer bit has 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 denote the chaining value obtained by applying the inner hash function to S_α by c_α.

Appendix A gives an illustration of this concept.

3 Sufficient conditions for sound tree hashing

In this paper, we assume that the four conditions for a sound tree hashing mode $\mathcal{T}$, formulated by Bertoni et al. [5] are fulfilled. In addition, we introduce a fifth condition specific for a tree hashing mode using a truncated permutation. First, let us briefly recall the four conditions of [5].

Consider a tree hashing mode $\mathcal{T}$ using an inner function $\mathcal{F}$ (e.g., a truncated permutation as in our case), and using the truncation of $\mathcal{F}$ to its first n output bits, denoted by $\mathcal{F}_n$ to compute chaining values.

Definition 1 ([5]). *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 $S_* = S'_*$.*

A collision of $\mathcal{F}_n$ can be used to generate an inner collision. However, an inner collision does not necessarily imply an output collision of $\mathcal{F}_n$. One can define tree hashing modes where it is possible to produce an inner collision without collision in $\mathcal{F}_n$ (see Appendix A for an illustrated example). To avoid this situation, the concept of *tree decodability* has been introduced.

Definition 2 ([5]). *A mode of use $\mathcal{T}$ is* tree-decodable *if for all tree instances S generated with $\mathcal{T}$ (i.e., there is 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.*

In [5], it has been proven that when $\mathcal{T}$ is tree-decodable, an inner collision in $\mathcal{T}[\mathcal{F}]$ implies an output collision in $\mathcal{F}_n$. This leads to the first condition.

Condition 1 ([5]) *$\mathcal{T}$ 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, the concept of *message-completeness* has been introduced.

Definition 3 ([5]). *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.*

Condition 2 ([5]) *$\mathcal{T}$ is message-complete.*

Similarly, we can have an output collision in $\mathcal{T}[\mathcal{F}]$ without an inner collision if the tree instance does not allow to fully determine the tree parameters A.

This leads to the condition for $\mathcal{T}$ that for all inputs (M, A), the resulting tree instance S fully determines A. This property has been called *parameter-completeness*.

Condition 3 ([5]) *$\mathcal{T}$ is parameter-complete.*

The fourth condition prevents a property that generalizes length extension to tree hashing. That is, given an output $h = \mathcal{T}[\mathcal{F}](M)$ of some message M, one can compute the output $h' = \mathcal{T}[\mathcal{F}](M')$ with M a substring of M', without knowing M (see Appendix A for an illustrated example). The simplest way to avoid this is to have domain separation between final and inner nodes.

Condition 4 ([5]) *$\mathcal{T}$ 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.*

For a tree hashing mode $\mathcal{T}$ using $\mathcal{P}_n$ to compute chaining values, generating an inner collision is easy due to the possibility of computing $\mathcal{P}^{-1}$, the inverse of the permutation. For instance, consider a leaf node instance S_α of a given known tree instance S. The evaluation of $\mathcal{P}_n(S_\alpha)$ gives a n-bit chaining value c which is in the parent node instance of S_α in S. The evaluation of $\mathcal{P}^{-1}(c||x)$ (with $||$ denoting concatenation) returns a b-bit value X. If X has the right coding for a leaf node (e.g., frame bit values indicating that they are leaf nodes), we can replace S_α by X in S and this gives an inner collision: another tree instance S' which differs from S only by one leaf node instance.

A solution to this problem, called *leaf node anchoring*, is to impose a fixed initial value IV in a fixed position in each leaf node. For the generation of an inner collision as described above to succeed, the evaluation of $\mathcal{P}^{-1}(p)$ shall returns an inner node instance X having the IV in the right position. It turns out that for the simplicity of the security proof, non-leaf nodes shall have a chaining value at that position. Without loss of generality, we take for the fixed position just the first n bits of the node. We use the notation $\lfloor x \rfloor_n$ to denote the truncation of a binary string x to its n first bits.

Definition 4. *A mode of use $\mathcal{T}$ is* leaf-anchored *if for any leaf node template Z_α generated with $\mathcal{T}$, $\lfloor Z_\alpha \rfloor_n$ contains frame bits coding a fixed value IV and for any non-leaf node Z_β generated with $\mathcal{T}$, $\lfloor Z_\beta \rfloor_n$ contains a chaining value.*

Condition 5 *$\mathcal{T}$ is leaf-node-anchored.*

4 The distinguisher's setting

We study the indifferentiability of a tree hashing mode $\mathcal{T}$, using a random permutation $\mathcal{P}$, from an ideal hash function $\mathcal{G}$. We base ourselves on the indifferentiability framework introduced by Maurer et al. in [10] and applied to hash functions by Coron et al. in [3].

The adversary shall distinguish between two systems (see Figure 1) using their responses to sequences of queries. The system at the left is $\mathcal{T}[\mathcal{P}]$ and $\mathcal{P}$, and the adversary can make queries to both subsystems separately, where the former in turn calls the latter to construct its responses. As $\mathcal{P}$ is a permutation, the distinguisher can also make calls to its inverse $\mathcal{P}^{-1}$. She has the following interfaces to this system:

- $\mathcal{H}$ taking as input (M, A) with $M \in \mathbb{Z}_2^*$ and A the value of the mode parameters, and returning a binary string $y \in \mathbb{Z}_2^n$, i.e., $y = \mathcal{T}[\mathcal{P}](M, A)$;
- $\mathcal{I}^{\pm 1}$ combining two sub-interfaces:
  - $\mathcal{I}^{+1}$ taking as input a binary string $s \in \mathbb{Z}_2^b$, and returning a binary string $p \in \mathbb{Z}_2^b$ with $p = \mathcal{P}(s)$;
  - $\mathcal{I}^{-1}$ taking as input a binary string $p \in \mathbb{Z}_2^b$, and returning a binary string $s \in \mathbb{Z}_2^b$ with $s = \mathcal{P}^{-1}(p)$.

The system at the right consists of an ideal hash function $\mathcal{G}$ and of a simulator $\mathcal{S}$ simulating the permutation. It offers the same interface as the left system. $\mathcal{G}$ provides the interface $\mathcal{H}$ and returns an output truncated to n bits when queried with (M, A). The permutation simulator provides the interface $\mathcal{I}^{\pm 1}$ combining two sub-interfaces $\mathcal{I}^{+1}$ and $\mathcal{I}^{-1}$, like in the left system.

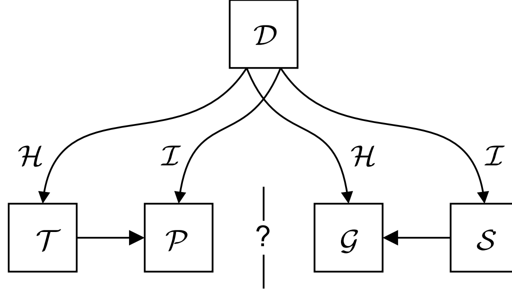

Fig. 1. The differentiability setup

First, the simulator should be *self-consistent*: if queried with the same query multiple times, it should give the same response. Second, the output of $\mathcal{S}$ should look *consistent* with what the distinguisher can obtain from the ideal hash function $\mathcal{G}$, like if $\mathcal{S}$ was $\mathcal{P}$ and $\mathcal{G}$ was $\mathcal{T}[\mathcal{P}]$. To achieve that, the simulator can query $\mathcal{G}$, denoted by $\mathcal{S}[\mathcal{G}]$. Note that the simulator does not see the distinguisher's queries to $\mathcal{G}$. Third, $\mathcal{S}$ must simulate a permutation consistently: $\mathcal{I}^{+1}(s) \neq \mathcal{I}^{+1}(s')$ iff $s \neq s'$, $\mathcal{I}^{-1}(p) \neq \mathcal{I}^{-1}(p')$ iff $p \neq p'$ and $p = \mathcal{I}^{+1}(s)$ iff $s = \mathcal{I}^{-1}(p)$. We call this property *permutation-consistency.*

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

In this setting, the distinguisher can send queries Q to both interfaces. Let $\mathcal{X}$ be either $(\mathcal{T}[\mathcal{P}], \mathcal{P})$ or $(\mathcal{G}, \mathcal{S}[\mathcal{G}])$. The sequence of queries Q to $\mathcal{X}$ consists of a sequence of queries to the interface $\mathcal{H}$, denoted $Q_{\mathcal{H}}$ and a sequence of queries to the interface $\mathcal{I}^{\pm 1}$, denoted $\mathcal{Q}_{\mathcal{I}^{\pm 1}}$. $Q_{\mathcal{H}}$ is a sequence of couples $Q_{\mathcal{H},i} = (M_i, A_i)$, and $\mathcal{Q}_{\mathcal{I}^{\pm 1}}$ is a sequence of couples $\mathcal{Q}_{\mathcal{I}^{\pm 1},i} = (k, f)$ with $k \in \mathbb{Z}_2^b$ and f a flag equal to 1 or -1, indicating whether the query k is sent to $\mathcal{I}^{+1}$ or $\mathcal{I}^{-1}$.

In the following, we use the concept of $\mathcal{T}$-*consistency* recalled below. 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.

Definition 5 ([5]). *For a given set of queries Q and their responses $\mathcal{X}(Q)$, the $\mathcal{T}$*-consistency *is 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 $\mathcal{I}^{\pm 1}$ interface (when the queries $\mathcal{Q}_{\mathcal{I}^{\pm 1}}$ suffice to perform this calculation), i.e., that $\mathcal{X}(Q_{\mathcal{H}}) = \mathcal{T}[\mathcal{X}(\mathcal{Q}_{\mathcal{I}^{\pm 1}})](Q_{\mathcal{H}})$.*

4.1 The cost of queries

We consider the same cost of queries setting as in [5]. The *cost* q of queries to a system $\mathcal{X}$ is the total number of calls to $\mathcal{P}$ or $\mathcal{P}^{-1}$ it would yield if $\mathcal{X} = (\mathcal{T}[\mathcal{P}], \mathcal{P})$, either directly due to queries $\mathcal{Q}_{\mathcal{I}^{\pm 1}}$, or indirectly via queries $Q_{\mathcal{H}}$ to $\mathcal{T}[\mathcal{P}]$. The cost of a sequence of queries is fully determined by their number and their input. Each query $\mathcal{Q}_{\mathcal{I}^{\pm 1},i}$ to $\mathcal{P}$ or $\mathcal{P}^{-1}$ contributes 1 to the cost. Each query $Q_{\mathcal{H},i} = (M_i, A_i)$ 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 $\mathcal{T}[\mathcal{P}]$ needs to make to $\mathcal{P}$ 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 $\mathcal{T}(|M|, A)$.

Duplicate queries are not taken into account. This means that two equal queries $\mathcal{Q}_{\mathcal{I}^{\pm 1},i}$ or two equal queries $Q_{\mathcal{H},i}$ are counted as one. 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.

4.2 Definition

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

Algorithm 1 The simulator $\mathcal{S}[\mathcal{G}]$

1: **Initialization:** $T = T^+ \cup T^- \leftarrow \emptyset$, $\mathcal{C} \leftarrow \{\text{IV}\}$

2: **Interface** $p = \mathcal{I}^{+1}(s)$ with $s, p \in \mathbb{Z}_2^b$
3: **if** $\exists(s,t) \in T$ **then**
4: **return** $p = t$
5: **end if**
6: **if** s is a final node instance **then**
7: Retrieve input from s using T^+ according to Algorithm 2
8: **if** Input retrieval returned (M, A) **then**
9: Set p to $\mathcal{G}(M, A)$
10: Append $(b - n)$ uniformly and independently drawn random bits to p
11: **else if** decoding returned a "dead-end at c" exception **then**
12: Choose p randomly and uniformly from $\mathbb{Z}_2^b \setminus T_r$
13: $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
14: **else** {decoding returned a "n-collision" or "invalid coding" exception}
15: Choose p randomly and uniformly from $\mathbb{Z}_2^b \setminus T_r$
16: **end if**
17: **else** {s is an inner node instance}
18: Choose p randomly and uniformly from $\mathbb{Z}_2^b \setminus T_r$
19: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\lfloor p \rfloor_n\}$
20: **end if**
21: Add the couple (s, p) to T^+
22: **return** p

23: **Interface** $s = \mathcal{I}^{-1}(p)$ with $s, p \in \mathbb{Z}_2^b$
24: **if** $\exists(i, p) \in T$ **then**
25: **return** $s = i$
26: **end if**
27: Choose s randomly and uniformly from $\mathbb{Z}_2^b \setminus T_l$
28: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\lfloor s \rfloor_n\}$
29: Add couple (s, p) to T^-
30: **return** s

Definition 6 ([7]). *A tree hashing mode $\mathcal{T}$ with oracle access to an ideal hash function $\mathcal{P}$ is said to be (t_D, t_S, q, ϵ)-indifferentiable from an ideal hash function $\mathcal{G}$ if there exists a simulator $\mathcal{S}[\mathcal{G}]$, such that for any distinguisher $\mathcal{D}$ it holds that:*

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

The simulator has oracle access to $\mathcal{G}$ and runs in time at most t_S. The distinguisher runs in time at most t_D and has a cost of at most q. Similarly, $\mathcal{T}[\mathcal{P}]$ is said to be indifferentiable from $\mathcal{G}$ if ϵ is a negligible function of the security parameter n.

5 Indifferentiability proof

In this section, we always assume that the five conditions presented in Section 3 are fulfilled by the tree hashing mode $\mathcal{T}$. We first describe the *input retrieval process* that we use to make the simulator satisfy T-consistency. Then we describe the simulator and its general goal. Finally, we prove the indifferentiability results by means of a series of lemmas and a final theorem.

5.1 The simulator

Algorithm 1 specifies the simulator $\mathcal{S}[\mathcal{G}]$. It has the following design principles. It is always self-consistent. It violates permutation-consistency only with a very small probability and satisfies $\mathcal{T}$-consistency as long as a particular event, called a *$\mathcal{C}$-collision*, does not occur. As long as permutation-consistency is not violated and there are no $\mathcal{C}$-collisions, its output has the same distribution as that of a random permutation.

To satisfy self-consistency, it keeps track of the queries and their responses in a table T containing couples (s, p) with $s, p \in \mathbb{Z}_2^b$. We denote the set of first members of these couples by T_l and the set of second members by T_r. Initially, this table is empty. When receiving a query $\mathcal{I}^{+1}(s)$ with s already in T_l, the simulator returns the second member of $(s, p) \in T$ (line 4). Similarly, when receiving a query $\mathcal{I}^{-1}(p)$ with p already in T_r, the simulator returns the first member of $(s, p) \in T$ (line 25).

In general the simulator satisfies permutation-consistency in the following way. When receiving a query $\mathcal{I}^{+1}(s)$ with $s \notin T_l$ it selects the response p randomly from the set of possible values, excluding those in T_r (lines 12, 15 and 18). When receiving a query $\mathcal{I}^{-1}(p)$ with $p \notin T_r$ it selects the response s randomly from the set of possible values, excluding those in T_l (line 27). This conflicts with $\mathcal{T}$-consistency in certain queries $\mathcal{I}^{+1}(s)$ with s a final node. In that case, and in that case only, permutation-consistency may be violated (lines 9-10). Note that if the simulator has violated permutation-consistency, there may be multiple pairs in T with the same second member and the simulator's response to $\mathcal{I}^{-1}$ (line 25) is not well-defined. This could be fixed but would complicate the description of the simulator and in our proof we consider the adversary has succeeded as soon as permutation-consistency is violated.

The simulator has a multiset $\mathcal{C}$ containing n-bit chaining values and the IV (Definition 4). A multiset is a generalization of a set, in that a member may occur more than once. The number of times a member occurs in a multiset is its *multiplicity*. Initially, $\mathcal{C}$ contains the IV and the simulator adds members to $\mathcal{C}$ when receiving queries:

- A query $\mathcal{I}^{+1}(s)$, with s an inner node, adds to $\mathcal{C}$ the chaining value $\lfloor p \rfloor_n$ (line 19).
- A query $\mathcal{I}^{-1}(p)$ adds $\lfloor s \rfloor_n$ (line 28) to $\mathcal{C}$. Due to leaf-node anchoring, if the response s is a leaf node, $\lfloor s \rfloor_n = \text{IV}$ and otherwise $\lfloor s \rfloor_n$ is a chaining value.
- A query $\mathcal{I}^{+1}(s)$, with s a final node, adds the chaining value c to $\mathcal{C}$ iff the input retrieval of s returns a "dead-end at c" exception.

The multiset $\mathcal{C}$ has no influence on the way the simulator generates its responses and its purpose is to define a concept that facilitates our proof: $\mathcal{C}$-collisions.

Definition 7. *There is a* $\mathcal{C}$-collision *in the simulator if its multiset $\mathcal{C}$ has at least one member with multiplicity larger than 1.*

The simulator satisfies $\mathcal{T}$-consistency by querying $\mathcal{G}$ if necessary. When making a query $\mathcal{I}^{+1}(s)$ with s a final node (line 6), the simulator performs input retrieval to s (line 7). If the input retrieval of s returns (M, A), the simulator calls $\mathcal{G}$ with (M, A) to guarantee $\mathcal{T}$-consistency (line 9); then it extends the received n-bit value $\mathcal{G}(M, A)$ with $(b - n)$ random bits to make a b-bit response p (line 10). Note that this may may violate permutation-consistency if the generated value p is already in T_l. If the input retrieval of s does not return an input (M, A), the simulator chooses p randomly from all possible values excluding T_r (lines 12 and 15).

5.2 Input retrieval

For $\mathcal{T}$-consistency, we use an input retrieval process very similar to the one called "$\mathcal{T}$-decoding" in [5]. This process extracts an input (M, A) from a given final node instance, using couples (s, p) in T. Input retrieval of a final node s using a table T does not necessarily lead to an input (M, A). In this context we provide the following definition.

Definition 8. *A final node instance s is* T-bound *for a given table T if there exists a tree instance S with $s = S_*$, such that given any proper final subtree S_J of S, for each expanding index β of S_J and the corresponding chaining value c_β, $\exists (S_\beta, p) \in T$ where $\lfloor p \rfloor_n = c_\beta$. Given the message M and the parameter A corresponding to S, we say that s is T-bound to (M, A) via S.*

Note that a final node instance may be T-bound to multiple inputs (M, A).

With respect to input retrieval we distinguish between couples (s, p) in T that are obtained from queries to $\mathcal{I}^{+1}$ and those to $\mathcal{I}^{-1}$, and we denote the former by T^+ and the latter by T^-.

Algorithm 2 Input retrieval

1: **input:** s and T^+
2: **output:** message M and tree parameters A, or an exception
3: Initialization: $J \leftarrow \{*\}$ and set S_* to s
4: **while** S_J has an expanding index β **do**
5: Let $c = c_\beta$, the chaining value corresponding to β extracted from S_J
6: **if** there is one entry $(s', p) \in T^+$ with $\lfloor p \rfloor_n = c$ **then**
7: Let $J \leftarrow J \cup \{\beta\}$ and set S_β to s'
8: **else if** there is no entry $(s', p) \in T^+$ with $\lfloor p \rfloor_n = c$ **then**
9: **return** "dead-end at c" exception
10: **else** {there is more than one entry $(s', p) \in T^+$ with $\lfloor p \rfloor_n = c$}
11: **return** "n-collision" exception
12: **end if**
13: **end while**
14: **if** S_J does not have valid coding to be a tree instance generated with $\mathcal{T}$ **then**
15: **return** "invalid coding" exception
16: **else**
17: Extract the message M and parameters A from the tree instance S_J
18: **return** (M, A)
19: **end if**

Thus, T^+ and T^- form a partition of T. Our input retrieval procedure is specified in Algorithm 2. It makes use of T^+ and ignores couples in T^-.

We define an n-collision in a table T.

Definition 9. *Two couples (s, p) and (s', p') in T with s and s' inner nodes, $s \neq s'$ and $\lfloor p \rfloor_n = \lfloor p' \rfloor_n$ is called an* n-collision *in T.*

If Algorithm 2 is successful for a given node s, it returns an input (M, A) and we say the node s is *input-retrievable*. Otherwise, it returns one of the following *exceptions*:

- "dead-end at c": for some expanding index β, there is no couple (s, p) in T^+ with $\lfloor p \rfloor_n = c_\beta$.
- "n-collision": for some expanding index β, there are more than one couple (s, p) in T^+ with $\lfloor p \rfloor_n = c_\beta$, hence there is an n-collision in T^+.
- "invalid coding": the constructed tree instance S is inconsistent with the tree hashing mode.

When there are no n-collisions in T^+, all T^+-bound final nodes are input-retrievable.

5.3 Events that violate $\mathcal{T}$-consistency

We now introduce events that force the simulator $\mathcal{S}[\mathcal{G}]$ to violate $\mathcal{T}$-consistency: *inner collision in the simulator*, *final node correction* and *fatal inverse*. We will prove (Lemma 6) that there are no other events that can force the simulator to violate $\mathcal{T}$-consistency.

Definition 10. *An* inner collision *in the simulator is the existence of a final node $s \in T_l$ that is T-bound to at least two inputs.*

An inner collision in the simulator violates $\mathcal{T}$*-consistency* with high probability. Let (s, p) be a couple in T with s a final node that is T-bound to both (M, A) and (M', A'). For $\mathcal{T}$-consistency both $\mathcal{G}(M, A)$ and $\mathcal{G}(M', A')$ must be equal to $\lfloor p \rfloor_n$, implying they must be equal. As these are the outputs of a random oracle for two different inputs, the probability that this is not the case is $1 - 2^{-n}$.

Definition 11. *A* final node correction *is a query to the simulator that causes a final node instance s in T_l that was not T-bound to become T-bound.*

A final node correction violates $\mathcal{T}$*-consistency* with high probability. Indeed, assume a couple $(s, p) \in T$, with s a final node instance that is not T-bound. If s becomes T-bound to (M, A), the adversary may query $\mathcal{G}$ with (M, A). The response $\mathcal{G}(M, A)$ will differ from $\lfloor p \rfloor_n$ with probability $1 - 2^{-n}$, since a random oracle chooses it responses randomly.

Definition 12. *A* fatal inverse *is a couple* $(s, p) \in T^-$ *with* s *a* T*-bound final node.*

A fatal inverse violates $\mathcal{T}$*-consistency* with high probability. Given a fatal inverse (s, p), the adversary can reconstruct (M, A) from previous queries and subsequently query $\mathcal{G}(M, A)$. For $\mathcal{T}$-consistency, the result must be equal to the $\lfloor p \rfloor_n$. The probability that it is different is $1 - 2^{-n}$.

5.4 Determining the differentiating advantage

In this section we give the proof by a series of lemmas and a final theorem.

Lemma 1. *As long as there are no* $\mathcal{C}$*-collisions in the simulator, a final node* s *that is* T*-bound to an input* (M, A)*, is also* T^+*-bound to* (M, A)*.*

Proof. Let s be a final node that is T-bound to an input (M, A) via a tree S, but not T^+-bound. This tree has one or more nodes S_γ with $(S_\gamma, p) \in T^-$ for some p. We denote the set of those nodes by S^-. Now let S_α be a node in S^- such that none of its child nodes (if any) is in S^-. As $(S_\alpha, p) \in T^-$, $\lfloor S_\alpha \rfloor_n$ is in $\mathcal{C}$. If S_α is a leaf node, then this implies a $\mathcal{C}$-collision because $\lfloor S_\alpha \rfloor_n = \text{IV}$ and the multiplicity of IV in $\mathcal{C}$ is at least two. Otherwise, S_α has a child node S_β with $(S_\beta, p') \in T^+$ and with $\lfloor p' \rfloor_n = \lfloor S_\alpha \rfloor_n$. The query $\mathcal{I}^{+1}(S_\beta)$ has added $\lfloor p' \rfloor_n = \lfloor S_\alpha \rfloor_n$ to $\mathcal{C}$. It follows that the simulator has a $\mathcal{C}$-collision as the member $\lfloor S_\alpha \rfloor_n$ has multiplicity at least two. □

Lemma 2. *An* n*-collision in* T^+ *implies a* $\mathcal{C}$*-collision in the simulator.*

Proof. Consider two couples (s, p) and (s', p') in T^+ with s and s' inner nodes that form an n-collision, i.e., $\lfloor p \rfloor_n = \lfloor p' \rfloor_n$. As both $\lfloor p \rfloor_n$ and $\lfloor p' \rfloor_n$ are added to $\mathcal{C}$, the simulator has a $\mathcal{C}$-collision. □

Corollary 1. *As long as there are no* $\mathcal{C}$*-collisions in the simulator, any final node* s *that is* T*-bound is input-retrievable.*

Proof. Let s be a final node that is T-bound to an input (M, A). Thanks to Lemma 1, s is T^+-bound to the input (M, A). Hence, there can be no "dead-end at c" exception (line 8 of Algorithm 2). Thanks to Lemma 2 the exception in line 10 of Algorithm 2 cannot occur. It follows that a T-bound final node is input-retrievable. □

Lemma 3. *An inner collision in the simulator implies a* $\mathcal{C}$*-collision in the simulator.*

Proof. Assume s is a final node, T-bound to both inputs (M, A) and (M', A'). Thanks to Lemma 1, s is T^+-bound to input (M, A) via some tree instance S and T^+-bound to input (M', A') via some tree instance S'. The trees S and S' have the same final node but must differ in at least one node. Let α be the index with the smallest height such that $S_\alpha \neq S'_\alpha$ and $S_{\text{parent}(\alpha)} = S'_{\text{parent}(\alpha)}$. Such an index exists as the final nodes of S and S' are equal. It follows that T^+ has an n-collision: it contains two couples (S_α, p) and (S'_α, p') with $\lfloor p \rfloor_n = \lfloor p' \rfloor_n$. According to Lemma 2 this implies a $\mathcal{C}$-collision. □

Lemma 4. *A final node correction implies a* $\mathcal{C}$*-collision in the simulator.*

Proof. Let s be a final node that is not T-bound and $(s, p) \in T$. Now suppose there is no $\mathcal{C}$-collision in the simulator and there is a final node correction making s T-bound. According to Lemma 1 s becomes T^+-bound to some input (M, A) via some tree instance S. This implies that all chaining values in the tree S are in $\mathcal{C}$. When the final node s was queried, it could not have returned an input (M, A). It could not have returned an "n-collision" exception either as according to Lemma 2 an n-collision in T^+ implies a $\mathcal{C}$-collision. So there are two remaining cases:

- Input retrieval returned a "dead-end at c" exception: this implies $\mathcal{C}$ contains c due to that query. As c is a chaining value in S, it would occur in $\mathcal{C}$ twice forming a $\mathcal{C}$-collision.

– Input retrieval returned an “invalid coding” exception: this implies that during input retrieval of s a tree S' was obtained that differs from the tree S to which s is T-bound after the final node correction. As there are no $\mathcal{C}$-collisions in the simulator, due to Corollary 1 s is now input-retrievable. However, if Algorithm 2 returns for a given final node s an “invalid coding” exception, adding couples to T^+ cannot change that if there are no $\mathcal{C}$-collisions: Algorithm 2 builds the tree S_J by iteratively processing expanding indices until it can no longer find one. As long as couples (s', p') are added to T^+ with $\lfloor p' \rfloor_n \neq c_\beta$ for all expanding indices $\beta \in J$, this process is not affected and an “invalid coding” exception will continue to be returned. If a couple (s', p') is added to T^+ with $\lfloor p' \rfloor_n = c_\beta$ for some expanding indices $\beta \in J$, this implies an n-collision and hence a C-collision.

□

Lemma 5. *A fatal inverse implies a $\mathcal{C}$-collision in the simulator.*

Proof. We assume that there is a fatal inverse, i.e., the simulator responds s to a query $\mathcal{I}^{-1}(p)$, with s a final node T-bound to an input (M, A) via a tree S. Due to Lemma 1, all chaining values in S are in $\mathcal{C}$. It follows that $\mathcal{C}$ also contains the chaining value in s that consists of its first n bits: $\lfloor s \rfloor_n$. The query $\mathcal{I}^{-1}(p)$ that returns s adds $\lfloor s \rfloor_n$ again to $\mathcal{C}$, resulting in a $\mathcal{C}$-collision. □

Lemma 6. *Violation of $\mathcal{T}$-consistency implies a $\mathcal{C}$-collision in the simulator.*

Proof. Assume the simulator has no $\mathcal{C}$-collisions after a number of queries. We know from Corollary 1 that any T-bound final node is input-retrievable. Now assume that a new query to the simulator violates $\mathcal{T}$-consistency. That is, after the simulator responded to the new query, there is a couple $(s, p) \in T$ with s a final node T-bound to (M, A), such that $\lfloor p \rfloor_n \neq \mathcal{G}(M, A)$. There are three possible cases for the state of the simulator prior to this query.

1. s was not T-bound: violation of $\mathcal{T}$-consistency is per definition due to a final node correction of $s = S_*$ due to the new query. According to Lemma 4, a final node correction implies a $\mathcal{C}$-collision.
2. s was T-bound to (M, A): if (s, p) was obtained from a query $\mathcal{I}^{+1}(s)$, the simulator would have guaranteed $\mathcal{T}$-consistency by querying $\mathcal{G}$ with (M, A) (see line 9 of Algorithm 1) and there cannot be a violation of $\mathcal{T}$-consistency. Thus, (s, p) was obtained from a query $\mathcal{I}^{-1}(p)$ and s is a fatal inverse of p. According to Lemma 5, a fatal inverse implies a $\mathcal{C}$-collision.
3. s was T-bound to an input $(M', A') \neq (M, A)$: violation of $\mathcal{T}$-consistency can only be due to an inner collision in the simulator due to the new query, causing s to become also T-bound to (M, A). According to Lemma 3, an inner collision in the simulator implies a $\mathcal{C}$-collision.

□

Lemma 7. *If there are no $\mathcal{C}$-collisions in the simulator, 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 8. *If there are no $\mathcal{C}$-collisions in the simulator and if it does not violate permutation-consistency, the output of the simulator has the same distribution to that of a random permutation.*

Proof. We distinguish between three types of query:

1. Query $\mathcal{I}^{+1}(s)$, with s an inner node instance or a final node instance that is not input-retrievable. The simulator generates the image p randomly with the constraint that it guarantees permutation-consistency by excluding elements in T_r.

2. Query $\mathcal{I}^{-1}(p)$. The simulator generates the pre-image s randomly with the constraint that it guarantees permutation-consistency by excluding elements in T_l.
3. Query $\mathcal{I}^{+1}(s)$ with s an input-retrievable final node instance. In this case, the output is generated randomly.

So for the first two types of queries, the responses of the simulator to queries are random with only constraint that they are permutation-consistent. For the last type, imposing that it does not violate permutation-consistency is equivalent to stating that it has the same distribution as a random permutation. □

Lemma 9. *The probability P_s that the simulator has a $\mathcal{C}$-collision or violates permutation-consistency in q queries is upper-bounded by $1 - e^{-\frac{q(q+1)}{2^{n+1}}}$.*

Proof. Assuming the simulator has received $i-1$ queries and there have been no $\mathcal{C}$-collisions and it is permutation-consistent we evaluate the probability that an additional query results in a $\mathcal{C}$-collision or the violation of permutation-consistency. Depending on the type of the new query, three cases must be considered.

1. Query $\mathcal{I}^{+1}(s)$ with s an input-retrievable final node instance. In this case, a $\mathcal{C}$-collision is impossible as no element is added to $\mathcal{C}$. Violation of permutation-consistency is possible though. According to Algorithm 1, all bits of the response p are chosen randomly and uniformly (see lines 9 and 10). After $i-1$ queries, there are at most $i-1$ values in T_r. So the probability that the simulator returns a response $p \in T_r$ is upper-bounded by $P_s \leq \frac{i-1}{2^b}$.
2. Query $\mathcal{I}^{+1}(s)$, with s an inner node instance or a final node instance that is not input-retrievable. Here, the simulator guarantees permutation-consistency by excluding the elements of T_r. Hence, it can choose from $2^b-(i-1)$ possible responses. As each query to the simulator adds at most one element to $\mathcal{C}$ and it has initially one element, it contains at most i elements. There are now at most $i \times 2^{b-n}$ possible responses with their first n bits in $\mathcal{C}$. Thus, the probability that the query results in a $\mathcal{C}$-collision is at most $\frac{i\times 2^{b-n}}{2^b-(i-1)}$.
3. Query $\mathcal{I}^{-1}(p)$. The simulator guarantees permutation-consistency by excluding the elements of T_l. Hence, it can choose from $2^b-(i-1)$ possible responses. There are now at most $i \times 2^{b-n}$ possible responses with their first n bits in $\mathcal{C}$. Thus, the probability that the query results in a $\mathcal{C}$-collision is at most $\frac{i\times 2^{b-n}}{2^b-(i-1)}$.

It follows that the probability that the simulator has a $\mathcal{C}$-collision or violates permutation-consistency in the i-th query if it had no $\mathcal{C}$-collision and was permutation-consistent before is upper-bounded by $\max(\frac{i}{2^b}, \frac{i\times 2^{b-n}}{2^b-(i-1)}) = \frac{i\times 2^{b-n}}{2^b-(i-1)}$. If $i \lll 2^b$ this can be closely approximated by $\frac{i}{2^n}$.

The probability P_s that a $\mathcal{C}$-collision or permutation-consistency violation occurs in the simulator in q queries is hence given by (using the approximation $\log(1-\epsilon) \approx -\epsilon$):

$$P_s = 1 - \prod_{i=1}^{q}\left(1 - \frac{i}{2^n}\right) \approx 1 - e^{-\sum_{i=1}^{q}\frac{i}{2^n}} = 1 - e^{-\frac{q(q+1)}{2^{n+1}}}$$

□

We have now all ingredients to prove our main theorem.

Theorem 1. *A tree hashing mode $\mathcal{T}[\mathcal{P}]$ that uses $\mathcal{P}_n$ for the chaining values and satisfies Conditions 1, 2, 3, 4 and 5 is (t_D, t_S, q, ϵ)-indifferentiable from an ideal hash function, for any for any t_D, $t_S = O(q^3)$, $q < 2^n$ and any $\epsilon > 1 - e^{-\frac{q(q+1)}{2^{n+1}}}$.*

Proof. Lemma 9 upper bounds the probability P_s of $\mathcal{C}$-collisions or violation of permutation-consistency happening during q queries to the simulator. Lemma 7 states that 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 $Q_{\mathcal{H}}$. The combination of these lemmas upper bound the probability P_s for any set of queries $Q_{\mathcal{H}}$ and $Q_{\mathcal{I}}$ as a function of their total cost q.

Lemma 8 states that if there are no $\mathcal{C}$-collisions in the simulator and it does not violate permutation-consistency, the output of the simulator has the same distribution as a random permutation. According to [9, Theorem 1], the adversary's advantage can therefore by bounded by $P_s = 1 - e^{\frac{q(q+1)}{2^{n+1}}}$.

We have $t_S = O(q^3)$ as the simulator may have to perform input-retrieval for at most q node instances, each requiring to look up at most q nodes in the table T^+ with at most q entries. □

If $1 \lll q \lll 2^n$ we have $P_s \approx q^2/2^{n+1}$.

6 Conclusions

We have proven that a hashing mode that calls a compression function consisting of a truncated fixed-input-length permutation achieves the best possible differentiating advantage if it satisfies five simple conditions. This is valid for both sequential and tree-hashing modes.

References

1. E. Andreeva, B. Mennink, and B. Preneel, *Security reductions of the second round SHA-3 candidates*, Cryptology ePrint Archive, Report 2010/381, 2010, `http://eprint.iacr.org/`.
2. 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. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, pp. 181–197.
5. ———, *Sufficient conditions for sound tree and sequential hashing modes*, Cryptology ePrint Archive, Report 2009/210, 2009, `http://eprint.iacr.org/`.
6. 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.
7. J. Coron, Y. Dodis, C. Malinaud, and P. Puniya, *Merkle-Damgård revisited: How to construct a hash function*, Advances in Cryptology – Crypto 2005 (V. Shoup, ed.), LNCS, no. 3621, Springer-Verlag, 2005, pp. 430–448.
8. 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.
9. U. Maurer, *Indistinguishability of random systems*, Advances in Cryptology – Eurocrypt 2002 (L. Knudsen, ed.), Lecture Notes in Computer Science, vol. 2332, Springer-Verlag, May 2002, pp. 110–132.
10. 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.
11. 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/`.
12. 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 Illustrations

In this section we illustrate the tree hashing mode described in section 2 and two undesired properties of tree hashing modes explained in Section 3 to introduce two of the four conditions for sound tree hashing.

In our figures of tree templates we use the following conventions. We depict message, chaining and frame blocks rather than individual bits, where a block is just a sequence of consecutive bits. Frame blocks are depicted by white rectangles with their 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 relation between the nodes is indicated by arrows, symbolizing the application of $\mathcal{F}$ (or $\mathcal{P}_n$) during template execution for a concrete input M.

Figure 2 shows a tree template consisting of a number of node templates. Each row represents a call to the inner function $\mathcal{F}$. Each node contains frame bits (with constant bit values). Leaf nodes contain message pointer bits representing bits taken form the message M. Except leaf nodes, other nodes contain chaining pointer bits representing chaining value bits taken from the output of a previous call to $\mathcal{F}$.

Figure 3 represents a tree instance obtained after executing the tree template with the message M and the parameters A. Message pointer bits in leaf nodes have been replaced by the message bits. The output of $\mathcal{F}$ after treating the final node constitutes the hash value.

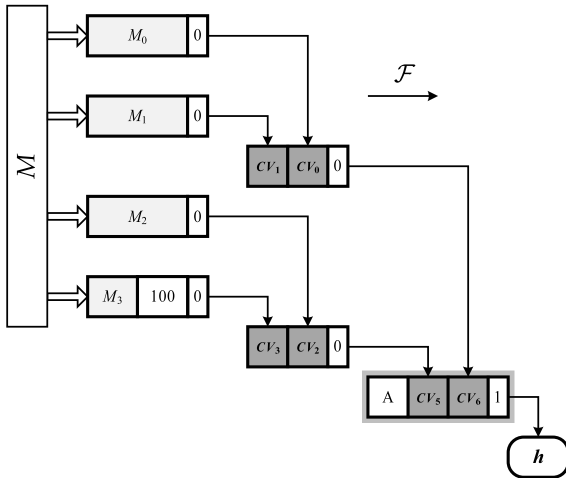

Fig. 2. A tree template.

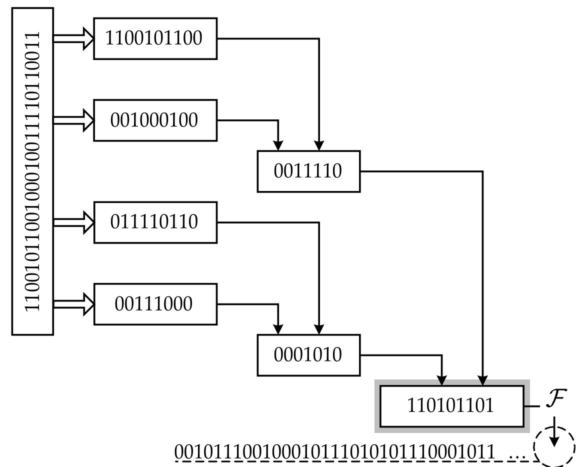

Fig. 3. A tree instance.

We now illustrate undesired properties using 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.

The first property is related to the existence of inner collisions in the absence of collisions in the output of $\mathcal{F}$ and is illustrated in Figure 4. The figure depicts two templates that are generated by a mode of use $\mathcal{T}$ for two different message lengths. All nodes have as first two bits frame bits with value 01. The template on the left has four nodes: three leaf nodes of height 1 and a final node that takes an input block and the chaining values corresponding to the three leaf nodes. The template on the right has three nodes: two leaf nodes of height 1 and a final node that takes an input block and the chaining values corresponding to the two leaf nodes. Note that the final node of the right template has a message block (indicated by M'_0) in the place where the final node of the left template has the concatenation of a message block M_0 and a chaining block CV_2. We can exploit this fact to construct an inner collision from any message M with length matching the left template. As can be seen in the figure, it suffices to form M' by replacing in M the block M_1 by $\mathcal{F}(01|M_1)$.

The second property, a generalization of length-extension to tree hashing, is illustrated in Figure 5. Given the output of $h = \mathcal{T}[\mathcal{F}](M)$ of some message M, length-extension is 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 5 depicts two templates corresponding with two different message lengths. The templates have a binary tree structure. The template at the left has three nodes: two leaf nodes and a final node containing the chaining values corresponding to the two leaf nodes. The template at

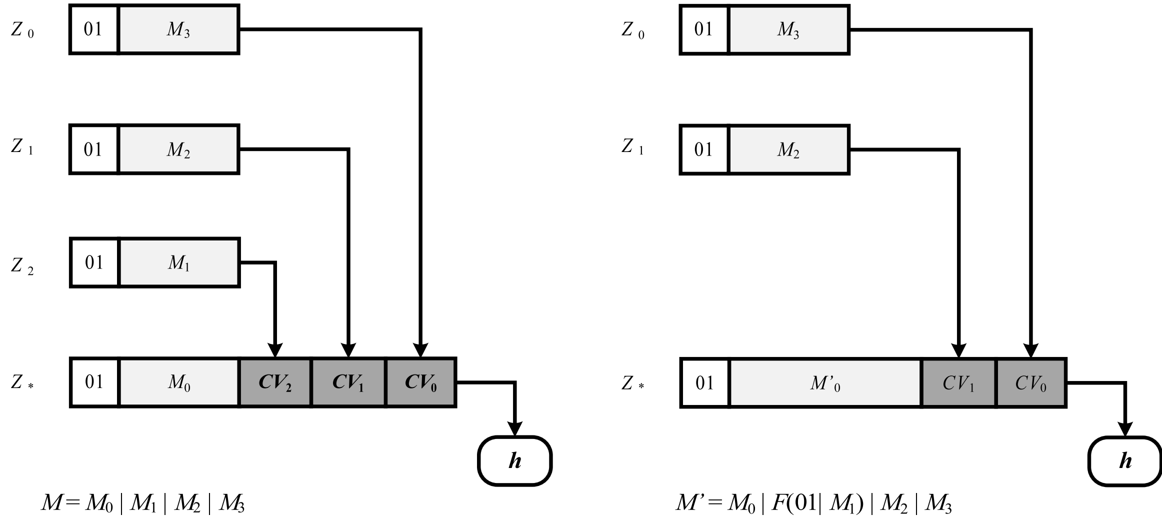

Fig. 4. Example of an inner collision without a collision in $\mathcal{F}$

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.

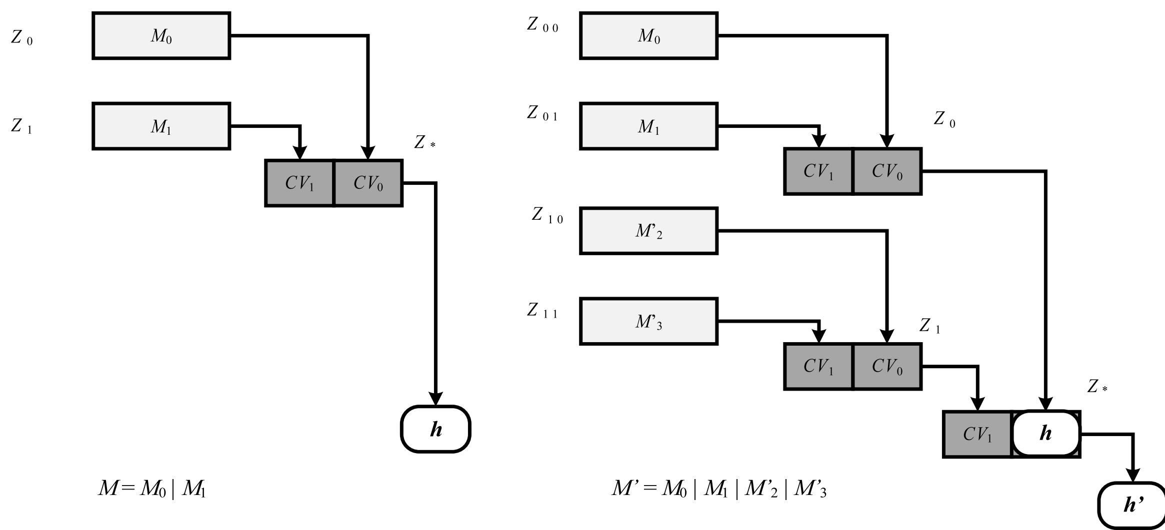

Fig. 5. Example of the generalization of length extension to tree hashing