Sufficient conditions for sound hashing using a truncated permutation

Sander van Dam

supervised by
JOAN DAEMEN

Radboud University

iCIS

10/8/2016

For my thesis I corrected and expanded upon the paper [1]. Due to the nature of my thesis, I was unable to write every part of it and also to include some mandatory parts as related work. Therefore I discussed with my supervisor (Joan Daemen) to write something about the theory behind my thesis and something about related work. That means this thesis consists of two documents. The first is a piece about the theory behind my thesis and the second is the corrected paper.

A word of thanks

First of all I'd like to thank my supervisor Joan Daemen for always giving me both very quick and extremely helpful answers to every question I had. Secondly I'd like to thank Bart Mennink for helping me with the computations of the probabilities for the proof. And, finally, I'd like to thank my parents and my fellow student Daniel Roeven for proofreading my thesis and picking it up at the printshop while I was out of the country.

[1] *Sufficient conditions for sound hashing using a truncated permutation* by Joan Daemen, Tony Dusenge, and Gilles Van Assche

Sufficient conditions for sound hashing using a truncated permutation – a preliminary

Sander van Dam

Radboud Univeristy Nijmegen

1 Introduction

In the paper we give a generic security proof for any tree hashing mode calling a random truncated permutation, where the tree hashing mode satisfies four conditions. We use the indifferentiability framework introduced by Maurer et al. [6] along with Patarin's H-coefficient technique [3]. This piece introduces relevant terminology and concepts and gives a high-level overview of the paper and specifically the changes that have been implemented.

2 Tree hashing with a truncated permutation

A hashing mode needs a compression function to operate, which we can build by truncating a permutation. A permutation is a one-to-one invertible function operating on a finite domain. In this paper we consider permutations that operate on b-bit strings. Meaning there are 2^b possible inputs and 2^b possible outputs (both from the same domain), making $(2^b)!$ possible permutations. A random permutation is a permutation that is uniformly chosen from the set of possible permutations. A truncated permutation is then a permutation whereafter it is mapped to an output only the first n bits are taken. This permutation that acts as the compression function is then used by a tree hashing mode. The basic idea of a tree hashing mode is to build a tree from blocks of input, using the truncated permutation as an underlying function.

Our tree hashing mode takes as input a message and a set of parameters. The set of parameters specifies how the tree is built. In order to reason this, we use a two-step process. First we build a tree template, which is a recipe specifying how messages with a certain length should be hashed. This means it is only dependent on the length of the message and parameters specifying how the tree should be built. This template is then executed to obtain a tree instance, where every node instance is a bit string, constructed according to the tree template. We refer to section 2 and appendix A of the paper for a more in-depth explanation of these concepts. Figure 1 shows a tree template and a tree instance.

3 Provable Security

Provable security usually refers to a mathematical (or formal) proof and thus mostly to cryptography. A system is called provably secure in cryptography if its security conditions can be formally stated in an adversarial model and a rigorous reduction can be given of the security of the system to some underlying hypotheses. Our proof takes place in a random permutation model.

4 Distinguishing advantage

Indistinguishability, sometimes referred to as a "security game", is a framework to determine the *distinguishing advantage* of the attacker. The basic idea is that one has an efficient algorithm called a distinguisher which can access two worlds; the real and the ideal world.

– *The real world* consists of the cryptographic system containing the mode (in our case, the tree hashing mode) we want to test and some idealized components (the permutation)

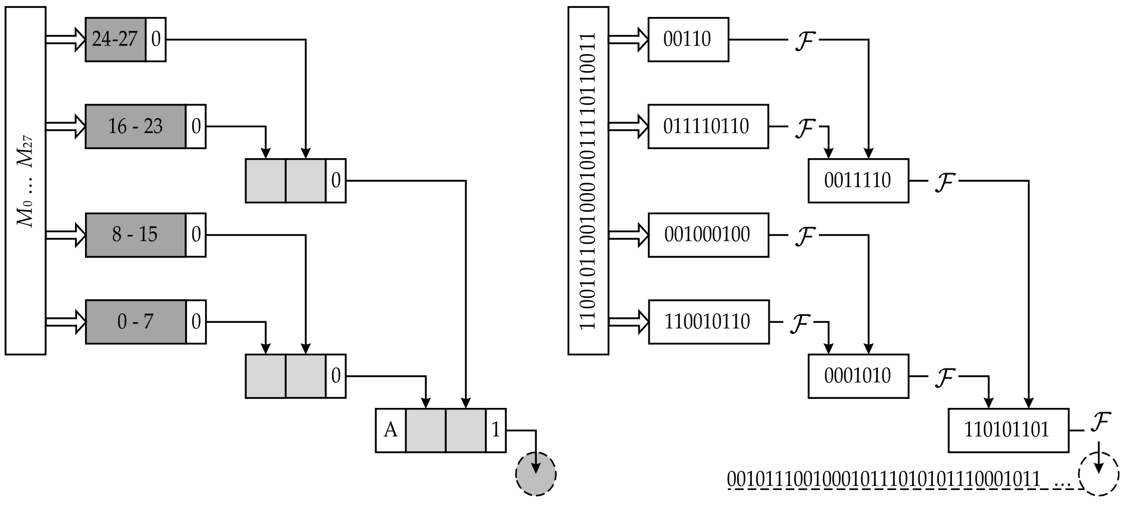

Figure1. Left a tree template, specifying how the tree will be built. On the right a tree instance, with the message and the chaining values filled into the tree.

– *The ideal world* is a perfectly random version of this (an ideal function accessed via the same interface). It handles input and output similarly but the function used returns a random response and acts deterministically.

The attacker will then be faced arbitrarily with either the real or the ideal world and he can query the corresponding system, while not knowing in which world he is. The attacker should not be able to distinguish which world he was in. The distance from a probability of $\frac{1}{2}$ (which is the chance the attacker would have by guessing) is what we call the *advantage* of the attacker. This distance is what we want to bound; it bounds the probability of all generic attacks on the hash function [6]. We use this model to compute an upper bound for the distinguishing advantage of the attacker.

4.1 Random Oracle

A random oracle is a theoretical black box that for every input chooses a (truly) random output[1] uniformly from its output domain. If queried with the same input multiple times, it will always return the same output. The input is a bit string and the output is a fixed-length bit string, in this paper its output is a fixed-length n-bit string.

4.2 Ideal hash function

Our ideal hash function is a function that calls the random oracle, similar to how the tree hashing mode calls the permutation. Parts of the changes made to the paper were because of new insights into the ideal hash function. Before, the ideal hash function presented the message and a set of parameters (specifying how the tree should be built) injectively coded in a string to the random oracle. Now the ideal hash functions generates a template from the message and the parameters, then codes the message and the template injectively in a string, presenting it to the random oracle. There was a condition specifying parameter-completeness, which made sure the tree could be built from the given parameters, but we achieve this anyway if we send the tree template instead to the ideal hash function. This allowed us to have one less condition for our proof. As we don't have to encode the parameters anymore, the construction is more efficient. These changes are reflected in Figure 2, as $\mathcal{G}$ now calls the random oracle, instead of being called by the simulator.

[1] Impractical in reality, but it helps to reason about cryptography, which will be clarified later in this piece.

5 Indifferentiability

A generalization of indistinguishability was introduced by Maurer et al. [6] Public interfaces within the framework can be accessed by anyone, while private interfaces can only be accessed by friendly people (so that the adversary can't directly access the interface). Indifferentiability makes it possible for an adversary to also access a private interface. Indifferentiability applies to our case because the adversary can directly query the permutation, instead of only being able to query it via the tree hashing mode while in the ideal world there is no counterpart for the permutation. We need this for our proof, because in e.g. a standard hash function the permutation would be specified and thus the adversary would be able to "query" it. This means our proof wouldn't be sufficient if we didn't let the adversary be able to directly query the permutation in our indifferentiability set-up. Figure 2 is a visualization of the concept of indifferentiability from the proof in our paper. The private interface doesn't exist in the ideal world, so we need to introduce a *simulator* for the sake of our proof.

5.1 $\mathcal{T}$-consistency and permutation consistency

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. *Permutation consistency* is the property that each member in a permutation couple can only occur once on the transcript. Meaning a transcript with two couples (s, p) and (s', p) with $s \neq s'$ violates permutation consistency.

5.2 The Simulator

A simulator simulates the ideal component called by the mode that is subject to the proof. In our case that means it simulates the permutation (which is also why it is accessed by the same interface as the permutation in figure 2). For this it uses an input retrieval algorithm. The input retrieval algorithm can separate cases, in our paper referring to things that would otherwise happen in the tree template (for a more detailed explanation of the tree template concept, we refer to appendix A of the paper), such that it ensures $\mathcal{T}$-consistency. If the call to S has an input that qualifies as a final node in a tree hash process and the queries that the simulator has received before allow it to reconstruct a full tree instance corresponding to a valid template, it returns a flag "success". Otherwise it returns a flag with an event related to the tree template. Then if flag "success" was returned, the simulator calls the random oracle. Otherwise it chooses a random output from the domain, excluding outputs it has already used, to account for permutation consistency (also keeping in tact the permutation couples).

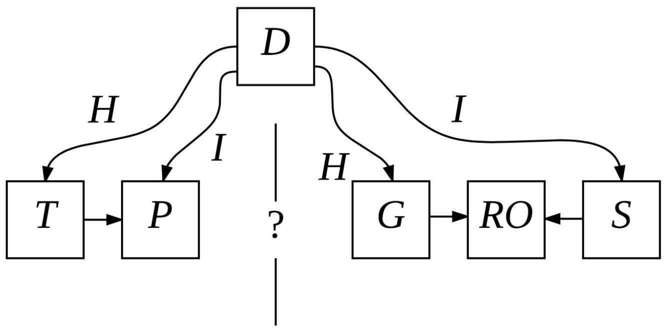

Figure2. Differentiating set-up for tree hashing modes with a truncated permutation

6 Patarin's H-coefficient technique

For our proof we use Patarin's H-coefficient technique. It is a technique used to get a concrete upper bound for the distinguishing advantage of the attacker. As mentioned before, the goal of the adversary is to distinguish between the real world (X) and the ideal world (Y). All interactions between the adversary and the system are written on a transcript T (in our case these are simply the permutation couples (s, p) with $p = f(s)$ and the input/output pairs for queries to $\mathcal{G}/\mathcal{T}$). We use D_X to denote the probability distribution for a given adversary of these transcripts in the real world and D_Y for the probability distribution in the ideal world. The transcripts are divided into "good" and "bad" transcripts on the basis of certain properties. For example, a violation of permutation consistency leads to a bad transcript. The trick is to define the criteria for transcripts to be good or bad such that the bound is as small as possible. Proofs using this technique usually start with defining a set of "bad" and "good" transcripts, such that $T = T_{good} \cup T_{bad}$. We can then bound the differentiating advantage as $Adv(\mathcal{A}) \leq \varepsilon + Pr(D_Y \in T_{bad})$, meaning we need to find the epsilon.If the following can be proven for a certain epsilon

$$\forall \tau \in T_{good} : \frac{Pr(D_X = \tau)}{Pr(D_Y = \tau)} \geq 1 - \varepsilon$$

, we can find the upper bound for the distinguishing advantage. The good and bad transcripts are defined so that ε is small, while at the same time still keeping $Pr(D_Y \in T_{bad})$ small. In some proofs (and in our proof too) the ratio $\frac{Pr(D_X=\tau)}{Pr(D_Y=\tau)}$ turns out to be greater than 1, such that the advantage can be simplified as $Adv(\mathcal{A}) \leq Pr(D_Y \in T_{bad})$ (which is the case in our paper). A way to compute either part of this ratio is to write $Pr(D_X = \tau)$ and $Pr(D_Y = \tau)$ as $\frac{|comp_X|}{|\Omega_X|}$ and $\frac{|comp_Y|}{|\Omega_Y|}$ respectively, where comp denotes a notion of compatibility. This can simplify computations, but does not do so necessarily.

7 Contributions to the paper

The two major contributions that have been made are the updated ideal hash function and the use of Patarin's H-coefficient technique. The updated ideal hash function solves a problem where the parameters had to be explicitly encoded, making the construction less efficient. The change from a series of lemma's to Patarin's H-coefficient technique makes the proof both shorter and a lot more structured, resulting in a more understandable proof.

- No changes have been made to parts 1 and 2.
- Part 3 is new and contains an explanation on Patarin's H-coefficient technique.
- The conditions in part 4 have been reworded to take into account the changes of the ideal function, the parameter completeness condition could be scrapped altogether due to the updated ideal hash function.
- The simulator and the input retrieval algorithms have been rewritten for these changes too. They have also been made compatible with the transcripts used in the H-coefficient technique.
- The bulk of the changes made are in the proof, part 6. A completely different proof method is used, Patarin's H-coefficient technique, meaning next to nothing from the original text remains.

8 Related Work

After Maurer et al. introduced their indifferentiability method, it was first applied to hashing in [4]. The first time indifferentiability was proven for a hashing mode calling a random permutation was [1]. Provable security of tree hashes was researched in [7] and indifferentiability of tree hashing modes based on a permutation was researched in [5]. Our paper has substantial added value, because previous work on the indifferentiability of a compression function construction based on a permutation proves indifferentiability in two phases. In this two-phase approach, each call to the permutation part of the input is fixed to a constant value and hence is not usable for feeding input. So it requires more calls to the permutation for the same input message length, making it less efficient. It is also the first time that indifferentiability is proven for tree hashing modes with an underlying random permutation. The closest related work is [2]. It has similar conditions and proves the same bound, except it does this for tree and sequential hashing modes calling a hash function instead of those calling a permutation.

References

1. 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.
2. ———, *Sufficient conditions for sound tree and sequential hashing modes*, International Journal of Information Security **13** (2014), 335–353, `http://dx.doi.org/10.1007/s10207-013-0220-y`.
3. Shan Chen and John Steinberger, *Tight security bounds for key-alternating ciphers*, (2014), 327–350.
4. 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.
5. 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.
6. 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.
7. 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/`.

Sufficient conditions for sound hashing using a truncated permutation

Joan Daemen[1], Tony Dusenge[2], Gilles Van Assche[1], and Sander van Dam[3]

[1] STMicroelectronics
[2] Université Libre de Bruxelles
[3] Radboud Universiteit Nijmegen

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 four 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 four 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 [9] 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 [9] 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 [5]. 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 [3]. 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 [11] and indifferentiability of permutation-based tree hashing modes was treated in [8], covering the mode used in the SHA-3 candidate MD6 [10]. 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 [4] 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 a block cipher-based construction.

In contrast, this paper addresses modes composed of only one layer. This allows for 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 in a single layer 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 give a high level overview of Patarin's H-coefficient technique in Section 3, and then we introduce a set of simple and easy-to-verify conditions for permutation-based tree hashing modes that result in sound tree hashing in Section 4. After adapting the indifferentiability setting of [7] to permutation-based tree hashing in Section 5, we detail our simulator and the decoding algorithm it calls in Section **??**, and then we provide in Section 6 the indifferentiability proof using Patarin's H-coefficient technique [6]. 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 [4] by Bertoni et al. and cover sequential hashing modes as a special case. In [4], 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 [4], 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 as 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 parent(α). 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 parent(α). 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* $Z_\alpha[x]$, $0 \leq x < |Z_\alpha|$, 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 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 Patarin's H-coefficient technique

This is a high-level overview of Patarin's H-coefficient technique. For a more in-depth explanation we refer to [6].

The goal of the adversary is to distinguish between the real world (X) and the ideal world (Y), written as

$$Adv(\mathcal{A}) = \Delta(X;Y)$$

where $\Delta(X;Y)$ denotes the statistical distance between X and Y. All interactions between the adversary and the system are written on a transcript T (in our case these are simply the permutation couples (s,p) with $p = f(s)$ and the input/output pairs for queries to $\mathcal{G}/\mathcal{T}$). We use D_X to denote the probability distribution for a given adversary of these transcripts in the real world and D_Y for the probability distribution in the ideal world. The transcripts are divided into "good" and "bad" transcripts on the basis of certain properties. For example, a violation of permutation consistency leads to a bad transcript. The trick is to define the criteria for transcripts to be good or bad such that the bound is as small as possible. Proofs using this technique usually start with defining a set of "bad" and "good" transcripts, such that $T = T_{good} \cup T_{bad}$. We can then bound the differentiating advantage as $Adv(\mathcal{A}) \leq \varepsilon + Pr(D_Y \in T_{bad})$, meaning we need to find the epsilon.If the following can be proven for a certain epsilon

$$\forall \tau \in T_{good} : \frac{Pr(D_X = \tau)}{Pr(D_Y = \tau)} \geq 1 - \varepsilon$$

, we can find the upper bound for the distinguishing advantage. The good and bad transcripts are defined so that ε is small, while at the same time still keeping $Pr(D_Y \in T_{bad})$ small. In some proofs (and in our proof too) the ratio $\frac{Pr(D_X=\tau)}{Pr(D_Y=\tau)}$ turns out to be greater than 1, such that the advantage can be simplified as $Adv(\mathcal{A}) \leq Pr(D_Y \in T_{bad})$ (which is the case in our paper). A way to compute either part of this ratio is to write $Pr(D_X = \tau)$ and $Pr(D_Y = \tau)$ as $\frac{|comp_X|}{|\Omega_X|}$ and $\frac{|comp_Y|}{|\Omega_Y|}$ respectively, where comp denotes a notion of compatibility. This can simplify computations, but does not do so necessarily.

Proofs using Patarin's H-coefficient technique consist of three parts. First we carefully define what bad and good transcripts are, using the transcripts themselves. Then we compute the ratio of the probabilities of good transcripts in both worlds to get ε. Finally we compute the probability of bad transcripts in the ideal world to compute the upper bound for the advantage of the adversary $\mathcal{A}$.

4 Sufficient conditions for sound tree hashing

In this paper, we assume that the three conditions for a sound tree hashing mode $\mathcal{T}$, formulated by Bertoni et al. in [4], are fulfilled. In addition, we introduce a fourth condition specific for a tree hashing mode using a truncated permutation. First, let us briefly recall the four conditions of [4].

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

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

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

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

Note that A_{decode} can be specific to the mode but can only use the information contained in the tree instance. Also, this definition includes the case where A_{decode} can identify the chaining values and their attributes in a node from the sole information in that node instance, or the case where it does so from information in that node instance and all its ancestors. In [4], 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 ([4]) *$\mathcal{T}$ 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. *A mode of use $\mathcal{T}$ is* message-complete *if for all combinations $(|M|, A)$ the resulting tree template contains all $|M|$ message pointer bits at least once and there is a deterministic algorithm $A_{message}$ that, given a tree instance S compliant with $\mathcal{T}$, provides the list of positions in S of message pointer bits 0 to $|M| - 1$. The running time of $A_{message}$ shall be $O(m)$ with m the total number of bits in S.*

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

Condition 2 ([4]) *$\mathcal{T}$ is message-complete.*

The third 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 3 ([4]) *$\mathcal{T}$ 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 return 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 4 *$\mathcal{T}$ is leaf-node-anchored.*

5 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 $\mathcal{G}$ using a random oracle $\mathcal{RO}$. We base ourselves on the indifferentiability framework introduced by Maurer et al. in [9] and applied to hash functions by Coron et al. in [7].

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 $\mathcal{G}$, of a simulator $\mathcal{S}$ simulating the permutation and a random oracle $\mathcal{RO}$ for $\mathcal{G}$ and $\mathcal{S}$ to query. 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 $\mathcal{I}^{-1}$, such as in the left system.

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 that which the distinguisher can obtain from the ideal hash function $\mathcal{G}$, like if $\mathcal{S}$ was $\mathcal{P}$ and $\mathcal{G}[\mathcal{RO}]$ was $\mathcal{T}[\mathcal{P}]$. 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 $\mathcal{G}[\mathcal{RO}]$. 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*.

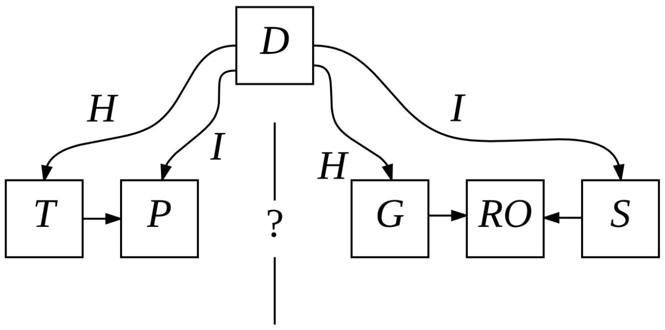

Fig. 1. The differentiating setup

Indifferentiability of $\mathcal{T}[\mathcal{P}]$ 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 $(\mathcal{T}[\mathcal{P}], \mathcal{P})$ 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 $\mathcal{I}^{\pm 1}$, 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 ([4]). *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 $\mathcal{X}(Q_{\mathcal{H}}) = \mathcal{T}[\mathcal{X}(\mathcal{Q}_{\mathcal{I}^{\pm 1}})](Q_{\mathcal{H}})$.*

5.1 The cost of queries

We consider the same cost of queries setting as in [4]. 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.

5.2 $\mathcal{T}$-decoding

For $\mathcal{T}$-consistency, we use a decoding process very similar to the "$\mathcal{T}$-decoding" in [4]. This process extracts an input (M, Z) from a given final node instance, using the first two members of quadruples (s, p, a, c) in T. Input retrieval of a final node s using a table T does not necessarily lead to an input (M, Z). In this context we provide the following definition.

Definition 6. *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, Z) via S.*

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

With respect to $\mathcal{T}$-decoding 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^-. Thus, T^+ and T^- form a partition of T. Our input retrieval procedure is specified in Algorithm 1. It makes use of T^+ and ignores couples in T^-.

We define an n-collision in a table T.

Definition 7. *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 1 is successful for a given node s, it returns an input (M, Z) and we say the node s is *$\mathcal{T}$-decodable*. Otherwise, it returns one of the following *flags*:

– "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^+.
– "incompliant 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.

Algorithm 1 $\mathcal{T}$-decoding

```
1: input: $s$ and set $T^+$
2: output: flag and pair $(M,Z)$ or chaining value $c$
3: Initialization: $J = \{*\}$, $S_* = s$ and $Z_* = \square^{|s|}$
4: while $A_{\text{decode}}(S_J)$ returns flag "final-subtree-compliant" do
5:     Fill in the chaining pointer bits of the expanding index in $Z$
6:     Let $c$ be the chaining value corresponding to expanding index $\beta$ extracted from $S_J$
7:     if there is exactly one entry $(s', p, *, *) \in T^+$ with $\lfloor p \rfloor_n = c$ then
8:         Let $J = J \cup \{\beta\}$, $S_\beta = s'$ and $Z_\beta = \square^{|s'|}$
9:     else if there is no entry $(s', p, *, *) \in T^+$ then
10:        return flag "dead end" and $c$
11:    else { There is more than one entry $(s', p, *, *) \in T^+$ with $\lfloor p \rfloor_n = c$}
12:        return flag "n-collision"
13:    end if
14: end while
15: if $A_{\text{decode}}(S_J)$ returns flag "incompliant" then
16:    return flag "incompliant coding"
17: else
18:    Reconstruct the message pointer bits in $Z_J$ by calling $A_{\text{message}}(S_J)$
19:    Determine the remaining undetermined bits (all frame bits) in $Z_J$ by copying them from $S_J$
20:    Reconstruct $M$ from $S_J$ using the message pointer bits in $Z_J$.
21:    return flag "success" and $(M, Z)$
22: end if
```

5.3 The simulator

Algorithm 2 specifies the simulator $\mathcal{S}[\mathcal{RO}]$. It has the following design principles:

– It is always self-consistent.
– It violates permutation-consistency only with a very small probability.
– It satisfies $\mathcal{T}$-consistency as long as a particular event, called a *$\mathcal{C}$-collision*, does not occur.

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

```
 1: Initialization: $T = T^+ \cup T^- \leftarrow (-, -, -, IV)$

 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: Call Algorithm 1 resulting in a flag and a result
 7: if Algorithm 1 returned flag “success” and $(M, Z)$ then
 8:     Set $p$ to $\mathcal{G}(M, Z)$
 9:     Append $(b - n)$ uniformly and independently drawn random bits to $p$
10:     $T^+ \leftarrow T^+ \cup (s, p,$“success”$, -)$
11: else if Algorithm 1 returned flag “dead-end” and $c$ then
12:     Choose $p$ randomly and uniformly from $\mathbb{Z}_2^b \setminus T_r$
13:     $T^+ \leftarrow T^+ \cup (s, p,$“no success”$, c)$
14: else {Algorithm 1 returned flag “$n$-collision” or “incompliant coding”}
15:     Choose $p$ randomly and uniformly from $\mathbb{Z}_2^b \setminus T_r$
16:     $T^+ \leftarrow T^+ \cup (s, p,$“no success”$, \{\lfloor p \rfloor_n\})$
17: end if
18: return $p$

19: Interface $s = \mathcal{I}^{-1}(p)$ with $s, p \in \mathbb{Z}_2^b$
20: if $\exists(i, p, *, *) \in T$ then
21:     return $s = i$
22: end if
23: Choose $s$ randomly and uniformly from $\mathbb{Z}_2^b \setminus T_l$
24: $T^- \leftarrow T^- \cup (s, p,$“no success”$, \{\lfloor s \rfloor_n\})$
25: return $s$
```

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 quadruples (s, p, a, c) with $s, p \in \mathbb{Z}_2^b$, a being a flag indicating whether c should have a value, and c being the chaining value corresponding to the query. We denote the set of first members of these quadruples by T_l and the set of second members by T_r. When receiving a query $\mathcal{I}^{+1}(s)$ with s already in T_l, the simulator returns the second member of $(s, p, a, c) \in T$ (line 4). Similarly, when receiving a query $\mathcal{I}^{-1}(p)$ with p already in T_r, the simulator returns the first member of $(s, p, a, c) \in T$ (line 21).

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 11 and 14). 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 23). 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 8-9). 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 21) 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 table T has in each quadruple a value c containing n-bit chaining values and the IV (Definition 4). The number of times a value c occurs in the set is its *multiplicity.* Initially, T contains the IV and the simulator adds members to T with a non-empty value c when receiving queries:

- If $\mathcal{T}$-decoding returns ”incompliant coding” or ”n-collision” a query $\mathcal{I}^{+1}(s)$, adds to T the chaining value $\lfloor p \rfloor_n$ for c(line 15).
- A query $\mathcal{I}^{-1}(p)$ adds $\lfloor s \rfloor_n$ (line 28) to T for the variable c. Due to leaf-node anchoring, if the response s is a leaf node, $\lfloor s \rfloor_n =$ IV and otherwise $\lfloor s \rfloor_n$ is a chaining value.

– A query $\mathcal{I}^{+1}(s)$ adds the chaining value c to T iff $\mathcal{T}$-decoding returns "dead-end at c" exception.

The values c in the quadruples in T have 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 8. *There is a $\mathcal{C}$-*collision *in the simulator if a value c in T has at least one member with multiplicity larger than 1.*

The simulator satisfies $\mathcal{T}$-consistency by querying $\mathcal{G}[\mathcal{RO}]$ if necessary. When making a query $\mathcal{I}^{+1}(s)$ with compliant coding, the simulator performs $\mathcal{T}$-decoding to s (line 6). If $\mathcal{T}$-decoding returns (M, Z), the simulator calls $\mathcal{G}[\mathcal{RO}]$ with (M, Z) to guarantee $\mathcal{T}$-consistency (line 8); then it extends the received n-bit value $\mathcal{G}[\mathcal{RO}](M, Z)$ with $(b-n)$ random bits to make a b-bit response p (line 9). 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, Z), the simulator chooses p randomly from all possible values excluding T_r (line 11).

6 The Proof

Notation We'd like to define the notation we use for falling factorials. A falling factorial is defined as $(x)_n = x(x-1)(x-2)\dots(x-n+1)$.

6.1 Definition of good and bad transcripts

The transcripts contain the same information as the state of the simulator, meaning all bad transcripts can be identified by looking at the transcripts themselves.

A bad transcript occurs in two cases

– A C-collision occurs in the simulator, which can be caused by the following mutually exclusive cases
  1. Dead-end at c
  2. n-collision
  3. invalid coding
– Permutation consistency is violated

The transcripts we use are the same as the tables T that the simulator builds. This means each transcript element consists of a quadruple (s, p, a, c) where s and p are the permutation couples. a is a flag with value "success" or "no success". These flags are determined mostly by the $\mathcal{T}$-decoding algorithm and in turn they determine what the value of c is. a has value "success" if $\mathcal{T}$-decoding returns the flag "success". In this case c gets value '-', as this case doesn't cause C-collisions. a has value "no success" in the following cases

– $\mathcal{T}$-decoding returns "dead-end". In this case c gets value c of the chaining value.
– $\mathcal{T}$-decoding returns "n-collision" or "incompliant coding". In these cases c gets the value $\{\lfloor p \rfloor_n\}$.
– The interface I^- is used. In this case c gets the value $\{\lfloor s \rfloor_n\}$.

A bad transcript is then a transcript that has (at least) two distinct entries (s, p, a, c) and (s', p', a', c'), where $c = c'$ or a transcript where permutation inconsistency occurs. We split the transcripts into two mutually exclusive parts, one denotes queries that return success when put through the $\mathcal{T}$-decoding algorithm and the other one denotes queries return a flag different from "success". $T = T_{succes} \cup T_{nosucces}$.

Notation of queries We use lowercase and capital Qs to differentiate between the number of queries up to a certain point and the final number of queries respectively. Queries on T_{succes} are denoted by q_1 and queries to $T_{nosucces}$ are denoted by q_2. This means we have six ways of denoting queries;

- q_1, denotes the queries up to a certain point, that had flag "success".
- q_2, denotes the queries up to a certain point, that didn't have flag "success".
- i, defined as $q_1 + q_2$, meaning it denotes all the queries up to a certain point.
- Q, denotes the total amount of queries after the complete transcript has been built.
- Q_1, denotes the total amount of queries that had flag "success".
- Q_2, denotes the total amount of queries that didn't have flag "success".

Something about cost and queries Due to the nature of the bad transcripts, one would get only good transcripts if one was to solely query G and T. That's why for every query to G/T we add a 'free' (not counted towards the cost) query to S/P respectively. By doing this we can focus on only the simulator for bad transcripts in the ideal world, which simplifies the proof.

6.2 Bounding the ratio $\mathbf{Pr}(D_X = T)/\mathbf{Pr}(D_Y = T)$ for good transcripts

The Real World We want to compute the probability to get a certain good transcript in the real world. We do this by dividing the amount of permutations compatible with this transcript by the total amount of permutations. We use Ω to denote the total amount of permutations and *comp* to denote the amount of compatible permutations.

$$\Omega_X = 2^b!$$

$$comp_X = (2^b - Q)!$$

$$Pr(D_X = T) = \frac{comp_X}{\Omega_X} = \frac{(2^b - Q)!}{2^b!} = (2^{-b})_Q$$

The Ideal World To compute the probability to get a certain good transcript in the ideal world, we split the computation into two parts. Queries that have flag "success" have a probability of $\frac{1}{2^b}$. Queries that don't have flag "success" have probability $\frac{1}{2^b - i}$. To get the total chance of getting a certain good transcript we then have to take the product over all queries of this. We define a function $\delta(i)$ that equals 0 when $i \in q_1$ and equals 1 when $i \in q_2$.

$$Pr(D_Y = T) = \prod_{i=1}^{Q} \frac{1}{2^b - \delta(i)i}$$

The ratio

$$\frac{Pr(D_X = \mathcal{T})}{Pr(D_Y = \mathcal{T})} = \frac{\prod_{q=1}^{Q} 2^b - \delta(q)q}{(2^b)_Q}$$

This is clearly larger than 1, because we can get one of two possibilities; certain values i would lead to $2^b - i$, equaling and thus cancelling out the terms from $Pr(D_X = \mathcal{T})$; while the other values would lead to 2^b, resulting in terms larger than 1. So $\mathbf{Adv}(\mathcal{A}) \leq \mathcal{E} +$ $\Pr(D_Y \in T_{bad})$ is reduced to $\mathbf{Adv}(\mathcal{A}) \leq \Pr(D_Y \in T_{bad})$.

6.3 Bounding the probability of bad transcripts in the ideal world

To compute the probability of a bad transcript in the ideal world, we need to compute the sum of the probabilities of a bad event occurring at each point of building the transcript. Because of the division of queries, a certain query in q_1 can only make the transcript bad by permutation inconsistency and a certain query in q_2 can only make a transcript bad by causing a C-collision. This means that we need to compute two things and add them to each other:

- the chance that the ith query, if it's a "success" query, gives a permutation inconsistency is
$$\frac{i}{2^b}$$
.
- the chance that the ith query, if it isn't a "success" query, gives a C-collision is
$$\frac{q_2}{2^n}$$
. This means that this would be $\frac{i}{2^n}$ when maximized (meaning all queries would be sent to the simulator).

  Meaning the chance for getting a bad transcript is
$$\Pr(D_Y \in \mathcal{T}_{bad}) \leq \sum_{q_1=0}^{Q_1} \frac{q_1}{2^b} + \sum_{q_2=0}^{Q_2} \frac{q_2}{2^n} = \frac{\frac{Q_1(Q_1-1)}{2}}{2^b} + \frac{\frac{Q_2(Q_2-1)}{2}}{2^n} = \frac{Q_1(Q_1-1)}{2^{b+1}} + \frac{Q_2(Q_2-1)}{2^{n+1}}.$$

b is assumed to be far greater than n. This means we can maximize the advantage of the attacker by only querying the simulator (such that $Q_1 = 0$ and $Q_2 = Q$), so that $\frac{Q_1(Q_1-1)}{2^{b+1}} + \frac{Q_2(Q_2-1)}{2^{n+1}}$ is in the worst case scenario $\frac{Q(Q-1)}{2^{n+1}}$. This means our differentiating bound for the worst case scenario is
$$\mathbf{Adv}(\mathcal{A}) \leq \frac{Q(Q-1)}{2^{n+1}} \leq \frac{Q^2}{2^{n+1}}.$$

7 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 four 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. 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.
4. ———, *Sufficient conditions for sound tree and sequential hashing modes*, International Journal of Information Security **13** (2014), 335–353, `http://dx.doi.org/10.1007/s10207-013-0220-y`.
5. 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.
6. Shan Chen and John Steinberger, *Tight security bounds for key-alternating ciphers*, (2014), 327–350.
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, 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.
10. 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/`.
11. 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 4 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 from 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.

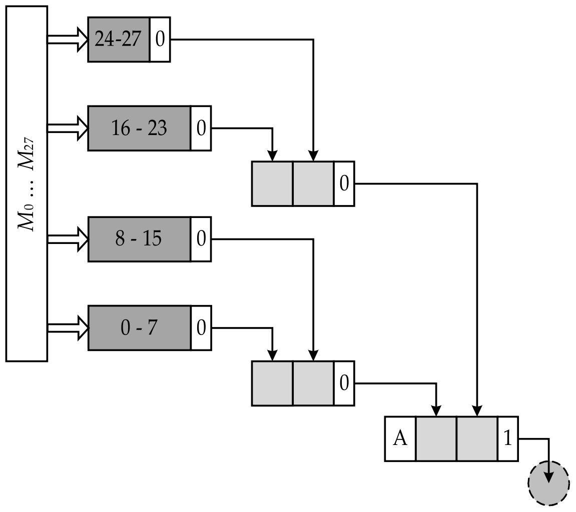

Fig. 2. A tree template.

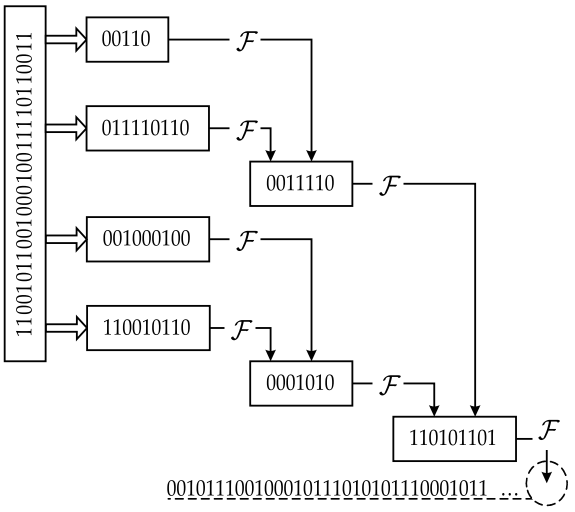

Fig. 3. A tree instance.

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

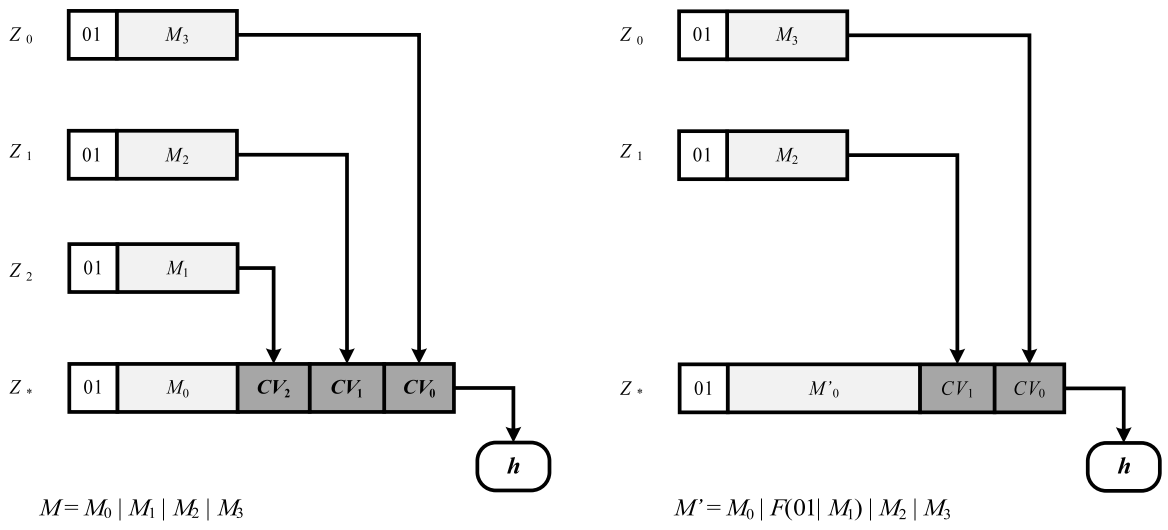
Z_0
01
M_3
Z_1
M_2
Z_2
M_1
Z_*
M_0
CV_2
CV_1
CV_0
h
$M = M_0 \mid M_1 \mid M_2 \mid M_3$
M'_0
$M' = M_0 \mid F(01 \mid M_1) \mid M_2 \mid M_3$

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

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 on the left has three nodes: two leaf nodes and a final node containing the chaining values corresponding to the two leaf nodes. The template on 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.

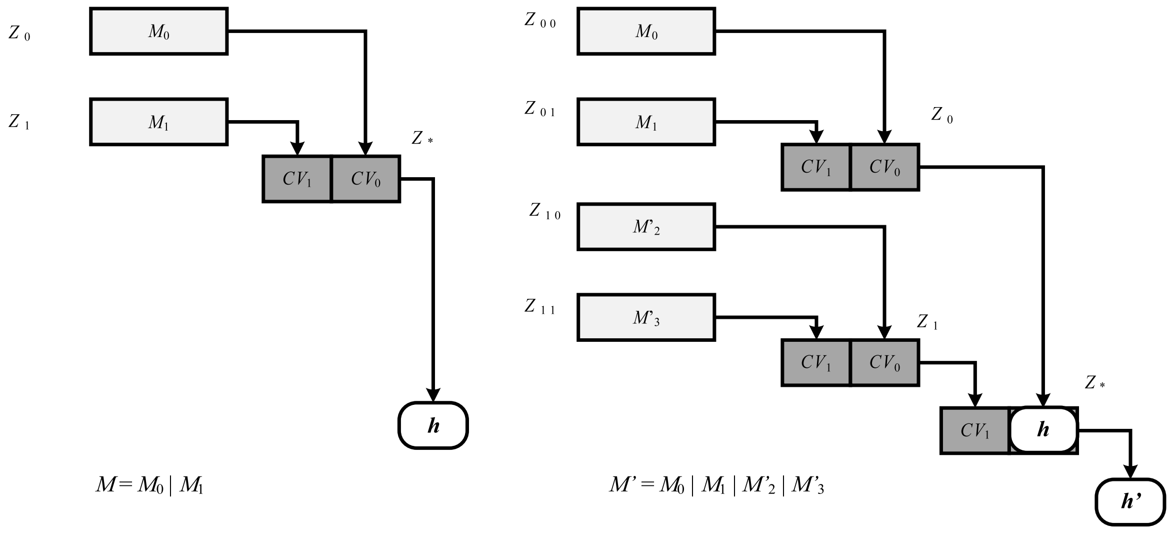

Fig. 5. Example of the generalization of length extension to tree hashing