Block-Cipher-Based Tree Hashing

Aldo Gunsing

Digital Security Group, Radboud University, Nijmegen, The Netherlands
aldo.gunsing@ru.nl

Abstract. First of all we take a thorough look at an error in a paper by Daemen et al. (ToSC 2018) which looks at minimal requirements for tree-based hashing based on multiple primitives, including block ciphers. This reveals that the error is more fundamental than previously shown by Gunsing et al. (ToSC 2020), which is mainly interested in its effect on the security bounds. It turns out that the cause for the error is due to an essential oversight in the interaction between the different oracles used in the indifferentiability proofs. In essence, it reduces the claim from the normal indifferentiability setting to the weaker sequential indifferentiability one. As a matter of fact, this error appeared in multiple earlier indifferentiability papers, including the optimal indifferentiability of the sum of permutations (EUROCRYPT 2018) and the recent ABR$^+$ construction (EUROCRYPT 2021). We discuss in detail how this oversight is caused and how it can be avoided.
We next demonstrate how the negative effects on the security bound of the construction by Daemen et al. can be resolved. Instead of only allowing a truncated output, we generalize the construction to allow for *any* finalization function and investigate the security of this for five different types of finalization. Our findings, among others, show that the security of the SHA-2 mode does not degrade if the feed-forward is dropped and that the modern BLAKE3 construction is secure in principle but that its use of the extendable output requires its counter used for random access to be public. Finally, we introduce the tree sponge, a generalization of the sequential sponge construction with parallel absorbing and squeezing.

Keywords: Hash Functions, Block Ciphers, Tree Hashing, Indifferentiability

1 Introduction

1.1 Hash Functions

Hash functions, which are functions $H : \{0,1\}^* \to \{0,1\}^n$ that compress an arbitrarily-sized message M to a fixed sized output h, form a fundamental part of many cryptographic constructions. In practice they are not built directly, but from a smaller compression function that only takes a fixed sized input, for example $f : \{0,1\}^{2n} \to \{0,1\}^n$. A popular and simple method for this is the Merkle-Damgård construction [Mer89,Dam89], which uses a fixed-sized compression function in a sequential manner to obtain the hash digest. This construction

is sometimes 'strengthened' by appending an encoding of the length of the message to the end, giving a collision resistant hash function as long as the internal compression function is collision resistant, a fact proven by Merkle and Damgård independently.

However, it turned out that collision resistance is not strong enough for some situations. For example, strengthened Merkle-Damgård is susceptible to another attack, called the length extension attack: given the hash digest $H(M)$ of a message M and its length $|M|$ it is possible to compute $H(\text{pad}(M) \parallel M')$ for any M', without knowing the original message M. This is possible as the digest $H(M)$ leaks the internal state of the hash function when the blocks of $\text{pad}(M)$ are processed. By using this state as the initial value, it is straightforward to compute $H(\text{pad}(M) \parallel M')$ for any M'. This is especially troublesome when the function is used in the keyed fashion as $H(K \parallel M)$, noting it should be possible to build a MAC in this way, as the output of the hash should be unpredictable when K is unknown. The attack above shows that this construction is insecure when the hash function is instantiated with (strengthened) Merkle-Damgård, even when the internal compression function is secure. A remedy for this is the HMAC construction [KBC97, Bel06], but this is less efficient and unsatisfying.

This weakness asks for a more sophisticated security analysis for hash functions, namely one that guarantees that the hash function behaves like a random oracle, which gives an randomly generated and independent output for every input. The most general security notion we have is the one of indifferentiability introduced by Maurer et al. [MRH04], which is further applied to hash functions by Coron et al. [CDMP05].

1.2 Previous Work

Using this stronger security notion of indifferentiability, multiple constructions have been shown indifferentiable. For example, prefix-free Merkle-Damgård [CDMP05, LGD+09, LLG11] and Merkle-Damgård with truncation [CDMP05, CN08, LGD+09, LLG11] have been shown indifferentiable, all assuming that the underlying compression function is an ideal compression function. However, this is a very strong and unrealistic assumption: most constructions use a block-cipher-based design with commonly the Davies-Meyer transformation [PGV93] on top. This transformation defines the compression function $f : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b$ as $f(k,x) = \mathcal{E}_k(x) \oplus x$ for a block cipher $\mathcal{E} : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b$. This transformation does make it hard to find collisions or an inverse, but it is not indifferentiable from an ideal compression function and has some undesirable properties. For example, the computation of $\mathcal{E}_k^{-1}(0^b) = x$ for an arbitrary $k \in \{0,1\}^\kappa$ immediately gives a fixed point x where $f(k,x) = \mathcal{E}_k(x) \oplus x = 0^b \oplus x = x$, while finding such fixed point is very difficult for an ideal compression function. This means that one cannot directly use this compression function in a construction that expects an ideal compression function; additional analysis is required. In short, we cannot use constructions based on an ideal compression function to argue security of block-cipher-based constructions.

There have been some constructions shown to be indifferentiable from an ideal compression function [Men13,LMN16,GBK16], but these are very complex and inefficient. There is some work that have dedicated analysis of block-cipher-based constructions: for example, prefix-free Merkle-Damgård with Davies-Meyer [CLNY06,GLC08] or Merkle-Damgård with Davies-Meyer with truncation [GLC08], etc. Another approach for creating a hash function is by processing the message in parallel by using a tree hash, a direction looked at by for example Dodis et al. [DRRS09], where an ideal compression function is used as a primitive. They also show that one can use a truncated permutation to create a compression function that is indifferentiable from a random function, which is later improved upon [CLL19], however, this gives an unoptimized construction and inferior security bounds as the abstraction to an ideal compression function requires extra overhead.

The most promising work for this approach was by Daemen et al. [DMA18] who looked at general tree hashing based on, among others, block ciphers. This paper defined very general properties that a tree hash should satisfy in order to be indifferentiable from a random oracle. Importantly, it supposedly proved that truncation nor a feed-forward is not one of the required properties. However, Samuel Neves pointed out a critical error in the paper indicating that truncation should be required, which was also the formal fix used in the errata by Gunsing et al. [GDM20]. This still leaves us with an unsatisfactory situation as truncation is not always a desirable option when the size of the block cipher is small.

1.3 Our Contribution

1.3.1 Identification of the Flaw In Section 3 we will discuss the nature of the error in more depth. It turns out that there is more to the error than the superficial correction of the bound in [GDM20] indicates. The original paper implicitly ignores some fundamental interaction between the primitive and construction oracles that appear in the definition of indifferentiability as they, after a transformation, incorrectly discard all queries made to the construction oracle. This can only be done when the construction queries are made after the primitive ones. In essence, one could reinterpret the proof to happen in the weaker sequential indifferentiability setting [MPS12] where all primitive queries happen before the construction queries, making this reasoning valid. The same reasoning error occurs in other papers as well [CN08,MPN10,MP15,Lee17,BN18,ABR21]. One [CN08] is about hash functions as well, but it does not make use of any invalid properties, which means that the bounds are not influenced at all and that the proof could be fixed in a straightforward manner. Most other ones [MPN10,MP15,Lee17,BN18] are about the indifferentiability of the sum of permutations and are all based on [MPN10] which contains the same error and the other papers copy the same faulty reasoning. At least the proof of the most recent work [BN18], claiming optimal indifferentiability of the sum of permutations, is significantly impacted. More recent work with the ABR$^+$ construction [ABR21] also contains the same error, although it should not influence its result.

1.3.2 Block-Cipher-Based Tree Hashing with General Finalization

We improve the state of the art by generalizing the the construction of Daemen et al. [DMA18], which only considers a truncated output. We generalize the construction to allow for *any* finalization function and analyze the security of this for five different types of finalization. These constructions and their security properties are summarized in Table 1. Section 4 contains the full security bounds.

Normal Truncation. First of all we re-prove the same construction as in [DMA18] but with more care where we take the error into account. This proves that the original properties are indeed sufficient when truncation is properly account for, as is also shown in the fixed version. Additionally, we generalize some properties slightly, allowing for a more flexible length of the initial value and digest size.

Truncation Without Subtree-Freeness. The most natural way to prevent the length extension attack is by requiring subtree-freeness, where the result of a hash can never be part of another hash. However, in order to prevent this situation, the mode has to use extra bits to mark, for example, the final node. This introduces extra overhead compared to a simpler mode. A different solution is to require more truncation, which was already done previously for the Merkle-Damgård mode specifically. We generalize this solution to tree hashes. It turns out that one can drop the subtree-freeness property by truncating to an even smaller digest, where the exact cost depends on the specific mode. In Section 5.1 we use this to prove the security of the mode used in truncated SHA-2 [SHA08], *without requiring any feed-forward*. This is a significant efficiency improvement, as SHA-2 uses a feed-forward in every compression call.

Chopping. Thirdly we look at chopping instead of truncating. Truncation keeps the first few bits of a string and drops the other bits, while chopping does the inverse: it drops the first few bits and keeps the remaining ones. At a first glance this should not make a difference, as the operations are symmetrical. However, as in our definition of tree hashes we assume that the chaining values are always the result of truncated outputs, it turns out that chopping the final value instead of truncating gives a superior security bound. It essentially voids the stronger requirement with respect to the digest size that the previous mode lost. In short, by using chopping as the finalization instead of truncation, we can drop the subtree-freeness requirement without compromising any security. In Section 5.3 we introduce the tree sponge, a generalization of the sponge construction allowing for parallel absorbing and squeezing, making full use of this result.

Enveloped. Fourthly we look at a generalized enveloped mode. This mode uses a fixed value in the data path of the final compression call, generalizing the Enveloped Merkle-Damgård construction [BR06]. Compared to normal Merkle-Damgård this switches the position of the chaining value from the data input to the key input. This simple change allows for a secure mode that does not require much overhead. We show that this approach generalizes from the sequential Merkle-Damgård mode to general tree-based hashes.

Feed-Forward. Fifthly we look at a feed-forward. We show that by having a feed-forward in the final compression call we do not need any truncation. The conventional approach, which has also been adopted in SHA-2, is that all compression calls use a feed-forward. However, we show that only the final one is required. More importantly, its use does not negate other conditions. For example, subtree-freeness is still required, which is not satisfied in SHA-2 and truncation is still required. In Section 5.2 we use this to analyze the security of the mode used in BLAKE3 [OANW20] when based on a block cipher. We show that the mode is secure in principle, but there is a non-negligible factor in the complexity of the simulator. As a consequence, the extendable output mode becomes insecure when a secret value is used for its offset.

mode	MD+LA+RD	SF	FA	finalization	bits of security	proof
truncation	✓	✓	—	$\lfloor y \rfloor_n$	$\min(m, c/2, b-n)$	Section A
	✓	—	—	$\lfloor y \rfloor_n$	$\min(m, c/2, c-n)$	Section A.6
chopping	✓	—	—	$\lceil y \rceil_n$	$\min(m, c/2, b-n)$	Section A.7
enveloped	✓	✓	full	y	$\min(m, c/2)$	Section B
feed-forward	✓	✓	partial	$x \oplus y$	$\min(m, c/2)$	Section C

Table 1: Summary of the indifferentiability bounds. The conditions MD, LA, RD, SF and FA stand for message-decodability, leaf-anchoring, radical-decodability, subtree-freeness and final-anchoring, respectively. The bits of security are with respect to the number of primitive queries either direct or indirect, where constant and logarithmic terms are ignored. b is the length of the data input of the block cipher, $c \leqslant b$ the capacity of the chaining values, $n \leqslant b$ the digest length and $m \leqslant c$ the length of IV_1 which is used for leaf-anchoring. For the finalization, x denotes the data input to the final block cipher call and y the output. The notations $\lfloor y \rfloor_n$ and $\lceil y \rceil_n$ denote truncation and chopping respectively (i.e. $x = \lfloor x \rfloor_n \parallel \lceil x \rceil_{|x|-n}$ for all n). Note that all chaining values are truncated block cipher outputs, hence the asymmetry between truncation and chopping. For the enveloped and feed-forward modes there is no truncation or similar hence $n = b$.

1.3.3 Comparison of the Variants The different modes above all come with different trade-offs:

– Truncation/Chopping is useful when the block length b is sufficiently large. This is often the case for permutations (which is simply the special case $\kappa = 0$) or large block ciphers. The results show that chopping is basically superior to truncation, when the chaining values are constructed using truncation. If this is the other way around the same result holds by symmetry. The important observation is that dropping a different part of the output in the finalization compared to the internal chaining values is superior to doing the

same operation twice. This is shown by the fact that an extra condition in the form of subtree-freeness can be dropped, or that the efficiency can be significantly improved: the change in the security term from $c - n$ to $b - n$ allows for a larger n with the same security level.
– If a large block size is not available, the enveloped mode is useful. This mode does not have any extra security terms and can achieve $b/2$ bits of security. The biggest disadvantage is that it requires one full extra block cipher call.
– The feed-forward mode is commonly used, but it does not reduce the required conditions compared to the other finalizations. For example, radical-decodability and subtree-freeness are still necessary. Its advantage compared to the enveloped mode is that it only requires partial final-anchoring, making it possible to process a larger message block in the final compression call, increasing its efficiency slightly.

2 Preliminaries

2.1 Notation

Our setup is in the ideal model and we denote $\mathcal{E} : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b$ for an ideal cipher with key length κ and block length b that is uniformly drawn from the set of all such block ciphers. For a bit string x of size at least n bits, we denote $\lfloor x \rfloor_n$ for the first n bits of x (truncation) and $\lceil x \rceil_n$ for the last n bits of x (chopping). Note that for any such x we have that $x = \lfloor x \rfloor_n \,\|\, \lceil x \rceil_{|x|-n}$, where $|x|$ denotes the length of a string x in bits and $\cdot\|\cdot$ concatenation. The uniform random drawing of an element x from a finite set X is denoted by $x \xleftarrow{\$} X$. We denote $n \leqslant b$ for the digest length, $c \leqslant b$ for the capacity, which is the size of the chaining values, and $m \leqslant c$ for the length of IV_1 which will be used for leaf-anchoring.

2.2 Tree Hashing

We follow the same tree hashing paradigm as in [DMA18], but specialized for block ciphers and with some small generalizations.

For our definition we use an explicit intermediate step of template generation in order to be able to reason about the hashing mode. It will consist of three steps: template construction, template execution and a finalization.

A block-cipher-based tree hashing mode $\mathcal{T} = (\mathcal{Z}, \zeta)$ consists of a template generating function $\mathcal{Z} : \mathbb{N} \times \mathcal{A} \to \mathcal{X}$ and a finalization function $\zeta : \{0,1\}^b \times \{0,1\}^b \to \{0,1\}^n$. Here, $\mathcal{A}$ is a set of parameters chosen by the mode and $\mathcal{X}$ is the set consisting of all possible templates and is independent of the mode. The resulting hash function $\mathcal{H}_\mathcal{T}[\mathcal{E}] : \{0,1\}^* \times \mathcal{A} \to \{0,1\}^n$ is based on an ideal cipher $\mathcal{E}$ and computes the hash digest of a message $M \in \{0,1\}^*$ with parameters $A \in \mathcal{A}$ in multiple steps:

– First it computes the tree template $Z = \mathcal{Z}(|M|, A)$ based on the message length $|M|$ and the parameters A. This step is elaborated on in Section 2.2.1.

– Then it executes the template based on an ideal cipher $\mathcal{E}$ to get the in- and output of the final node $(x, y) = \mathcal{Y}[\mathcal{E}](M, Z)$, with $\mathcal{Y}[\mathcal{E}] : \{0,1\}^* \times \mathcal{X} \to \{0,1\}^b \times \{0,1\}^b$. This function is the same for all modes and is elaborated on in Section 2.2.2.
– As the last step it applies the finalization function ζ to the in- and output to get the digest $h = \zeta_x(y)$. If the input x is not used we simply write $\zeta(y)$. This is a generalization compared to [DMA18], where only $\zeta(y) = \lfloor y \rfloor_n$ is considered. The major alternative is the feed-forward defined as $\zeta_x(y) = x \oplus y$. It has to be possible to randomly compute an inverse given some input x and digest h as $y \xleftarrow{\$} \zeta_x^{-1}(h)$.

2.2.1 Template Construction Based on the message length $\mu \in \mathbb{N}$ and the parameters $A \in \mathcal{A}$ a tree template is constructed. The template consists of a number of block cipher calls called nodes, where the inputs of the block ciphers are already determined as virtual bits. These come in three flavors:

– Frame bits: these are fixed bits that are determined solely on the message length μ and the parameters A. For example, these can be used for domain separation or can encode the length of the message.
– Message pointer bits: these bits reference specific bits in M. For example, a bit can reference the 'fifth bit of M', but this bit is currently unknown.
– Chaining pointer bits: these bits refer to the result of another compression call. We require that all first c bits of every compression call are used exactly once (except the final one which is special) and consecutively, where $c \leqslant b$ is the *capacity*. For example, if c consecutive bits refer to the result of (k, x) it will equal $\lfloor \mathcal{E}_k(x) \rfloor_c$ when instantiated.

There is one special block cipher call whose output is not used in the tree. This is called the *final node* and is denoted by final(S) for an instantiation S. A *leaf node* is a node that does not contain any chaining pointer bits.

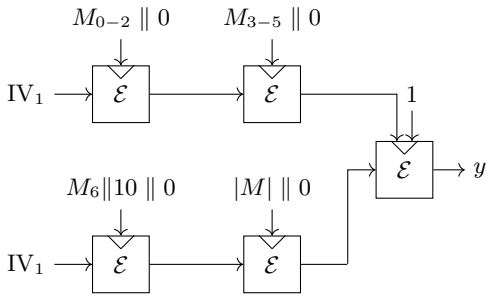

Fig. 1: Basic example of a block-cipher-based tree hashing mode with key size $\kappa = 4$, block size $b = 3$, capacity $c = 3$ and message length $\mu = 7$.

A basic example of a block-cipher-based tree hash is displayed in Figure 1. The different kind of virtual bits in the example are:

- Frame bits: the two IV_1's, the 10 (padding), the four 0's, the 1 and the encoding of the message length $|M|$.
- Message pointer bits: the three blocks of M_{0-2}, M_{3-5} and M_6.
- Chaining pointer bits: the four outputs of a call to $\mathcal{E}$ that are fed into another block cipher call.

Note that the output of the final node is denoted by y, which is not necessarily the hash digest; there is an additional finalization function ζ applied to get the hash digest. In other words, $h = \zeta_x(y)$, with h the hash digest and x and y the in- and outputs of the final node. Furthermore, it is possible to have a capacity $c < b$, in which case the chaining values are truncated.

2.2.2 Template Execution The procedure $\mathcal{Y}[\mathcal{E}](M, Z)$ executes the tree template Z on a message M with compatible length to get the hash digest $h \in \{0,1\}^n$. It uses the following procedure:

- It instantiates the template to get the corresponding tree S. This means that all message pointer bits are instantiated with their respective value of M and similarly for all chaining pointer bits, whose values depend on the block cipher $\mathcal{E}$.
- It gets the inputs of the final node $(k, x) = \mathrm{final}(S)$.
- It computes the output of the final node $y = \mathcal{E}_k(x)$.
- It returns the data input and output of the final node (x, y).

The tree S is represented as a list composed of values of the form (k, x, α), each representing one node. k and x are the key and data inputs to the block cipher and α denotes a different location in the tree. This value means that there is a block cipher call of the form $y = \mathcal{E}_k(x)$ and that the output $\lfloor y \rfloor_c$ is used in the position α. The output of the final node is not used in the tree, which is denoted by $\alpha = \perp$. An example is displayed in Table 2.

2.2.3 Definitions Now we define a few terms that allow us to reason about hashing modes. First of all we define the tree template set $\mathcal{Z}_\mathcal{T}$ as the set of all possible tree templates.

Definition 1 (tree template set [DMA18, BDPV14]). *For a mode of operation $\mathcal{T}$ we define* tree template set $\mathcal{Z}_\mathcal{T} \subseteq \mathcal{X}$ *as the range of the template construction function $\mathcal{Z}$:*

$$\mathcal{Z}_\mathcal{T} = \{\, \mathcal{Z}(\mu, A) \mid \mu \in \mathbb{N}, A \in \mathcal{A} \,\},$$

where μ covers all message lengths and A all parameters.

Next, we define some useful subsets of trees that correspond to the tree templates.

node	k	x	α
0	$\mathbf{1}_{0-2} \parallel 1$	$\mathbf{2}_0$	$\perp$
1	$M_{3-5} \parallel 0$	$\mathbf{3}_0$	$\mathbf{0}_0^k$
2	$\lvert M \rvert \parallel 0$	$\mathbf{4}_0$	$\mathbf{0}_0^x$
3	$M_{0-2} \parallel 0$	IV_1	$\mathbf{1}_0^x$
4	$M_6 \Vert 01 \parallel 0$	IV_1	$\mathbf{2}_0^x$

(a) List representation of the template of the example mode.

node	k	x	α
0	1101	101	$\perp$
1	0010	100	$\mathbf{0}_0^k$
2	1110	010	$\mathbf{0}_0^x$
3	0110	000	$\mathbf{1}_0^x$
4	1010	000	$\mathbf{2}_0^x$

(b) List representation of the instantiation of the example mode, with $\mathrm{IV}_1 = 000$, $M = 011\ 001\ 1$ and random block cipher outputs.

Table 2: List representation of the example mode displayed in Figure 1. The nodes are numbered **0** to **4**, with the key input denoted by $\mathbf{0}_0^k$ and data input denoted by $\mathbf{0}_0^x$ (for node **0**), where the subscript denotes the offset. The k, x and α denote the key input, the data input and the location the output is used ($\perp$ for the final node), respectively.

Definition 2 ((sub)tree set [DMA18, BDPV14]). *We say that a tree S* complies *with a template Z if it has the same tree topology and the frame bits in Z match those in S.*

For a mode of operation $\mathcal{T}$ we define the following sets:

- *$\mathcal{S}_{\mathcal{T}}$ is the set of all trees S such that there exists a $Z \in \mathcal{Z}_{\mathcal{T}}$ such that S complies with Z.*
- *$\mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$ is the set of trees S such that there exists a $S' \in \mathcal{S}_{\mathcal{T}}$ such that S is a proper subtree of S'.*
- *$\mathcal{S}_{\mathcal{T}}^{\mathrm{leaf}}$ is the subset of trees $S \in \mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$ such that there exists a $S' \in \mathcal{S}_{\mathcal{T}}$ such that S is a proper subtree of S' and S contains all its descendants in S'.*
- *$\mathcal{S}_{\mathcal{T}}^{\mathrm{final}}$ is the subset of trees $S \in \mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$ such that there exists a $S' \in \mathcal{S}_{\mathcal{T}}$ such that S is a proper subtree of S' and S contains the root node of S'.*

Intuitively, $\mathcal{S}_{\mathcal{T}}$ denotes the set of all possible trees, $\mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$ the set of all their proper subtrees, $\mathcal{S}_{\mathcal{T}}^{\mathrm{leaf}}$ the set of trees that cannot be extended backwards, i.e. ones that contain all necessary leafs and $\mathcal{S}_{\mathcal{T}}^{\mathrm{final}}$ the set of proper subtrees that contain the final node. Now we define the notion of a *radical*, which is an essential part of our requirements. Intuitively, a radical identifies bit positions which can only refer to a chaining value, but has no such value associated yet.

Definition 3 (radical [DMA18]). *A* radical α *in a tree instance $S \in \mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$ identifies c bit positions such that no node is attached to α in S, but in any $S' \in \mathcal{S}_{\mathcal{T}}$, with S a subtree of S', the value located by α is a chaining value (CV). This value is called the* radical CV *and is denoted as $S[\alpha]$.*

For example, take a Merkle-Damgård mode with the domain separation on the leaf node. That is, all templates are of the form displayed in Figure 2. Given just the final two nodes (which is $\mathbf{0} : (\mathbf{1}, M_4 \Vert 0, \perp); \mathbf{1} : (x, M_3 \Vert 0, \mathbf{0}^x)$ in the list

representation) with arbitrary M_3, M_4 and data input x for the node with key $M_3\|0$, this x will always be a chaining value. Only for leaf nodes the data input is not a chaining value, but we know that this is not a leaf node by the domain separation. This means that this position ($\mathbf{1}^x$) is a radical.

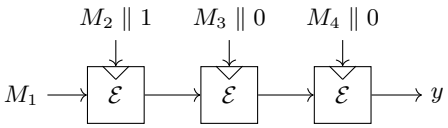

Fig. 2: A Merkle-Damgård mode with leaf-node separation. This means that data inputs are unambiguously either a chaining value or a message block, hence non-leaf subtrees have radicals.

Another example is displayed in Figure 3. This mode is similar, but with domain separation on the final node instead of the leaf node. However, this does mean that the similar final two nodes as before (which is $\mathbf{0} : (\mathbf{1}, M_4\|1, \bot); \mathbf{1} : (x, M_3\|0, \mathbf{0}^x)$ in the list representation) with arbitrary M_3, M_4 and data input x for the node with key $M_3\|0$ do not have a radical. The data input x could be a chaining value, but it could also represent a message block, in which case the message would be $x\|M_3\|M_4$. As the role of this x is ambiguous its position ($\mathbf{1}^x$) is not a radical, nor is any other position.

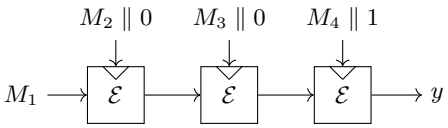

Fig. 3: A Merkle-Damgård mode with final-node separation. This means that data inputs are ambiguous and no radicals exist.

2.2.4 Conditions We look at the conditions that a tree hashing mode has to satisfy in order to be secure. Message-decodability states that a message can be successfully extracted from a full tree, leaf-anchoring requires the first few bits of every node to either denote a fixed value or a chaining value and radical-decodability states that the previously defined radicals can efficiently be identified from chosen set $\mathcal{S}_{\mathcal{T}}^{\text{rad}}$. In general, message-decodability is trivially satisfied and leaf-anchoring is a straightforward property. Radical-decodability is a more tricky definition and sometimes requires more work to show.

Definition 4 (message-decodability [DMA18]). *A mode of operation $\mathcal{T}$ is* message-decodable *if there is an efficient function* extract() *that on input of $S \in \mathcal{S}_{\mathcal{T}}$ returns the template Z it complies with and the message M, and on input of $S \notin \mathcal{S}_{\mathcal{T}}$ returns $\perp$.*

Definition 5 (leaf-anchoring [DMA18]). *A mode of operation $\mathcal{T}$ is* leaf-anchored *if for every template $Z \in \mathcal{Z}_{\mathcal{T}}$, the first $m \leqslant c$ of every leaf node encode $\mathrm{IV}_1 \in \{0,1\}^m$ as frame bits and the first c bits of every non-leaf node are chaining pointer bits.*

Definition 5 is a minor generalization of the original definition in [DMA18] as it allows for a more flexible length of IV_1.

Definition 6 (radical-decodability [DMA18]). *A mode of operation $\mathcal{T}$ is* radical-decodable *if there exists a set $\mathcal{S}_{\mathcal{T}}^{\mathrm{rad}}$ such that all trees $S \in \mathcal{S}_{\mathcal{T}}^{\mathrm{rad}}$ have a radical, and there exists an efficient deterministic function* radical() *that returns a radical upon presentation of an $S \in \mathcal{S}_{\mathcal{T}}^{\mathrm{rad}}$, and $\perp$ otherwise. The set $\mathcal{S}_{\mathcal{T}}^{\mathrm{rad}}$ must satisfy $\mathcal{S}_{\mathcal{T}}^{\mathrm{final}} \subseteq \mathcal{S}_{\mathcal{T}}^{\mathrm{rad}} \subseteq \mathcal{S}_{\mathcal{T}}^{\mathrm{sub}} \setminus \mathcal{S}_{\mathcal{T}}^{\mathrm{leaf}}$.*

In essence, radical-decodability requires the existence of an efficient function radical() that finds chaining values in a tree such that:

1. it only finds radicals (so they are chaining values in every possible tree),
2. it always reconstructs the full tree when starting at the final node and extending the tree based on the found radicals.

Note that $\mathcal{S}_{\mathcal{T}}^{\mathrm{final}}$ and $\mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$ only contain *proper* subtrees, so if a full tree consists of a single node, i.e. it hashes a message consisting of a single block, it is not part of $\mathcal{S}_{\mathcal{T}}^{\mathrm{final}}$ or $\mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$ hence it should not be in $\mathcal{S}_{\mathcal{T}}^{\mathrm{rad}}$ as the tree is already complete.

As a first example we take a Merkle-Damgård mode with leaf-anchoring, depicted in Figure 4. We take $\mathcal{S}_{\mathcal{T}}^{\mathrm{rad}} = \mathcal{S}_{\mathcal{T}}^{\mathrm{sub}} \setminus \mathcal{S}_{\mathcal{T}}^{\mathrm{leaf}}$, the largest possible set. This means that we have to identify a radical for any subtree that is not in $\mathcal{S}_{\mathcal{T}}^{\mathrm{leaf}}$. We do this by identifying radicals by the absence of IV_1. Our function radical() works as follows: first we identify the leftmost node (which exists as it is a sequential mode), then we return its data input if it is not equal to IV_1 and return $\perp$ otherwise. An implementation of radical() is illustrated in Algorithm 2. We check the two requirements for radical-decodability.

1. Radicals: indeed, by definition of the mode any data input that is not IV_1 has to be a chaining value.
2. Reconstruction: strictly speaking, it does not satisfy this property. If a chaining value is equal to IV_1 the function will stop too soon. However, this has a negligible probability of occurring and in fact our proof already takes it into account. This means that we can assume that no chaining value hits IV_1, hence our function will always reconstruct the message.

An implementation of extract() is also illustrated in Algorithm 3. It is similar to the procedure radical(), but it extracts the message instead of the radicals.

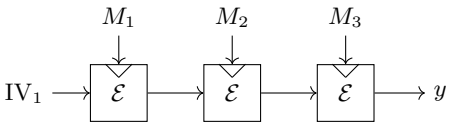

Fig. 4: A Merkle-Damgård mode with leaf-anchoring. Radical-decodability can be achieved by identifying radicals by the absence of IV_1 in the data input.

Algorithm 1 Helper function to lookup the node pointing to a location

Interface: lookup(S, α')
  for all i : $(k, x, \alpha) \in S$ **do**
    if $\alpha = \alpha'$ **then**
      return i : (k, x, α)
    end if
  end for
  return $\perp$

As a second example we take a Merkle-Damgård mode with final-node separation and length encoding, but without leaf anchoring, depicted in Figure 5. This means that we cannot identify radicals by the absence of IV_1 anymore. However, we can make use of the other properties. We take $\mathcal{S}_{\mathcal{T}}^{\mathrm{rad}} = \mathcal{S}_{\mathcal{T}}^{\mathrm{final}}$, the smallest possible set. This means that we only have to identify a radical for any subtree that contains the final node. As we have final-node separation we can identify this. If the given tree does not contain a final node, we always return $\perp$ as we do not know how long the message is, which is allowed by the definition of radical-decodability as those trees are not in $\mathcal{S}_{\mathcal{T}}^{\mathrm{rad}}$. If a tree does contain a final node we can read the length of the message from it. Using this, we know the number of block cipher calls, from which we can deduce whether the data input is a chaining value or a message block, satisfying radical-decodability. An implementation of radical() is illustrated in Algorithm 4. The procedure extract() is not illustrated but is again very similar to radical(), but it extracts the message instead of the radicals.

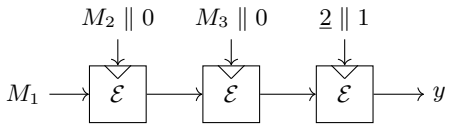

Fig. 5: A Merkle-Damgård mode with final-node separation and length encoding. Radical-decodability can be achieved by identifying the final node and using the length to know when to stop extending the tree.

Algorithm 2 Implementation of radical() for the mode pictured in Figure 4

```
Interface: radical(S)
  α' ← ⊥                                          ▷ initialize with the final node
  while lookup(S, α') ≠ ⊥ do
    i : (k, x, α) ← lookup(S, α')                 ▷ lookup the chaining value
    if x = IV_1 then                              ▷ apply leaf-anchoring
      return ⊥                                    ▷ full tree
    end if
    α' ← i_0^x          ▷ the data input contains the next potential radical
  end while
  return α'                                       ▷ radical found
```

Algorithm 3 Implementation of extract() for the mode pictured in Figure 4

```
Interface: extract(S)
  M' ← ε                                          ▷ initialize with the empty string
  α' ← ⊥
  while lookup(S, α') ≠ ⊥ do
    i : (k, x, α) ← lookup(S, α')
    M' ← k ‖ M'                    ▷ the key input contains a message block
    if x = IV_1 then
      return (M', ∅)               ▷ return the message and no parameters
    end if
    α' ← i_0^x
  end while
  return ⊥
```

Finally we may revisit the example in Figure 3. We already noted that no radicals exist for this mode, meaning that only $\mathcal{S}^{\text{rad}}_{\mathcal{T}} = \varnothing$ is possible. However, this contradicts the requirement that $\mathcal{S}^{\text{final}}_{\mathcal{T}} \subseteq \mathcal{S}^{\text{rad}}_{\mathcal{T}}$, hence this construction is not radical-decodable.

Now we define subtree-freeness, which is a generalization of the problem in length-extension attacks and states that a full tree can never be a subtree of a different tree.

Definition 7 (subtree-freeness [DMA18]). *A mode of operation $\mathcal{T}$ is* subtree-free *if*

$$\mathcal{S}_{\mathcal{T}} \cap \mathcal{S}^{\text{sub}}_{\mathcal{T}} = \varnothing.$$

Next, we introduce some new conditions not present in [DMA18]. These are about full and partial final-anchoring. Full final-anchoring states that the full input of the final node should contain a fixed value, while partial final-anchoring additionally allows for a single chaining value to be present.

Definition 8 (full final-anchoring). *A mode of operation $\mathcal{T}$ that is leaf-anchored is* fully final-anchored *if for every template $Z \in \mathcal{Z}_{\mathcal{T}}$ the first b bits of the final node encode $\text{IV}_2 \in \{0,1\}^b$ as frame bits.*

Algorithm 4 Implementation of radical() for the mode pictured in Figure 5

```
Interface: radical(S)
  i : (k, x, α) ← lookup(S, ⊥)
  ℓ ‖ s ← k                                          ▷ with |s| = 1
  if s = 0 then
      return ⊥                                       ▷ fail if it is not a final node
  end if
  α′ ← i_0^x
  for j ← 0 to ℓ do                                  ▷ process ℓ blocks based on the length encoding
      if lookup(S, α′) = ⊥ then
          return α′
      end if
      i : (k, x, α) ← lookup(S, α′)
      α′ ← i_0^x
  end for
  return ⊥
```

Strictly speaking, a mode cannot satisfy both leaf-anchoring and full final-anchoring as the definitions conflict on the first c bits of the final node. Leaf-anchoring requires these bits to be chaining pointer bits, while full final-anchoring requires them to encode IV_2. However, in our definition, we make an exception for this: the requirement of full final-anchoring overwrites the one of leaf-anchoring for the final node.

The final block cipher call of a mode with full final-anchoring will always look like the example in Figure 6.

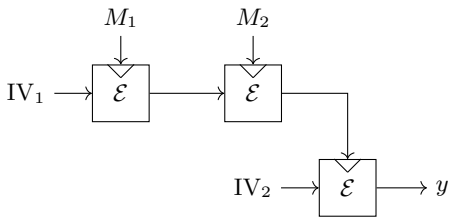

Fig. 6: Example mode using full final-anchoring.

Definition 9 (partial final-anchoring). *A mode of operation $\mathcal{T}$ that is leaf-anchored is* partially final-anchored *if for every template $Z \in \mathcal{Z}_{\mathcal{T}}$ the following holds for the final node:*

- *When it is a leaf node, it encodes IV'_1 as frame bits, where $\mathrm{IV}'_1 \in \{0,1\}^b$ with $\lfloor \mathrm{IV}'_1 \rfloor_m = \mathrm{IV}_1$.*
- *When it is a non-leaf node, the first c bits are chaining pointer bits and its last $b - c$ bits encode $\mathrm{IV}_2 \in \{0,1\}^{b-c}$ as frame bits.*

There may be P different possibilities for IV_2*, denoted by the set* $\mathcal{IV}_2$.

An example of a mode using partial final-anchoring is depicted in Figure 7.

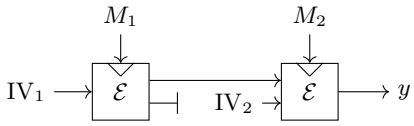

Fig. 7: Example mode using partial final-anchoring.

2.3 Indifferentiability

We use the indifferentiability framework introduced by Maurer et al. [MRH04] applied to hash functions by Coron et al. [CDMP05].

Definition 10. *Let* $\mathcal{T} = (\mathcal{Z}, \zeta)$ *be a hashing mode, with template generating function* $\mathcal{Z} : \mathbb{N} \times \mathcal{A} \to \mathcal{X}$ *and finalization function* $\zeta : \{0,1\}^b \times \{0,1\}^b \to \{0,1\}^n$*, based on an ideal cipher* $\mathcal{E} : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b$ *and let* $\mathcal{Y}[\mathcal{E}] : \{0,1\}^* \times \mathcal{X} \to \{0,1\}^b \times \{0,1\}^b$ *be the template execution function as described in Section 2.2.2. Let* $\mathcal{RO}$ *be a random oracle with the same domain and range as* $\zeta \circ \mathcal{Y}[\mathcal{E}]$ *and* $\mathcal{S}$ *be a simulator with oracle access to* $\mathcal{RO}$*. The indifferentiability advantage of a distinguisher* $\mathcal{D}$ *is defined as*

$$\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) = \left| \mathbb{P}\left[\mathcal{D}^{\zeta\circ\mathcal{Y}[\mathcal{E}],\mathcal{E},\mathcal{E}^{-1}} = 1\right] - \mathbb{P}\left[\mathcal{D}^{\mathcal{RO},\mathcal{S}[\mathcal{RO}],\mathcal{S}^{-1}[\mathcal{RO}]} = 1\right] \right|,$$

where $\mathcal{D}$ *can only make construction queries* (M, Z) *such that* $Z = \mathcal{Z}(|M|, A)$ *for some* $A \in \mathcal{A}$.

2.4 Elementary Results

Our proof will rely on the H-coefficient technique introduced by Patarin [Pat08] and modernized by Chen and Steinberger [CS14]. Let $\mathcal{D}$ be a information-theoretic deterministic distinguisher trying to distinguish $\mathcal{O}_1 = (\zeta \circ \mathcal{Y}[\mathcal{E}], \mathcal{E}, \mathcal{E}^{-1})$ and $\mathcal{O}_2 = (\mathcal{RO}, \mathcal{S}[\mathcal{RO}], \mathcal{S}^{-1}[\mathcal{RO}])$. Let ν be the *view* of $\mathcal{D}$ after interacting with either oracle, consisting of a list of all its queries made. Let $\mathcal{D}_{\mathcal{O}_1}$ denote the probability distribution of views of $\mathcal{D}$ interacting with $\mathcal{O}_1$ and $\mathcal{D}_{\mathcal{O}_2}$ likewise for $\mathcal{O}_2$. A view ν is *attainable* if it can be observed by $\mathcal{D}$ in the ideal world, i.e. $\mathbb{P}[\mathcal{D}_{\mathcal{O}_2}] > 0$. We define $\mathcal{V}$ as the set of all attainable views. The H-coefficient technique states the following for $\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D})$.

Lemma 1 (H-coefficient Technique [Pat08,CS14]). *Let* $\mathcal{O}_1 = (\zeta \circ \mathcal{Y}[\mathcal{E}], \mathcal{E}, \mathcal{E}^{-1})$ *and* $\mathcal{O}_2 = (\mathcal{RO}, \mathcal{S}[\mathcal{RO}], \mathcal{S}^{-1}[\mathcal{RO}])$*. Let* $\mathcal{D}$ *be a deterministic distinguisher and*

$\mathcal{V} = \mathcal{V}_{\text{good}} \cup \mathcal{V}_{\text{bad}}$ be a partition of the set of views into good and bad views. Let $\varepsilon \geqslant 0$ be such that for all $\nu \in \mathcal{V}_{\text{good}}$:

$$\frac{\mathbb{P}\left[\mathcal{D}_{\mathcal{O}_1} = \nu\right]}{\mathbb{P}\left[\mathcal{D}_{\mathcal{O}_2} = \nu\right]} \geqslant 1 - \varepsilon.$$

Then $\mathsf{Adv}^{\text{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) \leqslant \varepsilon + \mathbb{P}\left[\mathcal{D}_{\mathcal{O}_2} \in \mathcal{V}_{\text{bad}}\right]$.

We will have to upper bound the probability of multi-collisions. We use the following result based on Choi et al. [CLL19] for this.

Lemma 2. *Suppose we have a sequence of s elements where every element is randomly chosen from $\{0,1\}^a$ and let F_x denote the number of elements that hit the value $x \in \{0,1\}^a$. Then we have that*

$$\mathbb{E}\left[\max_x F_x\right] \leqslant \frac{2s}{2^a} + \ln(s) + a + 1.$$

Proof. We follow the same reasoning as in Equation 3 in [CLL19].

The variable F_x can be seen as the sum of s Bernoulli variables B_i with parameter $p = 1/2^a$. As

$$\mathbb{E}\left[e^{tB_i}\right] = 1 + p(e^t - 1) \leqslant 1 + 2pt \leqslant e^{2pt}$$

for any $0 < t \leqslant 1$, we can apply the Chernoff bound to get

$$\mathbb{P}\left[F_x \geqslant j\right] \leqslant e^{-t(j-2ps)} = e^{-t(j-2s/2^a)} \leqslant \frac{1}{s \cdot 2^a}.$$

for any $j \geqslant 2s/2^a + \ln(s \cdot 2^a)/t$. Therefore we have

$$\begin{aligned}
\mathbb{E}\left[\max_x F_x\right] &= \sum_{j \geqslant 1} \mathbb{P}\left[\max_x F_x \geqslant j\right] \\
&\leqslant \frac{2s}{2^a} + \ln(s \cdot 2^a)/t + \sum_{j > 2s/2^a + \ln(s \cdot 2^a)/t} \mathbb{P}\left[\max_x F_x \geqslant j\right] \\
&\leqslant \frac{2s}{2^a} + \ln(s \cdot 2^a)/t + \sum_{j > 2s/2^a + \ln(s \cdot 2^a)/t} \sum_{x \in \{0,1\}^a} \mathbb{P}\left[F_x \geqslant j\right] \\
&\leqslant \frac{2s}{2^a} + \ln(s \cdot 2^a)/t + \frac{s \cdot 2^a}{s \cdot 2^a} \\
&\leqslant \frac{2s}{2^a} + (\ln(s) + a)/t + 1.
\end{aligned}$$

Setting $t = 1$ gives the desired result. □

3 Errors

The faulty bounds in the original analysis in [DMA18] were superficially corrected in [GDM20]. Nevertheless, a more thorough investigation reveals that the root cause is more fundamental and applies to many earlier indifferentiability proofs.

3.1 Description of the Flaw

The main error is that the calls to the random oracle from the simulator are considered to be random. The output of the random oracle is indeed random. However, as the distinguisher also has access to this random oracle, it might know the output beforehand. Let us take a look at a simple example of this. Let $\mathcal{D}$ be a distinguisher that makes the following queries, where $\mathcal{T}$ is the construction oracle of a simple Merkle-Damgård-like mode and $\mathcal{E}$ the primitive oracle, with the message M consisting of one block:

- query $\mathcal{T}(M) = h$,
- query $\mathcal{E}_h(\mathrm{IV}_1) = y$,
- query $\mathcal{E}_M(\mathrm{IV}_1) = h$.

The final output needs to be h, as this computes the hash of M.

As the simulator simulates $\mathcal{E}$, the only queries that it sees are (h, IV_1) with output y and (M, IV_1) with output h. Although this h comes from the random oracle, its value is magically equal to the key input of the first query, from the simulator's point of view.

When presented in this way, it may be obvious that the output of (M, IV_1) cannot be considered random as it is part of the fundamental interaction between the oracles. However, it becomes more subtle when it is more abstractly presented and when some simplifications are made. It is very common to represent the queries to the oracle as two separate lists. In the above example we would get the list $\mathcal{M} = ((M, h))$ for the construction oracle and the list $\mathcal{L} = ((h, \mathrm{IV}_1, y), (M, \mathrm{IV}_1, h))$ for the primitive oracle. These lists contain duplicate information, as from either (M, h) or (M, IV_1, h) we can derive the fact that the hash of the message M is equal to h. In order to simplify the analysis we might be tempted to drop one of these queries. For example, we might drop all queries from $\mathcal{M}$ which can be derived from $\mathcal{L}$. In this case this means that $\mathcal{M}$ becomes empty and we would only have to consider $\mathcal{L} = ((h, \mathrm{IV}_1, y), (M, \mathrm{IV}_1, h))$. However, this is a faulty reasoning as the output of (M, IV_1) is always h and cannot be considered to be randomly generated.

3.2 Occurrence in Other Works

Besides [DMA18], where the error had a significant influence on the proven security bound, the same error appeared in multiple other papers as well [CN08, MPN10, MP15, Lee17, BN18, ABR21].

- In [CN08], the authors use the following reasoning in the Section 'Some Important Observations':
  "Thus, we assume that $\mathcal{A}$ do not make any $\mathcal{O}_1$-query which is computable from the previous query-responses of $\mathcal{O}_2$. More particularly, we can remove all those $\mathcal{O}_1$-queries from the final view which are computable from the query-responses of $\mathcal{O}_2$."
  Here, $\mathcal{A}$ denotes the distinguisher, $\mathcal{O}_1$ the construction oracle and $\mathcal{O}_2$ the primitive oracle. The first sentence is correct, but the second one is not.

The error can be fixed in a straightforward manner by not removing those queries from the final view. The probabilities have to be computed in a slightly different way, as some output will be known. However, as the only way these known outputs are used is in upper bounding the probability of a mutlicollision, whose analysis remain exactly the same, this does not have any influence on the bound.

- The same error appeared in work on the indifferentiability of the sum of permutations [MPN10, MP15, Lee17, BN18]. In the original paper [MPN10] they apply a common transformation to the distinguisher $\mathcal{D}$. The new distinguisher $\mathcal{D}'$ is the same as $\mathcal{D}$, but it additionally verifies all the construction queries. This transformation is fine as it simplifies some analysis, we use this transformation as well. However, after this transformation they simply ignore the queries to the random oracle, which is not correct. The other papers [MP15, Lee17, BN18] are based on this and also copy the same error. As a matter of fact [DMA18] copied the approach from them as well.
  Looking at the most recent work with the best bound [BN18], the error does have a significant impact on the proof. The primitive queries are viewed as random variables, without taking possible construction queries into account. These queries influence the distributions, which has a significant effect on the proof as these distributions are used in the χ^2-technique. This does not mean that the bound is necessarily incorrect, but the proof has to be significantly changed.
- The same error appeared in recent work of Andreeva et al. [ABR21]. In Section 5 the authors show that their ABR$^+$ mode is indifferentiable. However, again, after the common transformation to $\mathcal{D}'$ at the start of Section 5.3 the construction queries are incorrectly ignored. Nevertheless, in this case the error should not have an impact on the result. As the construction is fixed-length and uses a different random function (not permutation) in all locations, including the final one, the knowledge of the construction results will be independent and not influence other parts. In short, although the paper makes the same common error, the result should still be correct because of the specifics of the construction.

3.3 Possible Cause and Resolution

A possible cause for the error can be from the notations of the views. It is convenient to denote the interaction of the construction and primitive oracles separately. However, this notation can be misleading. It implies that the interaction between the two different oracles is somewhat disconnected, while this is not the case. The common error of dropping all the queries to the construction oracles is basically equivalent to changing the security model. Instead of always having access to both oracles, the adversary instead operates in two phases: first it is only allowed access to the primitive oracle and after it is done with this oracle, it is allowed to only query the construction oracle. This exact model does actually exist as sequential indifferentiability [MPS12]. In this setting the previously faulty transformation is valid, meaning that any proofs using it can

be reinterpreted as occurring in the weaker sequential indifferentiability model, making them still proving a positive, but significantly weaker, result.

A way to prevent the faulty reasoning is to denote the view in one list. In our example we would denote the view as $\nu = ((M, h), (h, \mathrm{IV}_1, y), (M, \mathrm{IV}_1, h))$, where the final h contains no randomness. This can complicate the analysis, depending on the used mode. It can be easier to use this reasoning when there is truncation involved, as that means that there is still some randomness in the query that can be used. For example, the work [CLL19] does do this correctly: they denote the view as a single list and also have a mode using truncation.

4 Results

For all results stated below we consider a tree hashing mode based on a b-bit block cipher and with capacity c denoting the size of the chaining values. These results are also summarized in Table 1.

Theorem 1. *Let $\mathcal{T}$ be a mode that is subtree-free, radical-decodable, message-decodable, leaf-anchored with IV_1-length m and has finalization function $\zeta(y) = \lfloor y \rfloor_n$. Then there exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$ that makes q queries total and r queries to the construction oracle we have*

$$\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) \leqslant \frac{q}{2^m} + \frac{q^2}{2^c} + \frac{q^2 + 2qr}{2^b} + \frac{(\ln(r) + n + 1)q}{2^{b-n}}.$$

The simulator $\mathcal{S}$ makes at most q queries to the random oracle.

The proof is given in Section A.

Theorem 2. *Let $\mathcal{T}$ be a mode that is radical-decodable, message-decodable, leaf-anchored with IV_1-length m and has finalization function $\zeta(y) = \lfloor y \rfloor_n$. Then there exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$ that makes q queries total and r queries to the construction oracle we have*

$$\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) \leqslant \frac{q + r}{2^m} + \frac{q^2 + 2qr}{2^c} + \frac{q^2 + 2qr}{2^b} + (\ln(r) + n + 1)\left(\frac{q}{2^{c-n}} + \frac{q}{2^{b-n}}\right).$$

The simulator $\mathcal{S}$ makes at most q queries to the random oracle.

The proof is given in Section A.6.

Theorem 3. *Let $\mathcal{T}$ be a mode that is radical-decodable, message-decodable, leaf-anchored with IV_1-length m and has finalization function $\zeta(y) = \lceil y \rceil_n$. Then there exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$ that makes q queries total and r queries to the construction oracle we either have*

$$\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) \leqslant \frac{q}{2^m} + \frac{q^2}{2^c} + \frac{q^2}{2^{b-n}},$$

if $b - n \geqslant c$, and

$$\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) \leqslant \frac{q + r}{2^m} + \frac{q^2}{2^c} + \frac{q^2 + 2qr}{2^b} + \frac{(2\ln(r) + 2n + 2)q}{2^{b-n}}$$

otherwise. The simulator $\mathcal{S}$ makes at most q queries to the random oracle.

The proof is given in Section A.7.

Theorem 4. *Let $\mathcal{T}$ be a mode that is subtree-free, radical-decodable, message-decodable, leaf-anchored with IV_1-length m, fully final-anchored and has finalization function $\zeta(y) = y$. Then there exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$ that makes q queries total and r queries to the construction oracle we have*

$$\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) \leqslant \frac{q}{2^m} + \frac{3q^2+q}{2^c} + \frac{3q^2+2qr}{2^b}.$$

The simulator $\mathcal{S}$ makes at most q queries to the random oracle.

The proof is given in Section B.

Theorem 5. *Let $\mathcal{T}$ be a mode that is subtree-free, radical-decodable, message-decodable, leaf-anchored with IV_1-length m, partially final-anchored with P possibilities for IV_2 and has finalization function $\zeta_x(y) = x \oplus y$. Then there exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$ that makes q queries total and r queries to the construction oracle we have*

$$\mathsf{Adv}^{\mathrm{diff}}_{\mathcal{T}[\mathcal{E}],\mathcal{S}}(\mathcal{D}) \leqslant \frac{q}{2^m} + \frac{3q^2+4qr+4r^2+2r}{2^c} + \frac{3q^2+2Pqr}{2^b}.$$

The simulator $\mathcal{S}$ makes at most Pq^2 queries to the random oracle.

The proof is given in Section C.

5 Applications

5.1 Truncated SHA-2

SHA-2 [SHA08] uses a straightforward Merkle-Damgård mode based on a block cipher with the Davies-Meyer feed-forward on top of it. By using Theorem 2 we are able to prove this mode secure, *without requiring any feed-forward*. The mode is illustrated in Figure 8.

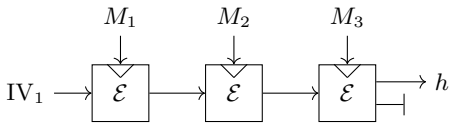

Fig. 8: Illustration of the Merkle-Damgård mode used in SHA-2, but without any feed-forward.

We show that this mode satisfies the required conditions. First of all, it is message-decodable as the message can be retrieved from the tree and it is also

leaf-anchored by definition. For radical decodability we take $\mathcal{S}_{\mathcal{T}}^{\text{rad}} = \mathcal{S}_{\mathcal{T}}^{\text{sub}} \setminus \mathcal{S}_{\mathcal{T}}^{\text{leaf}}$ as the largest possible set and identify radicals by the absence of the IV_1. This means that we can apply Theorem 2 with $m = c = b \in \{256, 512\}$ the internal state size and $n \in \{224, 256, 384, 512\}$ (the latter two only for $b = 512$) the digest length. If n is close to b this gives an insecure bound, as then the mode is vulnerable to a length extension attack.

5.2 BLAKE3

BLAKE3 [OANW20] is a recently introduced tree hash that makes full use of the parallelism that it provides. We will not describe the hashing mode in detail, but we show that with our results we can analyze the security of the mode of operation of BLAKE3. The BLAKE3 paper cites the article by Daemen et al. [DMA18] in the security analysis and show that it satisfies the required conditions. This works for the truncated version, but a full-length output version is also used. For this they informally state that the feed-forward is sufficient for this. Using our Theorem 5 we are able to show that this informal reasoning is not completely correct, as the extendable output mode introduces new security considerations.

We succinctly describe what the final compression call looks like, as that one is relevant for the applicability of the feed-forward and the partial final-anchoring. The data input to the final call is of the for form $\text{CV}\|\text{IV}_2\|t\|b\|d$ with key a message $M \in \{0,1\}^{512}$, where $\text{CV} \in \{0,1\}^{256}$ is the chaining value (or the initial value, if the block consists of one block), $\text{IV}_2 \in \{0,1\}^{128}$ a fixed value used for every compression call, $t \in \{0,1\}^{64}$ a counter for extendable output, $b \in \{0,1\}^{32}$ the number of bytes in the message M and $d \in \{0,1\}^{32}$ some flags. The output of the block cipher is $V_L\|V_H = \mathcal{E}_M(\text{CV}\|\text{IV}_2\|t\|b\|d)$, with $V_L, V_H \in \{0,1\}^{256}$. The final digest is $h = (V_L \oplus V_H) \parallel (V_H \oplus \text{CV})$. This is also illustrated in Figure 9.

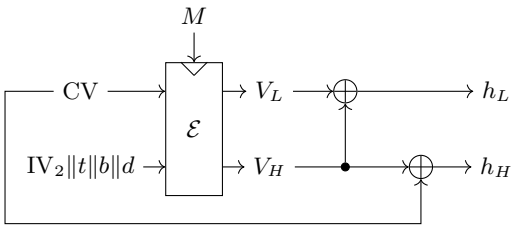

Fig. 9: Illustration of the final compression call of BLAKE3.

5.2.1 Fixed Output In the fixed output mode of BLAKE3 the final digest h is truncated to 256 bits, which corresponds to $V_L \oplus V_H$, where $V_L\|V_H$ is

the output of the final block cipher call. This means that no feed-forward with the previous chaining value is used. Although this finalization is different from truncation, which would be just V_L, this difference is not essential and the result could be easily modified for this finalization. Therefore, we do get an appropriate bound from Theorem 1 with $b = 512$, $m = 256$, $c = 256$ and $n = 256$.

5.2.2 Extendable Output In addition to a fixed output mode, BLAKE3 also introduces an extendable output mode, allowing for an arbitrary number of output bits, similar to the sponge construction [BDPV07]. In contrast to the sponge construction, which uses a sequential output, BLAKE3 uses a counter for its extendable output. It behaves similar to the generic construction where the counter is appended to the message, which would result in the output $H(M\|0) \| H(M\|1) \| H(M\|2) \| \ldots$ for a generic hash function H. In contrast to this generic construction, the counter is placed in the final compression call, making computing successive outputs much more efficient, while still allowing efficient random access in contrast to the sponge. However, as we will see, this feature of allowing efficient random access comes with new security considerations which BLAKE3 does not adhere fully.

For the extendable output mode the full output $h = (V_L \oplus V_H) \| (V_H \oplus \mathrm{CV})$ of the compression function is used. To get an arbitrary number of output bits BLAKE3 uses a counter t that is part of the final compression call. Let h_t denote the output with size b stated above when a counter t is used. Then the full output is equal to $h_0\|h_1\|h_2\|\ldots$.

Our definition of a tree hashing mode does not directly include this extendable output, but it can be achieved by making use of the parameters. Recall that the tree template does not only depend on the length of the message $|M|$, but also on the parameters A which can be chosen freely from a custom defined set $\mathcal{A}$. In this case we choose $\mathcal{A} = \{0, 1, \ldots, \ell - 1\}$, where ℓ is the maximum number of allowed output blocks, to represent the value of the counter t. This means that the output can be computed by computing the hashes of (M, t) for all relevant counters t. Note that definition allows for more freedom than a sequential construction as this t can start at any arbitrary offset. This extra freedom does correspond to the use of BLAKE3, as it indeed can efficiently compute the output starting at any offset.

As with the fixed output, this finalization does not directly correspond to our definition of the feed-forward. But, again, the proof only uses the randomness of the chaining value, which is included, so the bound of Theorem 5 is still applicable for this finalization.

A more significant problem arises when we look at the other new requirement for a secure mode that uses feed-forwarding. The mode should also satisfy partial final-anchoring, which means that there should be a limited number of possibilities for the input of the final compression call other than the chaining value. As stated earlier, for BLAKE3 this consists of $\mathrm{IV}_2\|t\|b\|d$, where our main focus will be t, which is the counter that underlies the extendable output. This t has ℓ possible values, which is maximum number of allowed output blocks. As

BLAKE3 allows for a maximum of 2^{64} output bytes we get $\ell = 2^{64}/64 = 2^{58}$, although the counter can in principle be any 64-bit value. The values b and d both have 2^6 possibilities, hence partial final-anchoring is satisfied with $P = \ell \cdot 2^{12}$, which is typically dominated by ℓ.

This means that we can apply Theorem 5 to this mode with $P = \ell \cdot 2^{12}$, $b = 512$, $m = 256$ and $c = 256$, with $\ell \leqslant 2^{58}$ the maximum number of output blocks in the extendable output mode. Although P is quite large, this still gives the expected security level as $P \cdot 2^c \leqslant 2^b$. There is a downside to the large P, though, as the simulator becomes quite inefficient with its query complexity of Pq^2. This is actually reflected in some non-ideal behavior of BLAKE3 that we describe next.

5.2.3 Computing the Counter Suppose that a query of the form $h_L \| h_H = h = H(M, t)$ is performed, where M is the message and t the block offset in the extendable mode, which corresponds to the counter. Assume that M and h are known to an attacker, but t is not. Ideally, the only way to retrieve t is to try all possible $t' \leqslant \ell$ and check whether $H(M, t')$ equals h. However, in the case of BLAKE3 this t can be retrieved much more efficiently. Recall that digest is defined as $(V_L \oplus V_H) \parallel (V_H \oplus \mathrm{CV})$, with $V_L \| V_H$ the output of the final block cipher call. As M is known to the attacker, it can compute CV. Furthermore, $h_L = V_L \oplus V_H$ and $h_H = V_H \oplus \mathrm{CV}$ are also known, so it can compute $V_H = h_H \oplus \mathrm{CV}$ and $V_L = h_L \oplus V_H$. This means that it can perform the inverse of the final block cipher call as $\mathcal{E}_m^{-1}(V_L \| V_H) = \mathrm{CV} \| \mathrm{IV}_2 \| t \| b \| d$, with m the message input to the final block, and retrieve t this way. This operation costs just one query to $\mathcal{E}$ (and some to compute CV), which is significantly less than the expected ℓ, which can be as high as 2^{58}.

This problem can be illustrated by the following example. Suppose that BLAKE3 is used as the following illustrative MAC. This MAC gets as input a key $K \in \{0,1\}^{128}$ and a message $M \in \{0,1\}^*$. It splits the key as $K = K_1 \| K_2$ with $K_1 \in \{0,1\}^{70}$ and $K_2 \in \{0,1\}^{58}$ and computes the MAC as $H(M \| K_1, t = K_2)$. For an ideal hash function this construction gives a secure MAC, as the offset can essentially be viewed as part of the input. However, when instantiated with BLAKE3 this is not the case. Given $h = H(M \| K_1, t = K_2)$ and M, but not K, an adversary can compute K in roughly 2^{70} queries, instead of the expected 2^{128}. This is done by first guessing an arbitrary $K_1 \in \{0,1\}^{70}$. Then the adversary can compute the offset K_2 from h and $M \| K_1$ as described above. If the guess of K_1 is correct, this computes the value of $K_2 \in \{0,1\}^{58}$ using a single query, performing a key-recovery attack. As there are 2^{70} possible values for K_1 this attack succeeds using roughly 2^{70} queries. Although BLAKE3 supports a dedicated keyed mode that is preferred, the previous example should still be secure. This shows that the counter in BLAKE3 can only contain public information.

5.2.4 Conclusion BLAKE3 makes full use of tree hashing capabilities with an interesting way of generating extendable output by making use of a counter. Although its tree structure is secure, its use of a counter, which makes efficient

random access possible, comes with new security considerations. In particular, from a usage perspective it behaves similar to an extra small efficient message input. However, its security properties do not align with this behavior as the counter can be efficiently computed by knowing the message and the hash output. This is not the case for a normal message input, making BLAKE3 in essence add an extra requirement in that the counter should always be public.

5.3 Tree Sponge

Here we introduce a tree generalization of the sponge construction [BDPV07]. The absorbing phase is generalized to have a tree structure, allowing for parallel compression. Additionally, the squeezing phase is modified to likewise allow for parallel expansion by making use of a counter. The construction requires a minimal number of frame bits: the only ones present are initial values required to prevent inverse queries from succeeding.

First of all we note that all our results also apply to permutations by simply setting $\kappa = 0$. The tree sponge makes use of the flexible conditions present in Theorem 3. The main observation is that subtree-freeness can be dropped without negative consequences when the chaining values and the hash digests originate from different parts of the output of the permutation. This is the same as in the original sponge construction, which has an inner part that outputs chaining values, which are secret, and an outer part that outputs hash digests, which are public.

5.3.1 Description The tree sponge contains three different phases and depends on a fixed parameter $w \in \mathbb{N}_{>0}$ representing the width:

- Absorbing. In this phase the message is split such that every part can be absorbed by a sponge of width w. The final part may be smaller. All different parts are absorbed this way in parallel and each generate their own chaining value.
- Combining. The chaining values generated in the previous phase are combined by using a tree structure. The chaining values are split into two non-empty parts, with the first part the largest possible power of two. The two parts are recursively reduced to a single chaining value and combined using a permutation call.
- Squeezing. The resulting chaining value is fed into multiple final permutation calls appended by $\mathrm{IV}_1 \| t$, with $t = 0, 1, \ldots$ a counter for an arbitrary long output.

An example of this mode is pictured in Figure 10 and an implementation is illustrated in Algorithm 5.

5.3.2 Security We show that this mode satisfies the required conditions. Again, message-decodability and leaf-anchoring are satisfied in a straightforward way. Radical-decodability is more interesting for this mode.

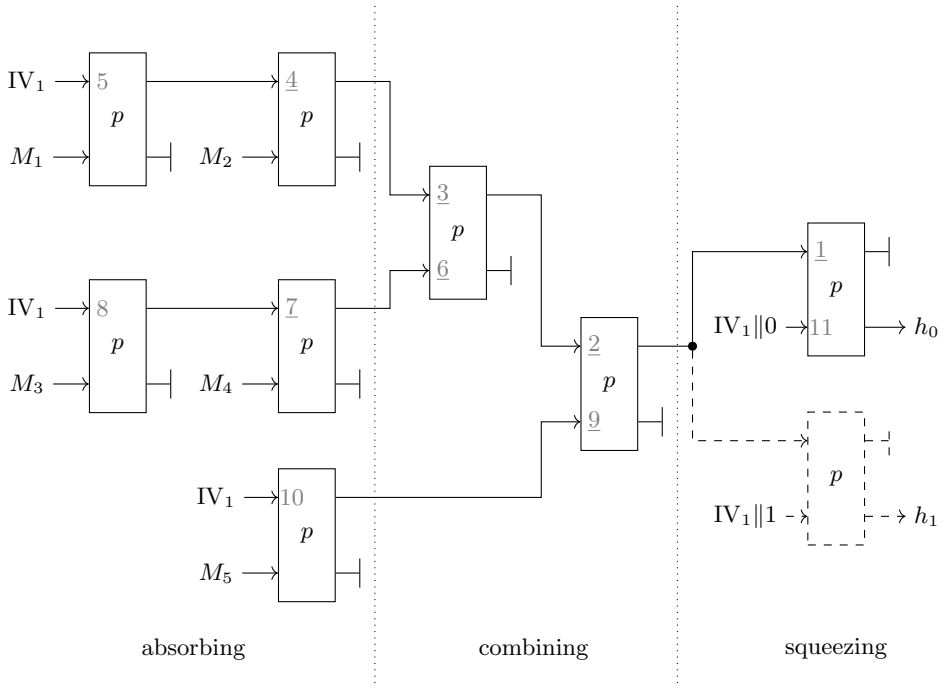

Fig. 10: Example of the minimal permutation-based tree hashing mode with $w = 2$ giving two blocks of output. The order in which the radicals are identified is indicated by the gray numbers 1–11, starting from the final permutation call resulting in h_0. Underlined numbers indicate a radical, while the other numbers indicate a different value: either a leaf or a counter value. The dashed permutation call resulting in h_1 is not part of the tree found by the radical finding algorithm.

We take $\mathcal{S}^{\text{rad}}_{\mathcal{T}} = \mathcal{S}^{\text{sub}}_{\mathcal{T}} \setminus \mathcal{S}^{\text{leaf}}_{\mathcal{T}}$ as the largest possible set and use leaf-anchoring to identify most radicals. We identify leaf nodes by the occurrence of IV_1, which do not have any radical. If we do not find a leaf node, we do not immediately know whether the node has one or two chaining values. However, we only have to find one radical at a time and we know that the first c bits will always point to a chaining value. We continue this process for the topmost chaining value until we hit a leaf node. Then we know that the first w calls after the leaf node are sequential and do not have any other chaining values. All the other nodes do have a chaining value in the bottom halve and we recursively continue the process on all those values. The only exception is the counter in the end, but we can recognize this again by the presence of IV_1. Furthermore, the fact that the final message block may have a width smaller than w does not matter as it is the final

Algorithm 5 Implementation of the tree sponge mode pictured in Figure 10

Interface: TreeSponge(M, t)
  CV $\leftarrow$ combine(M)
  return $\lceil p(\text{CV} \parallel \text{IV}_1 \parallel t) \rceil_{3b/4}$

Interface: combine(M)
  $W \leftarrow w \cdot b/2 + b/4$ ▷ maximum sequential absorption
  if $|M| \leqslant W$ **then**
    return absorb(M)
  end if
  $k \leftarrow \lfloor \log_2(|M|/W) \rfloor$ ▷ largest k such that $W \cdot 2^k \leqslant |M|$
  $M_L \parallel M_R \leftarrow M$ ▷ with $|M_L| = W \cdot 2^k$
  $\text{CV}_L \leftarrow$ combine(M_L)
  $\text{CV}_R \leftarrow$ combine(M_R)
  return $\lfloor p(\text{CV}_L \parallel \text{CV}_R) \rfloor_{b/2}$

Interface: absorb(M)
  $M_0 \parallel M_1 \parallel \cdots \parallel M_{\ell-1} \leftarrow M$ ▷ with $|M_0| = 3b/4$ and $|M_i| = b/2$
  $x \leftarrow \text{IV}_1 \parallel M_0$
  for $i \leftarrow 1$ to ℓ **do**
    $x \leftarrow \lfloor p(x) \rfloor_{b/2} \parallel M_i$
  end for
  return $\lfloor p(x) \rfloor_{b/2}$

block the algorithm finds. An example of this process is pictured in Figure 10 by the gray numbers and an implementation is illustrated in Algorithm 6.

Given a permutation of size b we choose $c = b/2$ as we have a binary tree. This leads to a security level of at most $b/4$, which is inherently the maximum for a permutation-based tree hash. Given this security level we are additionally able to choose $m = b/4$ and $n = 3b/4$ to optimize the efficiency while keeping the same security level.

6 Proof Sketch

The full proof is given in the Supplementary Material, but the main ideas in it are the following:

- The simulator uses radical-decodability to reconstruct the tree corresponding to a (potential) message. Message-decodability is used to reconstruct the message in order to be consistent with the random oracle. Otherwise randomly generated values are used.
- Various bad events are defined to make sure the following properties hold for good views:
  - The simulator is consistent with the random oracle.
  - The simulator is consistent as a permutation, i.e. $\mathcal{S}_k(x_1) = \mathcal{S}_k(x_2)$ implies $x_1 = x_2$ and similar for the inverse.

Algorithm 6 Implementation of radical() for the tree sponge mode pictured in Figure 10

Interface: radical(S)
  $(\alpha, \text{depth}) \leftarrow \text{radical}'(S, \bot)$
  return α

Interface: radical$'(S, \alpha')$
  if $\text{lookup}(S, \alpha') = \bot$ **then**
    return $(\alpha', \bot)$
  end if
  $\mathbf{i} : (k, x, \alpha) \leftarrow \text{lookup}(S, \alpha')$
  if $\lfloor x \rfloor_{b/4} = \text{IV}_1$ **then**
    return $(\bot, 1)$ ▷ end of path by leaf-anchoring
  end if
  $\alpha' \leftarrow \mathbf{i}_0^x$
  $(\alpha', \text{depth}) \leftarrow \text{radical}'(S, \alpha')$ ▷ scan the top half for radicals
  if $\alpha' \neq \bot$ **then**
    return $(\alpha', \bot)$
  end if
  if $\text{depth} \neq \bot \wedge \text{depth} < w$ **then**
    return $(\bot, \text{depth} + 1)$ ▷ absorb phase
  else
    $x_1 \parallel x_2 \leftarrow x$ ▷ with $|x_1| = |x_2| = b/2$
    if $\lfloor x_2 \rfloor_{b/4} = \text{IV}_1$ **then**
      return $(\bot, \bot)$ ▷ squeeze phase
    else
      $\alpha' \leftarrow \mathbf{i}_{b/2}^x$ ▷ combine phase
      return $\text{radical}'(S, \alpha')$ ▷ scan the bottom half for radicals
    end if
  end if

The main goal of the bad events is to prevent various collisions and to prevent inversions of the final compression call. This last property was not handled appropriate in [DMA18] and is solved by the various finalization functions:

- Truncation/Chopping: by throwing away part of the input the inverse calls can only succeed by guessing the discarded bits, which is negligible for sufficient truncation.
- Enveloped: as in the final compression call the message related can only be part of the key input, no information can be gained from an inverse call. The output contains the data input, which is constant. Notably, the inverse simulator has to modified to account for the possibility of making an unorthodox query by computing the hash normally, except for the final call for which the inverse is used.
- Feed-forward: this case is similar to the enveloped case. A key difference is that the final inverse call does not necessarily correspond to a single message, making the inverse simulator having to loop over all possibilities.

These bad events occur with negligible probability as all the values are randomly generated. Most of the difficulty comes from identifying the faulty queries. As the wrong simplification discussed in Section 3 cannot be applied, it becomes more tricky to identify the faulty queries, which become more varied. This is especially true for the feed-forward mode as the inverse queries can correspond to multiple messages.

Acknowledgments. Aldo Gunsing is supported by the Netherlands Organisation for Scientific Research (NWO) under TOP grant TOP1.18.002 SCALAR.

References

ABR21. Elena Andreeva, Rishiraj Bhattacharyya, and Arnab Roy. Compactness of Hashing Modes and Efficiency Beyond Merkle Tree. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 92–123. Springer, 2021.

BDPV07. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. Ecrypt Hash Workshop, 2007. http://sponge.noekeon.org/SpongeFunctions.pdf.

BDPV14. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sufficient conditions for sound tree and sequential hashing modes. *Int. J. Inf. Sec.*, 13(4):335–353, 2014.

Bel06. Mihir Bellare. New Proofs for NMAC and HMAC: Security without collision-resistance. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, 2006.

BN18. Srimanta Bhattacharya and Mridul Nandi. Full Indifferentiable Security of the Xor of Two or More Random Permutations Using the χ^2 Method. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 387–412. Springer, 2018.

BR06. Mihir Bellare and Thomas Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2006.

Bra90. Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.

CDMP05. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

CLL19. Wonseok Choi, Byeonghak Lee, and Jooyoung Lee. Indifferentiability of Truncated Random Permutations. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 175–195. Springer, 2019.

CLNY06. Donghoon Chang, Sangjin Lee, Mridul Nandi, and Moti Yung. Indifferentiable Security Analysis of Popular Hash Functions with Prefix-Free Padding. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 2006.

CN08. Donghoon Chang and Mridul Nandi. Improved Indifferentiability Security Analysis of chopMD Hash Function. In Kaisa Nyberg, editor, *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers*, volume 5086 of *Lecture Notes in Computer Science*, pages 429–443. Springer, 2008.

CS14. Shan Chen and John P. Steinberger. Tight Security Bounds for Key-Alternating Ciphers. In Nguyen and Oswald [NO14], pages 327–350.

Dam89. Ivan Damgård. A Design Principle for Hash Functions. In Brassard [Bra90], pages 416–427.

DMA18. Joan Daemen, Bart Mennink, and Gilles Van Assche. Sound Hashing Modes of Arbitrary Functions, Permutations, and Block Ciphers. *IACR Trans. Symmetric Cryptol.*, 2018(4):197–228, 2018.

DRRS09. Yevgeniy Dodis, Leonid Reyzin, Ronald L. Rivest, and Emily Shen. Indifferentiability of Permutation-Based Compression Functions and Tree-Based Modes of Operation, with Applications to MD6. In Orr Dunkelman, editor, *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, volume 5665 of *Lecture Notes in Computer Science*, pages 104–121. Springer, 2009.

GBK16. Praveen Gauravaram, Nasour Bagheri, and Lars R. Knudsen. Building indifferentiable compression functions from the PGV compression functions. *Des. Codes Cryptogr.*, 78(2):547–581, 2016.

GDM20. Aldo Gunsing, Joan Daemen, and Bart Mennink. Errata to Sound Hashing Modes of Arbitrary Functions, Permutations, and Block Ciphers. *IACR Trans. Symmetric Cryptol.*, 2020(3):362–366, 2020.

GLC08. Zheng Gong, Xuejia Lai, and Kefei Chen. A synthetic indifferentiability analysis of some block-cipher-based hash functions. *Des. Codes Cryptogr.*, 48(3):293–305, 2008.

KBC97. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. *RFC*, 2104:1–11, 1997.

Lee17. Jooyoung Lee. Indifferentiability of the Sum of Random Permutations Toward Optimal Security. *IEEE Trans. Inf. Theory*, 63(6):4050–4054, 2017.

LGD+09. Yiyuan Luo, Zheng Gong, Ming Duan, Bo Zhu, and Xuejia Lai. Revisiting the Indifferentiability of PGV Hash Functions. *IACR Cryptol. ePrint Arch.*, 2009:265, 2009.

LLG11. Yiyuan Luo, Xuejia Lai, and Zheng Gong. Indifferentiability of Domain Extension Modes for Hash Functions. In Liqun Chen, Moti Yung, and Liehuang Zhu, editors, *Trusted Systems - Third International Conference, INTRUST 2011, Beijing, China, November 27-29, 2011, Revised Selected Papers*, volume 7222 of *Lecture Notes in Computer Science*, pages 138–155. Springer, 2011.

LMN16. Atul Luykx, Bart Mennink, and Samuel Neves. Security Analysis of BLAKE2's Modes of Operation. *IACR Trans. Symmetric Cryptol.*, 2016(1):158–176, 2016.

Men13. Bart Mennink. Indifferentiability of Double Length Compression Functions. In Martijn Stam, editor, *Cryptography and Coding - 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings*, volume 8308 of *Lecture Notes in Computer Science*, pages 232–251. Springer, 2013.

Mer89. Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [Bra90], pages 428–446.

MP15. Bart Mennink and Bart Preneel. On the XOR of Multiple Random Permutations. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security - 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers*, volume 9092 of *Lecture Notes in Computer Science*, pages 619–634. Springer, 2015.

MPN10. Avradip Mandal, Jacques Patarin, and Valérie Nachef. Indifferentiability beyond the Birthday Bound for the Xor of Two Public Random Permutations. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings*, volume 6498 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2010.

MPS12. Avradip Mandal, Jacques Patarin, and Yannick Seurin. On the Public Indifferentiability and Correlation Intractability of the 6-Round Feistel Construction. In Ronald Cramer, editor, *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, volume 7194 of *Lecture Notes in Computer Science*, pages 285–302. Springer, 2012.

MRH04. Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In Moni Naor, editor, *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004.

NO14. Phong Q. Nguyen and Elisabeth Oswald, editors. *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*. Springer, 2014.

OANW20. Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. BLAKE3. https://blake3.io, 2020.

Pat08. Jacques Patarin. The "Coefficients H" Technique. In Roberto Maria Avanzi, Liam Keliher, and Francesco Sica, editors, *Selected Areas in Cryptography, 15th International Workshop, SAC 2008, Sackville, New Brunswick, Canada, August 14-15, Revised Selected Papers*, volume 5381 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 2008.

PGV93. Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In Douglas R. Stinson, editor, *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1993.

SHA08. National Institute of Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-3, October 2008.

Supplementary Material

A Security of Truncation (Theorem 1)

A.1 Distinguisher

We will consider a distinguisher $\mathcal{D}'$ based on $\mathcal{D}$. The distinguisher $\mathcal{D}'$ behaves similar to $\mathcal{D}$, but it will verify all its queries to the random oracle at the end by querying the primitive on the necessary queries. This distinguisher has the same advantage as $\mathcal{D}$ as it always outputs the same decision and the simulator is not influenced as the verification queries happen at the end. Moreover, it does increase the total query complexity as these verification queries have to be included, but these extra queries are limited to a maximum of Lr, where L is the maximum length of a message (in permutation calls).

A.2 Simulator

The simulator consists of two parts: the main simulator defined in Algorithm 8 and the helper functions radicalExtend[$\mathcal{L}$] and radicalValue[$\mathcal{L}$] defined in Algorithm 7, based on [DMA18]. The helper function radicalExtend[$\mathcal{L}$](S) extends all possible radicals in the given tree S with values from the list $\mathcal{L}$, while the function radicalValue[$\mathcal{L}$](S) returns the value of a radical that cannot be extended, if it exists, and $\perp$ otherwise. As fixed points are allowed in general, the algorithm radicalExtend[$\mathcal{L}$](S) could get in an infinite loop. To mitigate this problem, we can set a maximum message length L (in permutation calls) and let the algorithm return when its recursion exceeds this L. As this is about the computational complexity and not the query complexity this L does not show up in our bound.

The simulator calls radicalExtend[$\mathcal{L}^{\text{rel}}$](S) for some set $\mathcal{L}^{\text{rel}}$ we define later to determine whether the current query completes a tree in which case it should query the random oracle and answer accordingly. Otherwise it answers randomly. On inverse queries it always gives a random output.

The simulator does not consider all queries when looking to complete a tree, but only queries from $\mathcal{L}^{\text{rel}}$ which we recursively define as

$$\mathcal{L}^{\text{rel}} = \{\, (k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \mid \text{radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x_i) \notin \mathcal{S}_{\mathcal{T}} \cup \mathcal{S}^{\text{rad}}_{\mathcal{T}} \,\},$$

where $\mathcal{L}^{\text{fwd}}$ are all forward queries made to the simulator and where $\mathcal{L}^{\text{rel}}_{i-1}$ denotes the list $\mathcal{L}^{\text{rel}}$ up to index $i-1$. These should be the only queries that are part of a complete tree as non-final nodes, which is the reason that the simulator only considers these queries. The bad events will make sure that this will indeed be the case. Note that the simulator can efficiently derive $\mathcal{L}^{\text{rel}}$.

A.3 Views

We denote by $\nu = \{\, (X_1, Y_1), \ldots, (X_{q+r}, Y_{q+r}) \,\}$ the view seen by distinguisher $\mathcal{D}'$ on interaction, where X_i is the input to either the primitive or construction

Algorithm 7

Interface: radicalExtend$[\mathcal{L}](S)$
  if radical$(S) = \perp$ **then**
    return S
  end if
  for all $(k, x, y) \in \mathcal{L}$ **do**
    if $\lfloor y \rfloor_c = S[\text{radical}(S)]$ **then**
      $\mathbf{i} \leftarrow |S|$ ▷ fresh node identifier
      $S' \leftarrow S \cup \{\mathbf{i} : (k, x, \text{radical}(S))\}$ ▷ fill in the radical
      return radicalExtend$[\mathcal{L}](S')$ ▷ recursively fill in the rest
    end if
  end for
  return S ▷ no extension found

Interface: radicalValue$[\mathcal{L}](k, x)$
  $S \leftarrow$ radicalExtend$[\mathcal{L}](\{\mathbf{0} : (k, x, \perp)\})$
  return $S[\text{radical}(S)]$

oracle and Y_i the output. We split this view into $\mathcal{L}^{\text{fwd}}$ for the forwards primitive oracle, for which $X_i = (k_i, x_i)$ and $Y_i = y_i$, into $\mathcal{L}^{\text{inv}}$ for the inverse primitive oracle, for which $X_i = (k_i, y_i)$ and $Y_i = x_i$ and into $\mathcal{M}$ for the construction oracle, for which $X_i = (M_i, Z_i)$ and $Y_i = h_i$. We combine these lists as this preserves the order in which the distinguisher made the queries. With $\mathcal{L}_k$ we denote the sublist of the list $\mathcal{L}$ containing only queries with key k and with this key omitted from the list. With dom$\mathcal{L}_k$ we denote the inputs to the block cipher in the list $\mathcal{L}_k$ and with rng$\mathcal{L}_k$ likewise the outputs from the block cipher. We define several sublists which all require different analysis. First of all we define $\mathcal{L}^{\text{rad}}$ as

$$\mathcal{L}^{\text{rad}} = \{\, (k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \mid \text{radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x_i) \in \mathcal{S}^{\text{rad}}_{\mathcal{T}} \,\}\,.$$

These are radical queries and introduce the radical radicalValue$[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x_i)$, which should not be filled. Additionally we define $\mathcal{L}^{\text{complete}}$ as

$$\mathcal{L}^{\text{complete}} = \{\, (k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \mid \text{radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x_i) \in \mathcal{S}_{\mathcal{T}} \,\}\,.$$

These are queries that complete a tree and in the ideal world the simulator will call the random oracle to determine the result. As a shorthand, we use the following definition to get the corresponding message for such queries.

Definition 11. *Let* $1 \leqslant i \leqslant q + r$, $k \in \{0,1\}^\kappa$ *and* $x \in \{0,1\}^b$. *We define* extractFrom$[\mathcal{L}^{\text{rel}}_{i-1}](k, x)$ *as*

$$\begin{cases} \text{extract}(\text{radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k, x)) & \text{if radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k, x) \in \mathcal{S}_{\mathcal{T}}, \\ \perp & \text{otherwise.} \end{cases}$$

Now we define the set $\mathcal{L}^{\text{fwd,known}}$ as the queries in $\mathcal{L}^{\text{complete}}$ whose hash result is already known to the distinguisher. In the case of truncation, this means that

Algorithm 8

Interface: $\mathcal{S} : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b\ ,\ \ (k,x) \mapsto y$
  if $x \notin \mathrm{dom}\mathcal{L}_k$ **then**
    $S \leftarrow \mathrm{radicalExtend}[\mathcal{L}^{\mathrm{rel}}](k,x)$ ▷ radical-extend tree from single node
    if $S \in \mathcal{S}_{\mathcal{T}}$ **then** ▷ query completes tree
      $(M,Z) \leftarrow \mathrm{extract}(S)$ ▷ extract message and tree template
      $h \leftarrow \mathcal{RO}(M,Z)$
      $y \xleftarrow{\$} \zeta^{-1}(h)$
    else ▷ query does not complete tree
      $y \xleftarrow{\$} \{0,1\}^b$
    end if
    $\mathcal{L}_k(x) \leftarrow y$
  end if
  return $\mathcal{L}_k(x)$

Interface: $\mathcal{S}^{-1} : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b\ ,\ \ (k,y) \mapsto x$
  if $y \notin \mathrm{rng}\mathcal{L}_k$ **then**
    $x \xleftarrow{\$} \{0,1\}^b$
    $\mathcal{L}_k^{-1}(y) \leftarrow x$
  end if
  return $\mathcal{L}_k^{-1}(y)$

the distinguisher already knows the first n bits of the output, while the last $b-n$ bits are randomly generated.

$$\mathcal{L}^{\mathrm{fwd,known}} = \{\, (k_i, x_i, y_i) \in \mathcal{L}^{\mathrm{complete}} \mid \mathrm{extractFrom}[\mathcal{L}^{\mathrm{rel}}_{i-1}](k_i, x_i) \in \mathrm{dom}\mathcal{M}_{i-1} \,\}$$

This is the essence of the error in [DMA18]: there it was assumed that all bits are randomly generated.

The complement of this are queries which are completely unknown to the distinguisher. Even if a query completes a tree and queries the random oracle, if the distinguisher does not know the hash output, all bits are randomly generated from its point of view. We define $\mathcal{L}^{\mathrm{fwd,random}}$ as exactly this set.

$$\mathcal{L}^{\mathrm{fwd,random}} = \mathcal{L}^{\mathrm{fwd}} \setminus \mathcal{L}^{\mathrm{fwd,known}}.$$

Finally, we note that $\mathcal{L}^{\mathrm{rel}} = \mathcal{L}^{\mathrm{fwd}} \setminus (\mathcal{L}^{\mathrm{rad}} \cup \mathcal{L}^{\mathrm{complete}}) \subseteq \mathcal{L}^{\mathrm{fwd,random}}$, which means that all queries that the simulator considers in building the tree are completely randomly generated.

We also split $\mathcal{M}$ into two smaller sets, similar to $\mathcal{L}^{\mathrm{fwd,known}}$.

$$\mathcal{M}^{\mathrm{known}} = \{\, (M_i, Z_i, h_i) \in \mathcal{M} \mid \exists (k_j, x_j, y_j) \in \mathcal{L}^{\mathrm{complete}}_{i-1} : \mathrm{extractFrom}[\mathcal{L}^{\mathrm{rel}}_{j-1}](k_j, x_j) = (M_i, Z_i) \,\},$$
$$\mathcal{M}^{\mathrm{random}} = \mathcal{M} \setminus \mathcal{M}^{\mathrm{known}}.$$

The output of queries in $\mathcal{M}^{\mathrm{known}}$ are already completely known to the distinguisher and essentially contain no information, while queries in $\mathcal{M}^{\mathrm{random}}$ are completely randomly generated.

An attainable view ν is called *bad* if:

(i) There exist $(k_i, x_i, y_i), (k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ with $i < j$ such that $\lfloor y_i \rfloor_c = \lfloor y_j \rfloor_c$.
(ii) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{rad}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ with $i < j$ such that $\lfloor y_j \rfloor_c = \text{radicalValue}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x_i)$.
(iii) There exists $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv}}$ such that $\lfloor x_i \rfloor_m = \text{IV}_1$.
(iv) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ such that $\lfloor x_i \rfloor_c = \lfloor y_j \rfloor_c$
(v) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{rel}}$ such that $\lfloor y_i \rfloor_m = \text{IV}_1$.
(vi) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd}}$ with $i < j$ and $k_i = k_j$ such that $y_i = y_j$.
(vii) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{inv}}$ with $i < j$ and $k_i = k_j$ such that $x_i = x_j$.

Bad events (i)–(iv) ensure the consistency of the simulator with the random oracle, bad event (v) allows for the relaxation in the definitions of radical-decodability and subtree-freeness, while bad events (vi) and (vii) ensure that the simulator is consistent as a permutation.

An important property of the simulator is that it should be consistent with the random oracle on good views. To show this, we first define what the verification queries are.

Definition 12. *For every query $(M, Z, h) \in \mathcal{M}$ we define the $\iota^{\text{final}}(M, Z)$ as the index on which the distinguisher $\mathcal{D}'$ made the final verification query for message (M, Z). By our definition of $\mathcal{D}'$, we know that this query exists.*

Now we show that the simulator is consistent with the random oracle.

Lemma 3. *Let ν be an attainable view such that none of the bad events (i)–(v) happen. Then for every query $(M, Z, h) \in \mathcal{M}$ we have that $(k_i, x_i, y_i) \in \mathcal{L}^{\text{complete}}$ and $\zeta(y_i) = h$, where $i = \iota^{\text{final}}(M, Z)$.*

Proof. We show that the simulator agrees with the random oracle. Let $(M, Z, h) \in \mathcal{M}$. First we show that all non-final verification queries belong to $\mathcal{L}^{\text{rel}}$. In particular, we show that they cannot be in either $\mathcal{L}^{\text{complete}}$, $\mathcal{L}^{\text{rad}}$ or $\mathcal{L}^{\text{inv}}$. First of all, they cannot be in $\mathcal{L}^{\text{complete}}$, as that would contradict subtree-freeness in combination with radical-decodability. Second, we show that any subtree cannot contain queries in $\mathcal{L}^{\text{rad}}$ or $\mathcal{L}^{\text{inv}}$ by using induction based on leaf-anchoring, which states that any subtree will be a leaf node or a non-leaf node with a smaller subtree as a child.

Leaf nodes cannot be in $\mathcal{L}^{\text{rad}}$ as $\mathcal{S}^{\text{leaf}}_{\mathcal{T}} \cap \mathcal{S}^{\text{rad}}_{\mathcal{T}} = \varnothing$, and they cannot be in $\mathcal{L}^{\text{inv}}$ either by bad event (iii).

Non-leaf nodes cannot be in $\mathcal{L}^{\text{rad}}$ as the radical has to be filled in. As this refers to a smaller subtree, it has to consist of queries in $\mathcal{L}^{\text{rel}}$. This cannot be done earlier by the definition of radical-decodability and cannot be done later by bad event (ii). Non-leaf nodes cannot be in $\mathcal{L}^{\text{inv}}$ by bad event (iv).

This means that we can only consider the queries in $\mathcal{L}^{\text{rel}}$, which is what the simulator does. By bad event (i) the tree is also uniquely defined, which means that, in combination with radical-decodability and message-decodability,

the simulator also finds the right message if the final verification query happened last. If the final query did not happen last it had to be in $\mathcal{S}_{\mathcal{T}}^{\text{rad}}$ as $\mathcal{S}_{\mathcal{T}}^{\text{final}} \subseteq \mathcal{S}_{\mathcal{T}}^{\text{rad}}$, but that would contradict bad event (ii) in a later query. □

Corollary 1. *For any good view we have* $|\mathcal{M}^{\text{random}}| = |\mathcal{L}^{\text{fwd,known}}|$.

Proof. By Lemma 3 every query in $\mathcal{M}$ is correctly verified. For queries in $\mathcal{M}^{\text{known}}$ this happened in a previous query which cannot be in $\mathcal{L}^{\text{fwd,known}}$. For queries in $\mathcal{M}^{\text{random}}$ this happens later, hence that query has to be in $\mathcal{L}^{\text{fwd,known}}$. This means that $|\mathcal{M}^{\text{random}}| = |\mathcal{L}^{\text{fwd,known}}|$ as this mapping is bijective. □

A.4 Analysis of Bad Views

First we analyze bad events (i)–(v). As all relevant values are uniformly randomly generated from $\{0,1\}^c$ (for (i), (ii) and (iv)) or $\{0,1\}^m$ (for (iii) and (v)), we get the following upper bounds:

(i) $|\mathcal{L}^{\text{rel}}|^2/2^c$.
(ii) $|\mathcal{L}^{\text{rad}}| \cdot |\mathcal{L}^{\text{rel}}|/2^c$.
(iii) $|\mathcal{L}^{\text{inv}}|/2^m$.
(iv) $|\mathcal{L}^{\text{inv}}| \cdot |\mathcal{L}^{\text{rel}}|/2^c$.
(v) $|\mathcal{L}^{\text{rel}}|/2^m$.

Combining these upper bounds, we get a total upper bound of

$$\frac{|\mathcal{L}^{\text{inv}}| + |\mathcal{L}^{\text{rel}}|}{2^m} + \frac{(|\mathcal{L}^{\text{rel}}| + |\mathcal{L}^{\text{rad}}| + |\mathcal{L}^{\text{inv}}|) \cdot |\mathcal{L}^{\text{rel}}|}{2^c} \leqslant \frac{q}{2^m} + \frac{q^2}{2^c}.$$

The analysis of bad events (vi) and (vii) is more elaborate.

(vi) In order to make the analysis easier, we define the following helper event:
1. There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ such that $y_i = y_j$.

For this helper event to occur between $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ we need to have both $\lfloor y_i \rfloor_n = \lfloor y_j \rfloor_n$ and $\lceil y_i \rceil_n = \lceil y_j \rceil_n$. As $\lfloor y_j \rfloor_n = h_k$ for some $(M_k, Z_k, h_k) \in \mathcal{M}_{j-1}^{\text{random}}$ and $\lceil y_j \rceil_n$ is randomly generated, this gives a probability for a single suitable query with $h_k = \lfloor y_i \rfloor_n$ of $1/2^{b-n}$. Let F_h denote the number of queries $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ with $h_k = h$. As every query in $\mathcal{L}^{\text{fwd,known}}$ corresponds to a unique query in $\mathcal{M}^{\text{random}}$ there are at most $F_{\lfloor y_i \rfloor_n}$ suitable queries $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ for any $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$, giving a maximum of $|\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}| \cdot \max_h F_h$ possible pairs, which in turns gives an upper bound of

$$\sum_x \frac{|\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}|}{2^{b-n}} \cdot x \cdot \mathbb{P}\left[\max_h F_h = x\right] = \frac{q}{2^{b-n}} \cdot \mathbb{E}\left[\max_h F_h\right].$$

By Lemma 2 this leads to the upper bound

$$\frac{q}{2^{b-n}} \cdot \left(\frac{2r}{2^n} + \ln(r) + n + 1\right) \leqslant \frac{2qr}{2^b} + \frac{(\ln(r) + n + 1)q}{2^{b-n}}.$$

Now we derive the real bad event under the assumption that the helper event does not occur. Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd}}$ with $i < j$. By our helper event we can assume that $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,random}}$, hence it is randomly generated. This gives a bound of

$$\frac{|\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}| \cdot |\mathcal{L}^{\text{fwd,random}}|}{2^b}.$$

(vii) As $(k_j, x_j, y_j) \in \mathcal{L}^{\text{inv}}$ is randomly generated, the probability is bounded by

$$\frac{|\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}| \cdot |\mathcal{L}^{\text{inv}}|}{2^b}.$$

The probability of bad event (vi) and (vii) combined is bounded by

$$\frac{|\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}| \cdot (|\mathcal{L}^{\text{fwd,random}}| + |\mathcal{L}^{\text{inv}}|)}{2^b} \leqslant \frac{q^2}{2^b},$$

with the helper event having an upper bound of

$$\frac{2qr}{2^b} + \frac{(\ln(r) + n + 1)q}{2^{b-n}}.$$

Combining all these gives

$$\mathbb{P}\left[\mathcal{D}'^{\mathcal{O}_2} \in \mathcal{V}_{\text{bad}}\right] \leqslant \frac{q}{2^m} + \frac{q^2}{2^c} + \frac{q^2 + 2qr}{2^b} + \frac{(\ln(r) + n + 1)q}{2^{b-n}}.$$

A.5 Analysis of Good Views

A.5.1 Real World In the real world the randomness is generated by the primitive oracle. For every construction query, the necessary primitive queries are implicitly made. First of all, we want to make these implicit queries explicit. As our modified distinguisher $\mathcal{D}'$ verifies all the construction queries, this only may move some queries forward in the list. As the queries were already made implicitly, this does not influence the probability.

By Lemma 3 this means that every query $(M_i, Z_i, h_i) \in \mathcal{M}$ can be derived correctly from ν_{i-1}, hence these queries have a probability 1 of occurring. As we are working with a block cipher, every query $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ has a probability of $1/(2^b - q_{k,i})$ of occurring, where $q_{k,i}$ is the number of queries made with key k before query i. Furthermore, by bad events (vi) and (vii), and the fact that all construction queries are verified, there is no (implicit) permutation inconsistency and no outputs have probability 0. As all these probabilities are independent the final probability becomes

$$\begin{aligned}
\mathbb{P}\left[\mathcal{D}'^{\mathcal{O}_1} = \nu\right] &= \prod_{k \in \{0,1\}^\kappa} \prod_{i=0}^{q_k - 1} \frac{1}{2^b - i} \\
&\geqslant \prod_{k \in \{0,1\}^\kappa} \frac{1}{2^{bq_k}} \\
&= \frac{1}{2^{bq}},
\end{aligned}$$

where q_k is the total number of queries made with key k.

A.5.2 Ideal World In the ideal world the randomness is generated by both the simulator and the random oracle. Any query in $\mathcal{L}^{\text{fwd,known}}$ has a probability of $1/2^{b-n}$ of getting a specific value, as the first n bits are determined from $\mathcal{M}_{i-1}$. Correspondingly, any query in $\mathcal{M}^{\text{random}}$ has a probability of $1/2^n$, while queries in $\mathcal{M} \setminus \mathcal{M}^{\text{random}}$ have a probability of 1 as they are correctly determined from ν_{i-1} by Lemma 3. As $|\mathcal{M}^{\text{random}}| = |\mathcal{L}^{\text{fwd,known}}|$ we can alternatively count over queries in $\mathcal{L}^{\text{fwd,known}}$ with a probability of $1/2^b$.

Furthermore, any other query has a probability of $1/2^b$ of getting a specific value, as all bits are randomly generated. This means that we get a final upper bound of

$$\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} = \nu\right] \leqslant \frac{1}{2^{bq}}.$$

A.5.3 Ratio This gives us the following ratio:

$$\frac{\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_1} = \nu\right]}{\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} = \nu\right]} \geqslant \frac{2^{bq}}{2^{bq}} = 1.$$

A.6 Removing Subtree-Freenees

If we drop the subtree-freeness requirement, we have to change the definition of $\mathcal{L}^{\text{rel}}$ to also include $\mathcal{L}^{\text{complete}}$ in order for Lemma 3 to still hold. This influences the analysis of bad events (i), (ii), (iv) and (v), but only when the last query is in $\mathcal{L}^{\text{fwd,known}}$, as only those are not completely randomly generated. This means that we get the original bound, but with extra terms for the following bad events:

(i') There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{rel}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ such that $\lfloor y_i \rfloor_c = \lfloor y_j \rfloor_c$.
(ii') There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{rad}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ such that $\lfloor y_j \rfloor_c = \text{radicalValue}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x_i)$.
(iv') There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ such that $\lfloor x_i \rfloor_c = \lfloor y_j \rfloor_c$.
(v') There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,known}}$ such that $\lfloor y_i \rfloor_m = \text{IV}_1$.

We get the following upper bounds:

(i') Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{rel}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ and $\lfloor y_i \rfloor_n = \lfloor y_j \rfloor_n$. As $\lceil y_j \rceil_n$ is randomly generated, we get a probability of $1/2^{c-n}$ for a single such query. Similar to before, there are at most $|\mathcal{L}^{\text{rel}}| \cdot \max_h F_h$ such possible pairs, where F_h denotes the number of queries $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ with $h_k = h$. This leads to an upper bound of

$$\frac{|\mathcal{L}^{\text{rel}}| \cdot \mathbb{E}\left[\max_h F_h\right]}{2^{c-n}}.$$

(ii') This event is very similar, but now $\lfloor y_j \rfloor_c$ has to hit a radical value, giving an upper bound of

$$\frac{|\mathcal{L}^{\text{rad}}| \cdot \mathbb{E}\left[\max_h F_h\right]}{2^{c-n}}.$$

(iv') Again, by the same reasoning we get an upper bound of

$$\frac{|\mathcal{L}^{\text{inv}}| \cdot \mathbb{E}\left[\max_h F_h\right]}{2^{c-n}}.$$

(v') In this case we actually have to split based on the values of m and n. If $m \leqslant n$ we simply have to bound the probability of $\lfloor h_j \rfloor_m = \text{IV}_1$ occurring for any $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$, which is $r/2^m$. If $m > n$ we get a probability of $1/2^{m-n}$ for any query $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,known}}$ with $\lfloor y_i \rfloor_n = \lfloor \text{IV}_1 \rfloor_n$. This leads to the upper bound

$$\frac{\mathbb{E}\left[F_{\lfloor \text{IV}_1 \rfloor_n}\right]}{2^{m-n}} \leqslant \frac{r/2^n}{2^{m-n}} = \frac{r}{2^m},$$

which is the same as the previous case.

All in all, using Lemma 2 these values combined give an extra term of

$$\begin{aligned}
&\frac{r}{2^m} + (|\mathcal{L}^{\text{rel}}| + |\mathcal{L}^{\text{rad}}| + |\mathcal{L}^{\text{inv}}|) \cdot \frac{\mathbb{E}\left[\max_h F_h\right]}{2^{c-n}} \\
&\leqslant \frac{r}{2^m} + \frac{q}{2^{c-n}} \cdot \left(\frac{2r}{2^n} + \ln(r) + n + 1\right) \\
&= \frac{r}{2^m} + \frac{2qr}{2^c} + \frac{(\ln(r) + n + 1)q}{2^{c-n}}.
\end{aligned}$$

As these are extra terms, we get a total bound of

$$\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} \in \mathcal{V}_{\text{bad}}\right] \leqslant \frac{q+r}{2^m} + \frac{q^2+2qr}{2^c} + \frac{q^2+2qr}{2^b} + (\ln(r)+n+1)\left(\frac{q}{2^{c-n}} + \frac{q}{2^{b-n}}\right).$$

A.7 Chopping

Similar to Section A.6 we only have to modify the analysis of the bad views. We split the analysis based on the values of $b-n$ and c.

$b - n \geqslant c$. In this case all relevant bits in the bad events (i)–(v) are generated randomly, which gives the same upper bound as in Section A.4 of

$$\frac{q}{2^m} + \frac{q^2}{2^c}.$$

Furthermore, for bad events (vi) and (vii) we can always consider the first $b-n$ bits to be considered random, giving the upper bound of

$$\frac{q^2}{2^{b-n}}.$$

Combining these, we get the upper bound

$$\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} \in \mathcal{V}_{\text{bad}}\right] \leqslant \frac{q}{2^m} + \frac{q^2}{2^c} + \frac{q^2}{2^{b-n}}.$$

$b - n < c$. The analysis for bad events (i)–(v) is very similar to the one in Section A.6, but the number of fixed and randomly generated bits changes. In this case the first $b - n$ bits are randomly generated, while the bits from $b - n$ to c are fixed. This essentially means that we do not look for the number of queries $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ with $h_k = h$ for a specific h (previously denoted by F_h), but at the number of queries $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ with just $\lfloor h_k \rfloor_{c-(b-n)} = \lfloor h \rfloor_{c-(b-n)}$ for a specific h, denoted by $F_h^{c-(b-n)}$. This leads to the extra term

$$\begin{aligned}
&\frac{r}{2^m} + \frac{|\mathcal{L}^{\text{rel}}| + |\mathcal{L}^{\text{rad}}| + |\mathcal{L}^{\text{inv}}|}{2^{b-n}} \cdot \mathbb{E}\left[\max_h F_h^{c-(b-n)}\right] \\
&\leqslant \frac{r}{2^m} + \frac{q}{2^{b-n}} \cdot \left(\frac{2r}{2^{c-(b-n)}} + \ln(r) + c - (b-n) + 1\right) \\
&\leqslant \frac{r}{2^m} + \frac{2qr}{2^c} + \frac{(\ln(r) + n + 1)q}{2^{b-n}}.
\end{aligned}$$

Although we use chopping instead of truncation, we can reuse the analysis for bad events (vi) and (vii) in Section A.6, as those analysis are symmetrical. This gives a total bound of

$$\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} \in \mathcal{V}_{\text{bad}}\right] \leqslant \frac{q+r}{2^m} + \frac{q^2 + 2qr}{2^c} + \frac{q^2 + 2qr}{2^b} + \frac{(2\ln(r) + 2n + 2)q}{2^{b-n}}.$$

B Security of Enveloped (Theorem 4)

First of all we may assume that $q \leqslant 2^{c-2}$ as the bound holds trivially otherwise.

B.1 Simulator

The simulator is defined in Algorithm 9 and is based on the previous one given in Algorithm 8. In particular, the forward direction has remained unchanged. However, the inverse direction has been modified significantly. It mostly just gives a random result, but the mode has to satisfy one additional property in that it is difficult to invert the final compression call. This is prevented by having the IV_2 as the data path in this call, which is unlikely to be hit by chance. However, a distinguisher is able to fool a naive simulator by modifying its queries slightly. Instead of verifying the hash of a message M by building the complete tree in the forward direction, a distinguisher might do its final query in the backward direction. To perform this query the distinguisher has to get the hash of the message by querying the construction oracle beforehand. In this case the simulator cannot give a random result, but has to output IV_2. To circumvent this problem, the simulator assumes that any inverse query can be of this form and calls the construction oracle to check.

In the enveloped mode we define $\zeta_x(y) = y$ and $\text{getX}[\mathcal{L}] = \{\text{IV}_2\}$. These are changed for the feed-forward in Section C.

Algorithm 9

Interface: $\mathcal{S} : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b\ ,\ (k,x) \mapsto y$
  if $x \notin \mathrm{dom}\mathcal{L}_k$ **then**
    $S \leftarrow \mathrm{radicalExtend}[\mathcal{L}^{\mathrm{rel}}](k,x)$ ▷ radical-extend tree from single node
    if $S \in \mathcal{S}_\mathcal{T}$ **then** ▷ query completes tree
      $(M,Z) \leftarrow \mathrm{extract}(S)$ ▷ extract message and tree template
      $h \leftarrow \mathcal{RO}(M,Z)$
      $y \leftarrow \zeta_x^{-1}(h)$
    else ▷ query does not complete tree
      $y \xleftarrow{\$} \{0,1\}^b$
    end if
    $\mathcal{L}_k(x) \leftarrow y$
  end if
  return $\mathcal{L}_k(x)$

Interface: $\mathcal{S}^{-1} : \{0,1\}^\kappa \times \{0,1\}^b \to \{0,1\}^b\ ,\ (k,y) \mapsto x$
  if $y \notin \mathrm{rng}\mathcal{L}_k$ **then**
    for $x \in \mathrm{getX}[\mathcal{L}^{\mathrm{rel}}]$ **do** ▷ loop over all potential x
      $S \leftarrow \mathrm{radicalExtend}[\mathcal{L}^{\mathrm{rel}}](k,x)$
      if $S \in \mathcal{S}_\mathcal{T}$ **then**
        $(M,Z) \leftarrow \mathrm{extract}(S)$
        $h \leftarrow \mathcal{RO}(M,Z)$
        if $\zeta_x^{-1}(h) = y$ **then**
          $\mathcal{L}_k^{-1}(y) \leftarrow x$
          return x
        end if
      end if
    end for
    $x \xleftarrow{\$} \{0,1\}^b \setminus \mathrm{getX}[\mathcal{L}^{\mathrm{rel}}]$
    $\mathcal{L}_k^{-1}(y) \leftarrow x$
  end if
  return $\mathcal{L}_k^{-1}(y)$

B.2 Views

The definition of the views is very similar to the one in Section A.3, but the queries $\mathcal{L}^{\mathrm{inv}}$ are split similarly to $\mathcal{L}^{\mathrm{complete}}$ into

$$\mathcal{L}^{\mathrm{inv,known}} = \{\, (k_i, x_i, y_i) \in \mathcal{L}^{\mathrm{inv}} \mid \exists x \in \mathrm{getX}[\mathcal{L}^{\mathrm{rel}}_{i-1}] : \zeta_x^{-1}(\mathcal{M}_{i-1}(\mathrm{extractFrom}[\mathcal{L}^{\mathrm{rel}}_{i-1}](x \| k_i))) = y_i \,\},$$
$$\mathcal{L}^{\mathrm{inv,random}} = \mathcal{L}^{\mathrm{inv}} \setminus \mathcal{L}^{\mathrm{inv,known}},$$

where $\mathcal{L}^{\mathrm{inv,known}}$ are queries whose result is, mostly, as we will see later, already known to the distinguisher, whereas the result of queries in $\mathcal{L}^{\mathrm{inv,random}}$ are completely unknown to the distinguisher.

Furthermore, the different simulator has an influence on the values of the construction oracle: they are not necessarily fully random anymore, as some values can be discarded by the use of the inverse simulator. As there can be at

most q values discarded, the set of values from which the construction oracle chooses randomly, is at least $2^b - q \geqslant 2^b/2$.

This has influence on the distribution of some queries.

- $\mathcal{L}^{\text{complete}} \setminus \mathcal{L}^{\text{fwd,known}}$: as the result from the construction oracle is randomly chosen from at least $2^b/2$ values, every value has a probability of at most $2/2^b$ of occurring. We can use the same bound for queries in $\mathcal{L}^{\text{fwd,random}}$.
- $\mathcal{L}^{\text{inv,random}}$: there is a possibility that an x from $\text{getX}[\mathcal{L}^{\text{rel}}_{i-1}]$ succeeds, but every such x has a probability of at most $2/2^b$. Other values, which are distinct, are randomly chosen from at least $2^b - |\text{getX}[\mathcal{L}^{\text{rel}}]| \geqslant 2^b/2$ values.
- $\mathcal{L}^{\text{inv,known}}$: it is known that a specific $x \in \text{getX}[\mathcal{L}^{\text{rel}}_{i-1}]$ succeeds, but there is a possibility that an earlier one also succeeds. However, as in our case $|\text{getX}[\mathcal{L}^{\text{rel}}_{i-1}]| = 1$ this cannot happen.

The bad events (i)–(vii) remain mostly the same, but we have to modify bad events (iii) and (v) to also include $\lfloor \text{IV}_2 \rfloor_c$ to compensate for the minor violation of leaf-anchoring. Furthermore, we define the following extra bad events to make the analysis of bad event (vi) more structured.

(a) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ and $k_i = k_j$ such that $y_i = y_j$.
(b) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ and $k_i = k_j$ such that $y_i = y_j$.
(c) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ and $k_i = k_j$ such that $y_i = y_j$.

We also update Lemma 3 and Corollary 1 to include inverse queries, as they can complete a tree as well.

Lemma 4. *Let ν be a good view. Then for every query $(M, Z, h) \in \mathcal{M}$ we have that $(k_i, x_i, y_i) \in \mathcal{L}^{\text{complete}} \cup \mathcal{L}^{\text{inv}}$ and $\zeta_{x_i}(y_i) = h$, where $i = \iota^{\text{final}}(M, Z)$.*

Proof. This is similar to the proof of Lemma 3, but we have to change the analysis of the inverse queries. In this case the inverse queries for which the simulator finds a suitable digest can be a verification query. By subtree-freeness this can only happen for final verification queries and by design the simulator only gives consistent results for such queries. The analysis for the other inverse queries still holds and they cannot be verification queries. □

Corollary 2. *For any good view we have $|\mathcal{M}^{\text{random}}| = |\mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}|$.*

Proof. As before, this is similar to the proof of Corollary 1, but we have to include the queries $\mathcal{L}^{\text{inv,known}}$ as they can be final verification queries. □

B.3 Analysis of Bad Views

The analysis of the original bad events (i)–(v) is the same as in Section A.4, giving an upper bound of

$$\frac{q}{2^m} + \frac{q^2}{2^c}.$$

The modifications of (iii) and (v) lead to an extra term $|\mathcal{L}^{\text{inv}}|/2^c + |\mathcal{L}^{\text{rel}}|/2^c \leqslant q/2^c$.

(a) In order to make the analysis easier, we define the following helper event:
1. There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,random}}$ and $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$ with $i \neq j$ such that $y_i = h_j$.

For this helper event we know that y_i and h_j are both randomly generated from at least $2^b/2$ values, hence we get an upper bound of

$$\frac{2 \cdot |\mathcal{L}^{\text{fwd,random}}| \cdot |\mathcal{M}^{\text{random}}|}{2^b} \leqslant \frac{2r \cdot |\mathcal{L}^{\text{fwd,random}}|}{2^b}.$$

Now we derive the real bad event under the assumption that the helper event does not occur. Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$. There has to be a corresponding $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ for query j and $h_k = y_j = y_i$. As $i \neq k$ this contradicts the helper event.

(b) Again, in order to make the analysis easier, we define a helper event:
1. There exist $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ and $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$ with $i < j$ such that $h_i = h_j$.

For this helper event we know that h_j is randomly generated from at least $2^b/2$ values, hence we get an upper bound of

$$\frac{2 \cdot |\mathcal{M}^{\text{random}}|^2}{2^b} \leqslant \frac{2r \cdot |\mathcal{M}^{\text{random}}|}{2^b}.$$

Now we derive the real bad event under the assumption that the helper event does not occur. Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$. There has to be a corresponding $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ for query j and $h_k = y_j$. Similarly there has to be a corresponding $(M_\ell, Z_\ell, h_\ell) \in \mathcal{M}^{\text{random}}$ for query i and $h_\ell = y_i = h_k$. As $k \neq \ell$ this contradicts the helper event.

(c) We define the following helper events:
1. There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,random}}$ and $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$ with $i < j$ such that $y_i = h_j$.
2. There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ with $i < j$ such that $\lfloor y_j \rfloor_c = \text{radicalValue}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, \text{IV}_2)$.

For helper event (1) we know that h_j is uniformly randomly generated from at least $2^b/2$ values, leading to the following upper bound

$$\frac{2 \cdot |\mathcal{L}^{\text{inv,random}}| \cdot |\mathcal{M}^{\text{random}}|}{2^b} \leqslant \frac{2r \cdot |\mathcal{L}^{\text{inv,random}}|}{2^b}.$$

For helper event (2) we know that $\lfloor y_j \rfloor_c$ is uniformly randomly generated from at least $2^c/2$ values, giving an upper bound of

$$\frac{2 \cdot |\mathcal{L}^{\text{inv,random}}| \cdot |\mathcal{L}^{\text{rel}}|}{2^c} \leqslant \frac{2q^2}{2^c}.$$

Now we derive the real bad event under the assumption that the helper events do not occur. Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$

with $i < j$. There has to be a corresponding $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ with $k < j$ and $h_k = y_j = y_i$. Furthermore, by final leaf-anchoring we know that $x_j = \text{IV}_2$.
If $i < k$ this contradicts helper event (1) and $i = k$ cannot happen, hence we can assume that $k < i$. We separate different cases:
- $\text{radicalValue}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, \text{IV}_2) = \bot$. This means that $\text{radicalExtend}[\mathcal{L}^{\text{rel}}_{j-1}](k_j, x_j) = \text{radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, \text{IV}_2)$, therefore $\mathcal{RO}(\text{extractFrom}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, \text{IV}_2)) = y_j = y_i$, hence query i would have been in $\mathcal{L}^{\text{inv,known}}$.
- $\text{radicalValue}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, \text{IV}_2) \neq \bot$. As we know that (M_k, Z_k) is constructed in query j, this radical has to be filled after query i. However, that contradicts helper event (2).

(vi) Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd}}$ with $i < j$ and $k_i = k_j$. If $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,random}}$ then y_j is random from at least $2^b/2$ values, hence we get an upper bound of

$$\frac{2 \cdot |\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}| \cdot |\mathcal{L}^{\text{fwd,random}}|}{2^b} \leqslant \frac{2q \cdot |\mathcal{L}^{\text{fwd,random}}|}{2^b}.$$

Otherwise $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$, which means that we contradict bad event (a), (b) or (c) based on the specifics of query i.

(vii) Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{inv}}$ with $i < j$ and $k_i = k_j$. If $(k_j, x_j, y_j) \in \mathcal{L}^{\text{inv,random}}$ then x_j is randomly chosen from at least $2^b/2$ values, hence we get an upper bound of

$$\frac{2 \cdot |\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}| \cdot |\mathcal{L}^{\text{inv,random}}|}{2^b} \leqslant \frac{2q \cdot |\mathcal{L}^{\text{inv,random}}|}{2^b}$$

for this case. Otherwise, we can assume that $(k_j, x_j, y_j) \in \mathcal{L}^{\text{inv,known}}$.
Let $S_i = \text{radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k_i \| x_i)$ and $S_j = \text{radicalExtend}[\mathcal{L}^{\text{rel}}_{j-1}](k_j \| x_j)$. As $(k_j, x_j, y_j) \in \mathcal{L}^{\text{inv,known}}$, we know that $S_j \in \mathcal{S}_{\mathcal{T}}$. This means that $S_i \in \mathcal{S}_{\mathcal{T}} \cup \mathcal{S}^{\text{final}}_{\mathcal{T}} \subseteq \mathcal{S}_{\mathcal{T}} \cup \mathcal{S}^{\text{rad}}_{\mathcal{T}}$. If $S_i \in \mathcal{S}^{\text{rad}}_{\mathcal{T}}$ the radical had to be extended before query j, contradicting bad event (ii). Therefore $S_i \in \mathcal{S}_{\mathcal{T}}$ and we know that $S_i = S_j$.
If $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}}$ we get that $(k_i, x_i, y_i) \in \mathcal{L}^{\text{complete}}$ hence $y_i = y_j$, which cannot happen. If $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv}}$ we know that $x_i = x_j \in \text{getX}[\mathcal{L}^{\text{rel}}_{j-1}]$, which can only happen when query i returned a value derived from the random oracle. However, as $S_i = S_j$ this also means that $y_i = y_j$, which cannot happen.

By combining all upper bounds and using that $|\mathcal{L}^{\text{fwd,random}}| + |\mathcal{M}^{\text{random}}| + |\mathcal{L}^{\text{inv,random}}| \leqslant q$ and $|\mathcal{L}^{\text{fwd,random}}| + |\mathcal{L}^{\text{inv,random}}| \leqslant q$ we get an upper bound of

$$\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} \in \mathcal{V}_{\text{bad}}\right] \leqslant \frac{q}{2^m} + \frac{3q^2 + q}{2^c} + \frac{2q^2 + 2qr}{2^b}.$$

B.4 Analysis of Good Views

B.4.1 Real World The analysis of the real world is identical to the one in Section A.5.1, but with Lemma 4.

B.4.2 Ideal World In the ideal world the randomness is generated by both the simulator and the random oracle. For any query in $\mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}$ we trivially get an upper bound of 1 for hitting a specific value. Correspondingly, any query in $\mathcal{M}^{\text{random}}$ has a probability of at most $1/(2^b - q)$, while queries in $\mathcal{M} \setminus \mathcal{M}^{\text{random}}$ have a probability of 1 as they are correctly determined from ν_{i-1} by Lemma 4. As $|\mathcal{M}^{\text{random}}| = |\mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}|$ by Corollary 2 we can alternatively count over queries in $\mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}$ with a probability of at most $1/(2^b - q)$. Similarly, queries in $\mathcal{L}^{\text{complete}} \setminus \mathcal{L}^{\text{fwd,known}}$ have a probability of at most $1/(2^b - q)$ as well.

Any query in $\mathcal{L}^{\text{fwd}} \setminus \mathcal{L}^{\text{complete}}$ has a probability of $1/2^b$ of getting a specific value. For a query i in $\mathcal{L}^{\text{inv,random}}$ we separate based on whether it hits a value in $\text{getX}[\mathcal{L}_{i-1}^{\text{rel}}]$ or not. Every value in $\text{getX}[\mathcal{L}_{i-1}^{\text{rel}}]$ is hit with a probability of at most $1/(2^b - q)$, while any value outside $\text{getX}[\mathcal{L}_{i-1}^{\text{rel}}]$ is hit with a probability of at most $1/(2^b - |\text{getX}[\mathcal{L}_{i-1}^{\text{rel}}]|) = 1/(2^b - 1)$.

As all possibilities are upper bounded by $1/(2^b - q)$, we can use that bound for all queries, simplifying the final bound to

$$\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} = \nu\right] \leqslant \left(\frac{1}{2^b - q}\right)^q.$$

B.4.3 Ratio This gives us the following ratio:

$$\begin{aligned}\frac{\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_1} = \nu\right]}{\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} = \nu\right]} &\geqslant \left(\frac{2^b - q}{2^b}\right)^q \\ &= \left(1 - \frac{q}{2^b}\right)^q \\ &\geqslant 1 - \frac{q^2}{2^b}.\end{aligned}$$

C Security of Feed-Forward (Theorem 5)

First of all we may assume that $q \leqslant 2^{c-2}$ as the bound holds trivially otherwise. Furthermore, the simulator is the same as in Algorithm 9, but we have to update the finalization function to $\zeta_x(y) = x \oplus y$ and the set of possible inputs to $\text{getX}[\mathcal{L}] = \{\text{IV}'_1\} \cup \{\, \lfloor y \rfloor_c \| \text{IV}_2 : (k, x, y) \in \mathcal{L}, \text{IV}_2 \in \mathcal{IV}_2 \,\}$ accordingly.

C.1 Views

The views are similar to the ones in Section B.2, but we have to modify the extra events. Before we can do that we define some extra procedures in order to define them.

Definition 13. *Given a message (M, Z) and a list $\mathcal{L}$ we define* $\text{buildForward}[\mathcal{L}](M, Z)$ *as the procedure that builds a partial tree corresponding to (M, Z) up to the data input x of the final node and as far as possible from the bottom up by using*

queries from $\mathcal{L}$. This is similar to computing the hash result of (M, Z), but ignores the key input and it might not finish. We define $\text{buildX}[\mathcal{L}](M, Z)$ *as the data input x of the final node, or $\perp$ if it does not finish.*

When additionally given an $x \in \text{getX}[\mathcal{L}]$ we define $\text{buildBackward}[\mathcal{L}](M, Z, \lfloor x \rfloor_c)$ *as the procedure that builds the tree corresponding to (M, Z) as far as possible by starting with $\lfloor x \rfloor_c$ as the first c bits in the final node and filling in all chaining values. Note that $\lfloor x \rfloor_c$ is either $\lfloor \text{IV}_1' \rfloor_c$ or a chaining value. This is similar to the procedure* $\text{radicalExtend}[\mathcal{L}](k, x)$*, but it does not need the key k as it already knows the template from the message (M, Z) and it tries to fill in all chaining values, not just one.*

We will now define the additional bad events.

(a) There exists a $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,known}}$ such that $x_i \neq x$, where x is the predicted value.
(b) Let $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$, $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ and $x \in \{0,1\}^b$ with $i < j$. This bad event occurs when:
  – $\text{buildX}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i) = \perp$, but $\text{buildX}[\mathcal{L}^{\text{rel}}_{j}](M_i, Z_i) \neq \perp$.
  – $\text{buildForward}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i)$ and $\text{buildBackward}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i, x)$ do not contradict each other.
  – $\text{buildBackward}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i, x)$ is not trivial.
(c) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ and $k_i = k_j$ such that $y_i = y_j$.
(d) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ and $k_i = k_j$ such that $y_i = y_j$.
(e) There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$ and $k_i = k_j$ such that $y_i = y_j$.

Bad event (a) ensures that the predicted value for queries in $\mathcal{L}^{\text{inv,known}}$ will be hit, bad event (b) will make sure that there will be a unique query that defines the $\lfloor x \rfloor_c$ in the feed-forward, which will make reasoning over $\lfloor y \rfloor_c = \lfloor x \oplus h \rfloor_c$ easier. The bad events (c), (d) and (e) will make the analysis of (i) more structured.

We note that Corollary 2 still holds as bad event (a) ensures that the queries in $\mathcal{L}^{\text{inv,known}}$ behave as expected and verify the correct messages.

C.2 Analysis of Bad Views

The analysis of bad events (i)–(v) are the same as in Section A.4, giving an upper bound of

$$\frac{q}{2^m} + \frac{q^2}{2^c}.$$

The main goal of the analysis is to make sure that there is no permutation inconsistency in the simulator. Most outputs are randomly generated, making such analysis straightforward. However, the simulator also has to be consistent with the construction oracle, which forces some outputs. In particular, the queries in $\mathcal{L}^{\text{fwd,known}}$ are difficult to analyze. As the mode uses a feed-forward in the

finalization, we know that $h = x \oplus y$, where h is the hash digest, x is the input to the final query and y is the output of the final query. This means that the final output $y = h \oplus x$ is determined when both the hash digest h and the input x are known. As both h and $\lfloor x \rfloor_c$ are randomly generated, the same holds for $\lfloor y \rfloor_c$ when the last one of h and $\lfloor x \rfloor_c$ is. However, there are a few caveats:

- If the message consists of a single message block, x equals the fixed value IV'_1. This does not impose any problems, as this means that h, which is random, will always be generated last.
- The distinguisher might make clever queries which make $\lfloor x \rfloor_c$ not uniformly random. For example, take a simple Merkle-Damgård construction. The distinguisher can query almost all blocks, except the first and last, where it guesses the first chaining value. When constructing the start of the message it might hit the first chaining value and if it misses it might hit the second, etc. All in all, this means that some chosen values of $\lfloor x \rfloor_c$ might be more probable than others, although not significantly so. This would make the analysis very complicated, so we introduce bad event (b) to make sure that there is only a specific query that determines $\lfloor x \rfloor_c$, making sure that it is randomly generated. The problematic cases could also be prevented by making the procedure radicalExtend$[\mathcal{L}](k, x)$ not depend on the unknown k. However, this modification is not possible without compromising the generality.

Definition 14. *Let* $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$. *Query* $j \geqslant i$ completes (M_i, Z_i) *if:*

- $j = i$ *and* $\text{buildX}[\mathcal{L}^{\text{rel}}_{i-1}](M_i, Z_i) \neq \perp$.
- $j > i$, $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ *and* j *is the final node in* $\text{buildForward}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i)$ *(so* $\lfloor \text{buildX}[\mathcal{L}^{\text{rel}}_j](M_i, Z_i) \rfloor_c = \lfloor y_j \rfloor_c$*).*

Note that a query can complete multiple messages. As the distinguisher verifies all the queries in $\mathcal{M}^{\text{random}}$ *at the end, we know that for every* $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ *there is always exactly one query* $y^{\text{index}}(M_i, Z_i) \geqslant i$ *that completes* (M_i, Z_i) *with value* $y^{\text{final}}(M_i, Z_i) = \text{buildX}[\mathcal{L}^{\text{rel}}_j](M_i, Z_i) \oplus h_i$, *where* $j = y^{\text{index}}(M_i, Z_i)$.

This definition does not depend on the output of query $y^{\text{index}}(M_i, Z_i)$ hence $\lfloor y^{\text{final}}(M_i, Z_i) \rfloor_c$ can be considered random at query $y^{\text{index}}(M_i, Z_i)$. Due to the fact that the construction oracle no longer perfectly random, the probability of getting a specific value is at most $2^{b-c}/(2^b - q) \leqslant 2/2^c$. Furthermore, after this query the output of the verification query of (M_i, Z_i) is determined.

Lemma 5. *Let* $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ *and let* y *be the output of the verification query of* (M_i, Z_i). *If bad event (b) does not occur, we have* $y = y^{\text{final}}(M_i, Z_i)$.

Proof. Let $j \geqslant i$ be the query that determines y. If $j = i$ we have that $\text{buildX}[\mathcal{L}^{\text{rel}}_{i-1}](M_i, Z_i) \neq \perp$, hence $j = y^{\text{index}}(M_i, Z_i)$. If $j > i$ we know that $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ and that query j determines x. This means that $\text{buildX}[\mathcal{L}^{\text{rel}}_j](M_i, Z_i) = x \neq \perp$ and $\text{buildX}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i) = \perp$. Furthermore, $\text{buildForward}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i)$ and $\text{buildBackward}[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i, x)$ cannot contradict each other, as query j gives as

output x. Finally, by bad event (b) this means that buildBackward$[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i, x)$ has to be trivial, which means that the final query has not happened in the first $j-1$ queries, hence it happened at query j and we get $y = y^{\text{final}}(M_i, Z_i)$ as desired. □

(a) For query $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,known}}$ there are at most $|\text{getX}[\mathcal{L}^{\text{rel}}_{i-1}]| \leqslant 1 + P(q-1) \leqslant Pq$ possibilities before the predicted x that can be hit. Every such possibility has a probability of at most $2/2^b$ to be hit, which leads to an upper bound of

$$\frac{2 \cdot |\mathcal{L}^{\text{inv,known}}| \cdot |\text{getX}[\mathcal{L}^{\text{rel}}]|}{2^b} \leqslant \frac{2Pqr}{2^b}.$$

(b) In order to upper bound this bad event, we first show that for every $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ and $x \in \{0,1\}^b$ there is at most one possible $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ that has a possibility of satisfying bad event (b). For this, we first define precisely what such query is.

Definition 15. *Let $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$, $(k_j, x_j, y_j) \in \mathcal{L}^{\text{rel}}$ and $x \in \{0,1\}^b$ with $i < j$. We call query j a* candidate *when there is a $y \in \{0,1\}^b$ such that bad event (a) is satisfied, but where in the final condition* buildForward$[\mathcal{L}^{\text{rel}}_j](M_i, Z_i)$ *is replaced by* buildForward$[\mathcal{L}^{\text{rel}}_{j-1} \cup \{(k_j, x_j, y)\}](M_i, Z_i)$. *Note that this definition does not depend on the output y_j.*

Now we show that there is at most one such candidate for every $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ and $x \in \{0,1\}^b$.

Lemma 6. *For every $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ and $x \in \{0,1\}^b$ there is at most one candidate.*

Proof. Let j be the first candidate, if one exists. If j is successful, we have buildX$[\mathcal{L}^{\text{rel}}_{k-1}](M_i, Z_i) \neq \perp$ for any $k > j$, hence there can be no later candidate. If j is not successful it means that buildForward$[\mathcal{L}^{\text{rel}}_j](M_i, Z_i)$ and buildBackward$[\mathcal{L}^{\text{rel}}_j](M_i, Z_i, x)$ contradict each other, as j was the final part missing. As these keep contradicting each other in further queries, no later candidate is possible. □

First of all, as buildBackward$[\mathcal{L}^{\text{rel}}_{j-1}](M_i, Z_i, x)$ is not trivial, we need to have that $x \in \text{getX}[\mathcal{L}^{\text{rel}}_{j-1}]$. As the definition only depends on $\lfloor x \rfloor_c$ this leads to at most q possible values for x. Furthermore, by Lemma 6 there is at most one candidate, hence we can upper bound the bad event by

$$\frac{2 \cdot |\mathcal{M}^{\text{random}}| \cdot q}{2^c} \leqslant \frac{2qr}{2^c}.$$

(c) In order to make the analysis easier, we define the following helper events:
1. There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,random}}$ and $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$ with $i \neq y^{\text{index}}(M_j, Z_j)$ such that $\lfloor y_i \rfloor_c = \lfloor y^{\text{final}}(M_j, Z_j) \rfloor_c$.
2. There exists $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ such that $\lfloor h_i \rfloor_c = 0^c$.

For these helper events we know that $\lfloor y_i \rfloor_c$, $\lfloor y^{\text{final}}(M_j, Z_j) \rfloor_c$ (helper event (1)) and $\lfloor h_i \rfloor_c$ (helper event (2)) are all randomly generated from at least $2^c/2$ values. This leads to the following upper bound

$$\frac{2 \cdot |\mathcal{L}^{\text{fwd,random}}| \cdot |\mathcal{M}^{\text{random}}|}{2^c} + \frac{2 \cdot |\mathcal{M}^{\text{random}}|}{2^c} \leqslant \frac{2 \cdot |\mathcal{L}^{\text{fwd,random}}| \cdot r}{2^c} + \frac{2r}{2^c}.$$

Now we derive the real bad event under the assumption that the helper events do not occur. Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$. There has to be a corresponding $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ for query j with $k' = y^{\text{index}}(M_k, Z_k) < j$, hence $y^{\text{final}}(M_k, Z_k) = y_j = y_i$.
If $i \neq k'$ this contradicts helper event (1) and if $i = k'$ then $\lfloor h_k \rfloor_c = \lfloor y_j \oplus \text{buildX}[\mathcal{L}^{\text{rel}}_{k'}](M_k, Z_k) \rfloor_c = \lfloor y_j \oplus y_i \rfloor_c = 0^c$, contradicting helper event (2).

(d) Again, in order to make the analysis easier, we define some helper events:
1. There exist $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ and $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$ with $y^{\text{index}}(M_i, Z_i) < y^{\text{index}}(M_j, Z_j)$ such that $\lfloor y^{\text{final}}(M_i, Z_i) \rfloor_c = \lfloor y^{\text{final}}(M_j, Z_j) \rfloor_c$.
2. There exist $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ and $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$ with $i < j$ such that $\lfloor h_i \rfloor_c = \lfloor h_j \rfloor_c$.

As $\lfloor y^{\text{final}}(M', Z') \rfloor_c$ (helper event (1)) and $\lfloor h_j \rfloor_c$ (helper event (2)) are randomly generated from at least $2^c/2$ values, we get the following upper bound for these helper events:

$$\frac{2 \cdot |\mathcal{M}^{\text{random}}|^2}{2^c} + \frac{2 \cdot |\mathcal{M}^{\text{random}}|^2}{2^c} \leqslant \frac{4r^2}{2^c}.$$

Now we derive the real bad event under the assumption that the helper events do not occur. Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd,known}} \cup \mathcal{L}^{\text{inv,known}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$. There has to be a corresponding $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ for query j with $k' = y^{\text{index}}(M_k, Z_k) < j$, hence $y^{\text{final}}(M_k, Z_k) = y_j = y_i$. Similarly there has to be such corresponding $(M_\ell, Z_\ell, h_\ell) \in \mathcal{M}^{\text{random}}$ for query i with $\ell' = y^{\text{index}}(M_\ell, Z_\ell) < i$.
If $k' \neq \ell'$ this contradicts helper event (1) and $k' = \ell'$ can only happen when $x = \text{buildX}[\mathcal{L}^{\text{rel}}_{k'}](M_k, Z_k) = \text{buildX}[\mathcal{L}^{\text{rel}}_{\ell'}](M_\ell, Z_\ell)$, hence $\lfloor h_k \rfloor_c = \lfloor y_i \oplus x \rfloor_c = \lfloor y_j \oplus x \rfloor_c = \lfloor h_\ell \rfloor_c$, which contradicts helper event (2).

(e) We define the following helper events:
1. There exist $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,random}}$ and $(M_j, Z_j, h_j) \in \mathcal{M}^{\text{random}}$ with $i < y^{\text{index}}(M_j, Z_j)$ such that $\lfloor y_i \rfloor_c = \lfloor y^{\text{final}}(M_j, Z_j) \rfloor_c$.
2. There exist $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$, $(k_j, x_j, y_j) \in \mathcal{L}^{\text{inv,random}}$ and $(k_k, x_k, y_k) \in \mathcal{L}^{\text{rel}}$ with $i \leqslant y^{\text{index}}(M_i, Z_i) < j < k$ and $y^{\text{final}}(M_i, Z_i) = y_j$ such that $\lfloor y_k \rfloor_c = \text{radicalValue}[\mathcal{L}^{\text{rel}}_{j-1}](k_j, x)$, where $x = h_i \oplus y_j$.

For helper event (1) we know that $\lfloor y^{\text{final}}_j(M, Z) \rfloor_c$ is randomly generated from at least $2^c/2$ values, leading to the following upper bound

$$\frac{2 \cdot |\mathcal{L}^{\text{inv,random}}| \cdot |\mathcal{M}^{\text{random}}|}{2^c} \leqslant \frac{2 \cdot |\mathcal{L}^{\text{inv,random}}| \cdot r}{2^c}.$$

For helper event (2) we know that $\lfloor y_k \rfloor_c$ is randomly generated from at least $2^c/2$ values and in the previous bad event we saw that for every $\lfloor y_j \rfloor_c$ there

is at most one $(M_i, Z_i, h_i) \in \mathcal{M}^{\text{random}}$ with $\lfloor y^{\text{final}}(M_i, Z_i) \rfloor_c = \lfloor y_j \rfloor_c$, hence we get the upper bound of

$$\frac{2 \cdot |\mathcal{L}^{\text{inv,random}}| \cdot |\mathcal{L}^{\text{rel}}|}{2^c} \leqslant \frac{2q^2}{2^c}.$$

Now we derive the real bad event under the assumption that the helper events do not occur. Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{inv,random}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$ with $i < j$. There has to be a corresponding $(M_k, Z_k, h_k) \in \mathcal{M}^{\text{random}}$ with $k' = y^{\text{index}}(M_k, Z_k) < j$, hence $y^{\text{final}}(M_k, Z_k) = y_j = y_i$.
If $i < k'$ this contradicts helper event (1) and $i = k'$ cannot happen as k' cannot be an inverse query, hence we can assume that $k' < i$. This also means that $x = h_k \oplus y^{\text{final}}(M_k, Z_k)$ is known at query i and that $x \in \text{getX}[\mathcal{L}^{\text{rel}}_{i-1}]$.
We separate different cases:

- $\text{radicalValue}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x) = \perp$. As $\text{radicalExtend}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x) = \text{radicalExtend}[\mathcal{L}^{\text{rel}}_{j-1}](k_j, x)$ we know that $\zeta_x^{-1}(\mathcal{RO}(\text{extractFrom}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x))) = y_j = y_i$, hence query i would have been in $\mathcal{L}^{\text{inv,known}}$.
- $\text{radicalValue}[\mathcal{L}^{\text{rel}}_{i-1}](k_i, x) \neq \perp$. As we know that (M_k, Z_k) is constructed in query j, this radical has to be filled after query i. However, that contradicts helper event (2).

Now we look at the 'real' bad events (vi) and (vii).

(vi) Let $(k_i, x_i, y_i) \in \mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}$ and $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd}}$ with $i < j$ and $k_i = k_j$. If $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,random}}$ then y_j is randomly chosen from at least $2^b/2$ values, hence we get an upper bound of

$$\frac{2 \cdot |\mathcal{L}^{\text{fwd}} \cup \mathcal{L}^{\text{inv}}| \cdot |\mathcal{L}^{\text{fwd,random}}|}{2^b} \leqslant \frac{2q \cdot |\mathcal{L}^{\text{fwd,random}}|}{2^b}.$$

Otherwise $(k_j, x_j, y_j) \in \mathcal{L}^{\text{fwd,known}}$, which means that we contradict bad event (c), (d) or (e) based on the specifics of query i.

(vii) The analysis is the same as in Section B, giving an upper bound of

$$\frac{2q \cdot |\mathcal{L}^{\text{inv,random}}|}{2^b}.$$

By combining all upper bounds and using that $|\mathcal{L}^{\text{fwd,random}}| + |\mathcal{L}^{\text{inv,random}}| \leqslant q$ we get an upper bound of

$$\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} \in \mathcal{V}_{\text{bad}}\right] \leqslant \frac{2q^2 + 4qr + 4r^2 + 2r}{2^c} + \frac{2q^2 + 2Pqr}{2^b}.$$

C.3 Analysis of Good Views

The analysis of the good views is identical to the one in Section B, giving a ratio of

$$\frac{\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_1} = \nu\right]}{\mathbb{P}\left[\mathcal{D}'_{\mathcal{O}_2} = \nu\right]} \geqslant 1 - \frac{q^2}{2^b}.$$