The Parazoa Family: Generalizing the Sponge Hash Functions

Elena Andreeva, Bart Mennink and Bart Preneel

Dept. Electrical Engineering, ESAT/COSIC and IBBT
Katholieke Universiteit Leuven, Belgium
{elena.andreeva, bart.mennink, bart.preneel}@esat.kuleuven.be

Abstract. Sponge functions were introduced by Bertoni et al. as an alternative to the classical Merkle-Damgård design. Many hash function submissions to the SHA-3 competition launched by NIST in 2007, such as CubeHash, Fugue, Hamsi, JH, Keccak and Luffa, derive from the original sponge design, and security guarantees from some of these constructions are typically based on indifferentiability results. Although indifferentiability proofs for these designs often bear significant similarities, these have so far been obtained independently for each construction. In this work, we introduce the *parazoa* family of hash functions as a generalization of "sponge-like" functions. Similarly to the sponge design, the parazoa family consists of compression and extraction phases. The parazoa hash functions, however, extend the sponge construction by enabling the use of a wider class of compression and extraction functions that need to satisfy certain properties. More importantly, we prove that the parazoa functions satisfy the indifferentiability notion of Maurer et al. under the assumption that the underlying permutation is ideal. Not surprisingly, our indifferentiability result confirms the bound on the original sponge function, but it also carries over to a wider spectrum of hash functions and eliminates the need for a separate indifferentiability analysis.
Keywords. Parazoa functions, sponge functions, hash function design, indifferentiability.

1 Introduction

Traditionally, hash functions are designed following the Merkle-Damgård iterative design [16, 26]: to construct a cryptographic hash function $\mathcal{H} : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$ that maps bit strings of arbitrary length to outputs of fixed length, one first builds a fixed input length compression function $f : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \to \mathbb{Z}_2^n$, and then applies it in an iterative manner. Here, the input message $M \in \mathbb{Z}_2^*$ is first *padded* injectively into a bit string of a length multiple of m. The main design objective behind the Merkle-Damgård iteration is collision security preservation: showing that the Merkle-Damgård hash function is collision secure (Col) when the underlying compression function f is assumed to be also Col secure. The preservation is achieved by applying the Merkle-Damgård-*strengthening* [24]: a suffix-free message padding function used in conjunction with a fixed initialization value IV. A broad range of hash function applications requires further security requirements, such as preimage resistance, second preimage resistance, and resistance to the length extension attack. Together with Col security, these are outlined as the *main* security requirements by NIST [27] in their call for the design of a future SHA-3 hash algorithm. Unfortunately, the strengthened Merkle-Damgård design does not preserve the properties of second preimage and preimage security [2], and moreover, it does not preclude length extension attacks [15]. As a result, several Merkle-Damgård design alternatives have appeared in the literature. These achieve some of the former security properties and among others include the chop-Merkle-Damgård [18], prefix-free Merkle-Damgård [15], HAIFA [13], NMAC [4] and Enveloped Merkle-Damgård [5] hash functions. Provided that the padding rule is suffix-free, all of the outlined hash functions preserve Col [1].
To exhibit a preservation result for a security property X, one assumes that the underlying compression function f also satisfies the property X. Although this idea is widely employed, one may wonder if it is strictly needed. Instead, one may consider a different approach and iterate a weak compression functions sufficiently many times to obtain a strong hash function.
In 2007, Bertoni et al. introduced the *sponge* hash functions [10] as an alternative of the Merkle-Damgård design. The sponge hash function design begins with an absorbing phase, in which the message is compressed iteratively, and ends with an extraction phase, in which the hash digest is extracted in a possibly iterative manner. The sponge design idea is to obtain a secure hash function by iterating a compression function that does not necessarily satisfy the main hash function security properties. Sponge functions iteratively "absorb" message blocks of r-bits per compression function f call, where

the iterated state is of size $r + c$ bits with c being the so-called "capacity". Finally, the hash digest is extracted r bits at a time by applying the extraction function g. The sponge function employs a single $(r + c)$-bits permutation π, and the compression function f and extraction function g are defined as $f(v_r \| v_c, M_i) = \pi((v_r \oplus M_i) \| v_c)$, where M_i is an r-bit message block, and $g(v_r \| v_c) = (\pi(v_r \| v_c), v_r)$. Following the indifferentiability framework of Maurer et al. [25], it has been proven in [8] that sponge functions are indifferentiable from a random oracle under the assumption that π behaves like a random permutation. In particular, a sponge function behaves like a random oracle for up to $O(2^{c/2})$ queries. Since the introduction of sponge functions, it has been a standard practice in the cryptographic community to call hash functions *"sponge-like"* if they bear resemblances with the original sponge design in terms of iterating a wide state and employing underlying permutations in an extraction and absorbing phases. Despite the similarities, the indifferentiability results of the sponge hash function do not carry over in a straightforward manner to the "sponge-like" constructions and hence, an independent security analysis is required. At times even a small adjustment to the sponge design may render it insecure (cf. App. A). It is thus an interesting research problem to come up with a secure class of hash functions generalizing the original sponge construction, and the results of which could simply be carried over to its members.

1.1 Our Contributions

In this work, we introduce the *parazoa* family of hash functions[1], as a generalization of the sponge hash function. Our generalization is crafted towards obtaining secure "sponge-like" hash functions in the indifferentiability theoretical framework by Maurer et al. [25]. The parazoa hash function family allows for a wider class of compression and extraction functions that satisfy a set of simple conditions. These conditions facilitate the indifferentiability proof, but we note that these are easily satisfied and realistic for practical purposes. Similar to the original sponge design, parazoa functions allow for variable length outputs. In [28, Sect. 4.2], Stam analyzes permutation based compression functions satisfying certain criteria, "overloaded single call Type-I compression functions", that are similar to the compression functions employed in the parazoa design (albeit the requirements posited in [28, Def. 17] are stronger). The major difference is that in the parazoa design the compression function is *not* required to be preimage/collision resistant. In particular, Stam leaves it as an open problem to analyze security of overloaded single call compression functions in the iteration.
We prove that the maximum advantage of any distinguisher in differentiating a parazoa hash function, based on ideal primitive π, from a random oracle is upper bounded by $O((Kq)^2/2^{s-p-d})$, where the distinguisher makes at most q queries of length at most K blocks. Here, s denotes the iterated state size, p denotes the number of bits extracted in one execution of the extraction function, and d is called the capacity loss, is a quantity inherent to the specific parazoa design (cf. Table 1). Even though the indifferentiability proof focuses on parazoa designs where both the compression and extraction function are based on one single permutation, the result easily extends to designs where multiple random permutations and/or random functions are employed.
Naturally, the sponge function design [10] falls within the categorization of parazoa functions, and our indifferentiability result confirms the bound of [8]. Additional hash function designs covered by the parazoa specification are Grindahl [22], and SHA-3 candidate hash functions CubeHash [7], Fugue [20], JH [30], Keccak [9] and (a restricted variant of) Luffa [17], two of which advanced to the final round of NIST's hash function competition. The implications of our indifferentiability results on these functions are summarized in Table 1, and we elaborate on it in Sect. 5. We note that not all obtained bounds are as expected. In particular, our indifferentiability bound on JH is worse than the indifferentiability bound proven by Bhattacharyya et al. [12]. The difference may be a price to pay in return for generality. For the generic parazoa design, we note that our indifferentiability bound is optimal: for the original sponge design, the best generic attack meets the derived security bound [10]. Still, for concrete instantiations of the parazoa hash function a design-specific proof may result in a better bound.

[1] Parazoa is the name of the subkingdom of animals to which the sponges belong [29].

Table 1. Implications of the indifferentiability result for the original sponge function design, the Grindahl hash function, and several second round SHA-3 candidates. Here, s is the internal state size, m the number of message bits compressed in one round, p the number of bits extracted in one extraction round, n the number of output bits, and the capacity loss d is further explained in Sect. 4. Parameter q denotes the total number of queries made by the distinguisher, and K is the maximal length of these queries in blocks. For the second round SHA-3 candidates, $n \in \{224, 256, 384, 512\}$. The hash functions JH and Keccak advanced to the final round of NIST's SHA-3 competition. The results hold under the assumption that the underlying permutations are ideal. For concrete instantiations of these permutations, we refer to Sect. 5. For Fugue, an indifferentiability result has been derived by Halevi et al. [21], but we do not include the bound as their work considers a different model.

Algorithm	$(\boldsymbol{s, m, p})$	$\boldsymbol{d}$	**Existing indiff. bound**	**Our indiff. bound**
Sponge	$(r+c, r, r)$	0	$O\left((Kq)^2/2^c\right)$ [8]	$O\left((Kq)^2/2^c\right)$
Grindahl	(s, m, n)	m	—	$O\left((Kq)^2/2^{s-n-m}\right)$
Quark	$(r+c, r, r)$	0	$O\left((Kq)^2/2^c\right)$ [8]	$O\left((Kq)^2/2^c\right)$
PHOTON ($r' \leq r$)	$(r+c, r, r')$	$r-r'$	$O\left((Kq)^2/2^c\right)$ [8]	$O\left((Kq)^2/2^c\right)$
PHOTON ($r' \geq r$)	$(r+c, r, r')$	0	—	$O\left((Kq)^2/2^{c+r-r'}\right)$
SPONGENT	$(r+c, r, r)$	0	$O\left((Kq)^2/2^c\right)$ [8]	$O\left((Kq)^2/2^c\right)$
CubeHash-n	$(1024, 257, n)$	1	—	$O\left((Kq)^2/2^{1023-n}\right)$
Fugue-n ($n \leq 256$)	$(960, 32, n)$	m	[21]	$O\left((Kq)^2/2^{928-n}\right)$
Fugue-n ($n > 256$)	$(1152, 32, n)$	m	[21]	$O\left((Kq)^2/2^{1120-n}\right)$
JH-n	$(1024, 512, n)$	m	$O\left(q^3/2^{512} + Kq^3/2^{1024-n}\right)$ [12]	$O\left((Kq)^2/2^{512-n}\right)$
Keccak-n	$(1600, s-2n, n)$	$s-3n$	$O\left((Kq)^2/2^{2n}\right)$ [8]	$O\left((Kq)^2/2^{2n}\right)$
Luffa-n ($n \leq 256$)	$(768, 256, 256)$	0	—	$O\left((Kq)^2/2^{512}\right)$
Luffa-384	$(1024, 256, 256)$	0	—	$O\left((Kq)^2/2^{768}\right)$
Luffa-512	$(1280, 256, 256)$	0	—	$O\left((Kq)^2/2^{1024}\right)$

1.2 Outline

In Sect. 2, we introduce some mathematical background. Parazoa functions are introduced and formalized in Sect. 3, and an indifferentiability result for parazoa functions is given in Sect. 4. We finish the paper with concluding remarks in Sect. 5.

2 Preliminaries

By $\mathbb{Z}_2^*$ we denote the set of bit strings of arbitrary length. For a positive integer $n \in \mathbb{N}$, we denote by $\mathbb{Z}_2^n$ the set of bit strings of length n and by $(\mathbb{Z}_2^n)^*$ the set of bit strings of length a multiple of n. For two bit strings x, y, we denote by $x\|y$ their concatenation. The function $\mathsf{chop}_n(x)$ chops off the n rightmost bits of a bit string x. If $\mathcal{X}$ is a set, by $x \stackrel{\$}{\leftarrow} \mathcal{X}$ we denote the uniformly random sampling of an element from $\mathcal{X}$. By $y \leftarrow \mathsf{A}(x)$ and $y \stackrel{\$}{\leftarrow} \mathsf{A}(x)$, we denote the assignment to y of the output of a deterministic and randomized algorithm A, respectively, when run on input x. For a function f, by $\mathsf{dom}(f)$ and $\mathsf{rng}(f)$ we denote its domain and range, respectively. A random oracle [6] is a function that provides a random output for each new query. A random l-bit permutation is a function that is taken uniformly at random from the set of all l-bit permutations. A random primitive will also be called "ideal". A function $f : \mathbb{Z}_2^m \to \mathbb{Z}_2^n$ for $m \geq n$ is called *balanced* if any $y \in \mathbb{Z}_2^n$ has exactly 2^{m-n} preimages under f. We define its inverse function by $f^{-1} : y \mapsto \{x \in \mathbb{Z}_2^m \mid f(x) = y\}$.

2.1 Indifferentiability

The indifferentiability framework, introduced by Maurer et al. [25], is a powerful notion to guarantee security of cryptographic primitives. Informally, it gives a sufficient condition under which an ideal primitive $\mathcal{R}$ can be replaced by some construction $\mathcal{C}^{\mathcal{G}}$ based on an ideal subcomponent $\mathcal{G}$. The indifferentiability bound provides security guarantees from the hash function against any security attack [1].

Definition 1. *A Turing machine $\mathcal{C}$ with oracle access to an ideal primitive $\mathcal{G}$ is called $(t_D, t_S, q, \varepsilon)$ indifferentiable from an ideal primitive $\mathcal{R}$ if there exists a simulator $\mathcal{S}$, such that for any information-theoretic distinguisher $\mathcal{D}$ it holds that:*

$$\mathbf{Adv}^{\mathsf{pro}}_{\mathcal{C},\mathcal{S}}(\mathcal{D}) = \left|\mathbf{Pr}\left(\mathcal{D}^{\mathcal{C}^{\mathcal{G}},\mathcal{G}} = 1\right) - \mathbf{Pr}\left(\mathcal{D}^{\mathcal{R},\mathcal{S}^{\mathcal{R}}} = 1\right)\right| < \varepsilon.$$

The simulator has oracle access to $\mathcal{R}$ and runs in time at most t_S. The distinguisher runs in time at most t_D and makes at most q queries.

Distinguisher $\mathcal{D}$ can query both its "left oracle" L (either $\mathcal{C}$ or $\mathcal{R}$) and its "right oracle" R (either $\mathcal{G}$ or $\mathcal{S}$). We refer to $\mathcal{C}^{\mathcal{G}}, \mathcal{G}$ as the "real world", and to $\mathcal{R}, \mathcal{S}^{\mathcal{R}}$ as the "simulated world"; the distinguisher $\mathcal{D}$ converses with either of these worlds and its goal is to tell both worlds apart. In the remainder, $\mathcal{R}$ will be a random oracle RO, and $\mathcal{G}$ will be a random permutation π. The right oracle R has two interfaces, as the distinguisher can make forward as well as inverse queries to the permutation π.

3 Parazoa Functions

Informally, parazoa functions process a message M as follows. Firstly, the message is *padded* into several integral message blocks of m bits, using a padding function $\mathsf{pad} : \mathbb{Z}_2^* \to (\mathbb{Z}_2^m)^*$. Throughout, by k we denote the number of message blocks of a padded message. Then, these message blocks are *absorbed* by the s-bit state (compression phase), by applying sequentially a compression function $f : \mathbb{Z}_2^s \times \mathbb{Z}_2^m \to \mathbb{Z}_2^s$ on the state and the message. Next, the state is *squeezed* to obtain $l \geq 1$ output data blocks of p bits sequentially (extraction phase). The corresponding extraction function is denoted by $g : \mathbb{Z}_2^s \to \mathbb{Z}_2^s \times \mathbb{Z}_2^p$. It operates on the state, and returns an updated state and the extract. A *finalization* function $\mathsf{fin} : \mathbb{Z}_2^{pl} \to \mathbb{Z}_2^n$ combines these l data blocks of p bits into the n-bit message digest. We require that $m, p \leq s$, and that $pl \geq n$. Both the compression function and the extraction function are based on an s-bit permutation π. Throughout, we assume this permutation to be ideal.
These functions are further explained in Sects. 3.1-3.4 (for ease of presentation, the function pad is introduced at last), together with the requirements of these functions for the security proof in Sect. 4. Now, for a fixed initialization vector IV of size s, the parazoa function $\mathcal{H}$ processes a message M as follows:

$$\begin{aligned} \mathcal{H}(M) = h, \text{ where: } \quad & (M_1, \ldots, M_k) \leftarrow \mathsf{pad}(M); \quad v_0 \leftarrow \mathsf{IV}, && \text{(1a)} \\ & v_i \leftarrow f(v_{i-1}, M_i) \text{ for } i = 1, \ldots, k, && \text{(1b)} \\ & (v_{k+i}, P_i) \leftarrow g(v_{k+i-1}) \text{ for } i = 1, \ldots, l, && \text{(1c)} \\ & h \leftarrow \mathsf{fin}(P_1, \ldots, P_l). && \text{(1d)} \end{aligned}$$

This function is depicted in Fig. 1.

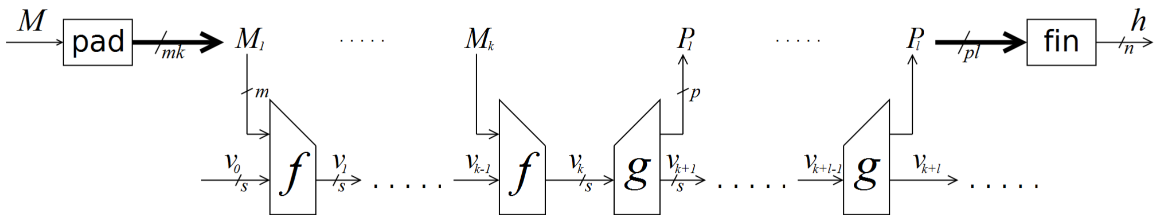

Fig. 1. The parazoa hash function $\mathcal{H}$ of (1).

3.1 Compression Function f

On input of a state value v_{i-1} and a message input M_i, the compression function first uses an injection function $\mathsf{L}_{\text{in}} : \mathbb{Z}_2^s \times \mathbb{Z}_2^m \to \mathbb{Z}_2^s$ to inject the message into the state, and then permutes the state with π. This state is then transformed and combined with a feed-forward using a function $\mathsf{L}_{\text{out}} : \mathbb{Z}_2^s \times \mathbb{Z}_2^s \times \mathbb{Z}_2^m \to \mathbb{Z}_2^s$. Formally, the compression function f is defined as $f(v_{i-1}, M_i) = v_i$, where $x \leftarrow \mathsf{L}_{\text{in}}(v_{i-1}, M_i)$, $y \leftarrow \pi(x)$ and $v_i \leftarrow \mathsf{L}_{\text{out}}(y, v_{i-1}, M_i)$. The compression function f is depicted in Fig. 2.

For any $x \in \mathbb{Z}_2^s$ we define its *capacity set* $\mathcal{C}(x) = \{v \in \mathbb{Z}_2^s \mid \exists\, M \in \mathbb{Z}_2^m \text{ s.t. } \mathsf{L}_{\text{in}}(v, M) = x\}$. Intuitively, $\mathcal{C}(x)$ denotes the set of state values $v \in \mathbb{Z}_2^s$ for which some message injection results in x as input to the permutation. For two values $v, v' \in \mathbb{Z}_2^s$, we define the function $\mathsf{same}\mathcal{C}(v, v')$, that outputs **true** if and only if v and v' are both a member of a capacity set $\mathcal{C}(x)$ for some x.

Requirement from L_{in}. We require L_{in} to satisfy the following properties: (a) for any $x \in \mathbb{Z}_2^s$ and $v \in \mathcal{C}(x)$, there exists *exactly one* $M \in \mathbb{Z}_2^m$ such that $\mathsf{L}_{\text{in}}(v, M) = x$, and (b) if $\mathcal{C}(x) \cap \mathcal{C}(x') \neq \emptyset$, then $\mathcal{C}(x) = \mathcal{C}(x')$. Intuitively, the first requirement guarantees that for a state value $v \in \mathbb{Z}_2^s$, a different M results in a different $x = \mathsf{L}_{\text{in}}(v, M)$. The second requirement intuitively guarantees that two elements $x, x' \in \mathbb{Z}_2^s$ either have the same or disjoint capacity sets. As becomes clear in the proof, this requirement can be relaxed at a security loss of factor 2^m. We notice that these requirements are easily satisfied, and standard injection functions L_{in} satisfy both. In particular, commonly used injection functions, e.g. functions that consist of XORing the message with and/or inserting it in a part of the state, clearly satisfy both properties. Note that the second requirement is satisfied for any linear transformation.
Requirement from L_{out}. We require that for any $(v, M) \in \mathbb{Z}_2^s \times \mathbb{Z}_2^m$, the function $\mathsf{L}_{\text{out}}(\cdot, v, M)$ is a bijection on the state. Its inverse function is denoted by $\mathsf{L}_{\text{out}}^{-1}[v, M]$.

3.2 Extraction Function g

On input of a state value v_{k+i-1}, the extraction function g employs an extracting transformation $\mathsf{L}_{\text{ex}} : \mathbb{Z}_2^s \to \mathbb{Z}_2^p$ that outputs a data block, and then permutes the state with π. Formally, the extraction function g is defined as $g(v_{k+i-1}) = (v_{k+i}, P_i)$, where $P_i \leftarrow \mathsf{L}_{\text{ex}}(v_{k+i-1})$ and $v_{k+i} \leftarrow \pi(v_{k+i-1})$. The function g is depicted in Fig. 2. Similar to f, one can consider an additional transformation after the call to the permutation, which may have v_{k+i-1} as extra input. This generalization would, however, make the proof considerably more complex (see Sect. 5).

Requirement from L_{ex}. We require L_{ex} to be balanced. Intuitively, this requirement means that each extract $P \in \mathbb{Z}_2^p$ is equally likely to occur. Accordingly, the function $\mathsf{L}_{\text{ex}}^{-1}$ is defined as described in Sect. 2, i.e. $\mathsf{L}_{\text{ex}}^{-1}(P) = \{v \in \mathbb{Z}_2^s \mid \mathsf{L}_{\text{ex}}(v) = P\}$.

3.3 Finalization Function fin

The function fin combines the l bit strings, obtained from squeezing the state, into the message digest. In most of the existing sponge-based designs, the finalization function simply consists of concatenating a required number of blocks, $l = \lceil n/p \rceil$, and chopping it to the required length of n bits. Parazoa functions allow for a generalized finalization function.

Requirement from fin. We require fin to be balanced. Intuitively, this requirements means that each digest h is equally likely to occur. Accordingly, the function fin^{-1} is defined as described in Sect. 2, i.e. $\mathsf{fin}^{-1}(h) = \{(P_1, \ldots, P_l) \mid \mathsf{fin}(P_1, \ldots, P_l) = h\}$.

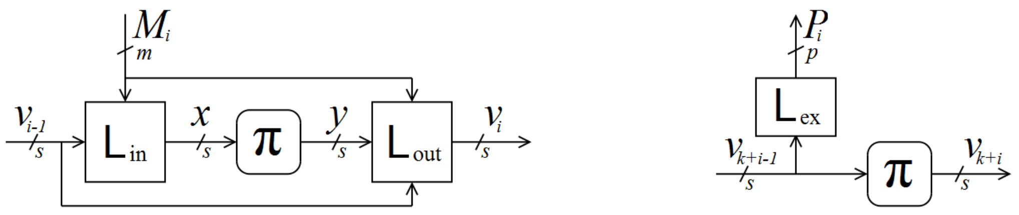

Fig. 2. The compression function f (left), and the extraction function g (right).

3.4 Padding Function pad

The padding function pad is an injective mapping that transforms messages of arbitrary length into messages of length an integral multiple of the block size m. Associated to pad is the function depad, that processes a message M' as follows: if $M' = \mathsf{pad}(M)$ for some message M, it outputs this M, otherwise it outputs $\perp$. Note that the output is unique as the padding function is injective.

Requirement from pad. We require pad to satisfy the following property: we either have $l = 1$, or the last block of a padded message, M_k, satisfies for any $x \in \mathbb{Z}_2^s$ and $(v', M') \in \mathbb{Z}_2^s \times \mathbb{Z}_2^m$:

$$\mathsf{L}_{\text{in}}(x, M_k) \neq x \text{ and } \mathsf{L}_{\text{in}}(\mathsf{L}_{\text{out}}(x, v', M'), M_k) \neq x. \tag{2}$$

As explained in Sect. 4.3 in more detail, this requirement comes from the fact that permutation queries to the simulator corresponding to the extraction phase (1c), may correspond to compression function executions f as well. We notice that for the original sponge design, condition (2) translates to requiring that the last block of a padded message is not a zero-block (which is exactly the requirement as posited by the authors of the sponge design in [8]). Because parazoa functions generalize these functions significantly, this requirement has become more complex accordingly.

4 Indifferentiability Analysis of Parazoa Functions

In this section, we prove the parazoa function of Sect. 3 indifferentiable from a random oracle, under the assumption that the underlying permutation π behaves like an ideal primitive. Intuitively, the proof consists of demonstrating that there exists a simulator such that no distinguisher can differentiate the real world $\mathcal{H}^\pi, \pi$ from the simulated world $\mathsf{RO}, \mathsf{S}^{\mathsf{RO}}$, except with negligible probability.

For the purpose of the proof, we introduce a technical variable d which we refer to as the *capacity loss*. Consider the set of all couples (v, x) such that $\mathsf{L}_{\text{in}}(v, M) = x$ for some M (M is uniquely determinable from v, x). We define $d \geq 0$ to be the minimal value such that

Criterion 1. For fixed x and fixed $P \in \mathbb{Z}_2^p$, there are at most 2^d couples (v, x) such that $v \in \mathsf{L}_{\text{ex}}^{-1}(P)$;
Criterion 2. For fixed v and fixed $P \in \mathbb{Z}_2^p$, there are at most 2^d couples (v, x) such that $x \in \mathsf{L}_{\text{ex}}^{-1}(P)$.

Notice that, as a consequence of the first criterion, we obtain $|\mathcal{C}(x)| \leq 2^{p+d}$ for any x, as any $v \in \mathbb{Z}_2^s$ satisfies $\mathsf{L}_{\text{ex}}(v) = P$ for exactly one $P \in \mathbb{Z}_2^p$. We note that the second criterion is not needed in case $l = 1$ (see Sect. 4.3). The reason why we opt for the name "capacity loss", as well as an intuition behind this parameter, is given in Sect. 4.1.

Theorem 1. *Let π be a random s-bit permutation, and let RO be a random oracle. Let $\mathcal{H}$ be a parazoa function parameterized by l, m, n, p, s, t. Let $\mathcal{D}$ be a distinguisher that makes at most q_1 left queries of maximal length $(K-1)m$ bits, where $K \geq 1$, q_2 right queries, and runs in time t. Then:*

$$\mathbf{Adv}_{\mathcal{H},\mathsf{S}}^{\mathsf{pro}}(\mathcal{D}) = O\left(\frac{((K+l)q_1 + q_2)^2}{2^{s-p-d}}\right), \tag{3}$$

where S makes at most $q_s \leq q_2$ queries to RO and runs in time $O(q_2^2)$.

We note that the bound is optimal for the generic parazoa design: for the original sponge design, as a particular instantiation of the parazoa functions, the best generic attack requires about $2^{(s-p-d)/2}$ queries [10] and meets the derived indifferentiability bound. In what remains of this section, an intuition behind the capacity loss d is given in Sect. 4.1, basic preliminary definitions for the description of our simulator are given in Sect. 4.2, and the simulator used in the proof is introduced and explained in more detail in Sect. 4.3 and Fig. 5. Then, Thm. 1 is formally proven in App. B.

4.1 On the Capacity Loss d

We will provide an intuition for the technical parameter d. It is used in the computation of the indifferentiability security bound: in some cases, the simulator has to generate a value v such that it is *not* a member of $\mathcal{C}(x)$, for one or more values of x, and moreover such that $\mathsf{L}_{\mathsf{ex}}(v) = P$ for some fixed P. Then, by the first criterion, the simulator can choose out of at least 2^{s-p-d} values for v. Intuitively, the value $s - p - d$ represents the number of bits of information of a state value that (1) cannot be affected by the distinguisher by ways of message injection, and (2) that cannot be obtained by the distinguisher by ways of extraction. We demonstrate this by ways of two examples.

- Example 1: the sponge hash function [10] (cf. Sect. 1). The functions L_{in} and L_{ex} of the sponge hash function design are given in Fig. 3. We start with the first criterion that defines parameter d. Fix any $x \in \mathbb{Z}_2^s$ and fix any $P \in \mathbb{Z}_2^p$, where $p = r$. We are considering the number of choices for v that satisfy $\mathsf{L}_{\mathsf{in}}(v, M) = x$ for some M, and that satisfy $\mathsf{L}_{\mathsf{ex}}(v) = P$. As becomes clear from Fig. 3, the first requirement uniquely fixes the last c bits of v, while the second statement fixes the first r bits of c. Altogether, the value v is uniquely determined by the fixed values x and P, and criterion 1 of the definition of d is satisfied for $d = 0$. Criterion 2 is equivalent, and we find that for sponge functions $d = 0$, hence the value attains its minimum;

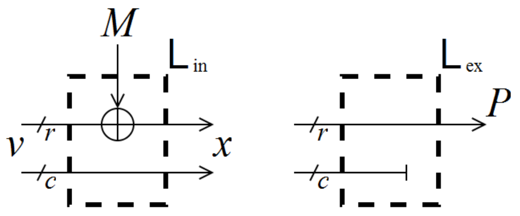

Fig. 3. The functions L_{in} and L_{ex} of the sponge hash function [10].

- Example 2: the sponge-like hash function of App. A. The functions L_{in} and L_{ex} of this hash function design are given in Fig. 4. Fix any $x \in \mathbb{Z}_2^s$ and fix any $P \in \mathbb{Z}_2^p$, where $p = c$. Again, we are considering the number of choices for v that satisfy $\mathsf{L}_{\mathsf{in}}(v, M) = x$ for some M, and that satisfy $\mathsf{L}_{\mathsf{ex}}(v) = P$. As can be observed from Fig. 4, if the rightmost c bits of x are equal to P, there are exactly 2^r choices for v that satisfy these requirement, namely $\{(M\|0^c) \oplus x \mid M \in \mathbb{Z}_2^r\}$. Again, criterion 2 is equivalent; we thus obtain $d = r = s - p$. In this example the distinguisher has "full control over the state": the first r bits can be freely adjusted by message injection, and the last c bits can be obtained (in most cases) by message extraction.

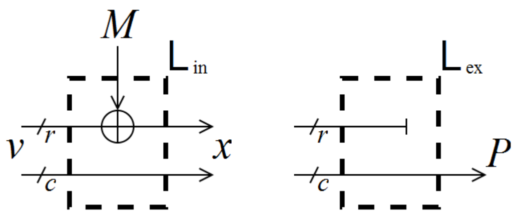

Fig. 4. The functions L_{in} and L_{ex} of the sponge-like hash function of App. A. For sake of explanation, we consider the state size to be $r + c$.

We note that the sponge-like hash function of App. A (example 2) still fits in the parazoa framework, and the indifferentiability result of Thm. 1 applies. Yet, as $d = s - p$ we obtain a trivial bound. The same occurs if we consider an example parazoa function where the message input size m equals the state size s: it turns out that $d = s - p$, and the indifferentiability result of Thm. 1 results in a trivial bound.

The value d varies between 0 and $s-p$ by construction, and ideally attains its minimum. In this case, the number of bits of information that cannot be controlled by the adversary is $s-p$. This value is exactly the capacity of the sponge hash function. For increasing d the value $s-p-d$ decreases, and therefore we call d the "capacity loss".

4.2 Defining the Simulator

The simulator maintains an initially empty database Sim-π that represents the simulated permutation. It consists of tuples $(x, y) \in \mathbb{Z}_2^s \times \mathbb{Z}_2^s$, where y denotes the sampled image of x under π. The simulator maintains a graph (V, E), which initially consists of the node IV and includes no edges. The edges in E are labeled by messages $M \in \mathbb{Z}_2^m$ and define input-output pairs of the compression function f: an edge $v \xrightarrow{M} w$ means that $f(v, M) = w$. Abusing notation, we denote by $v \xrightarrow{\bar{M}} w$ for $\bar{M} \in (\mathbb{Z}_2^m)^*$ that there is a path from v to w with the edges labeled by $\bar{M}$. By definition of L_{in}, one query pair (x, y) adds at most 2^{p+d} edges to the graph, namely the edges leaving from the vertices in $\mathcal{C}(x)$. By V_{out} we denote the set of nodes in V with an outgoing edge in (V, E). Notice that $V_{\mathrm{out}} = \bigcup_{x \in \mathsf{dom}(\text{Sim-}\pi)} \mathcal{C}(x)$. By $\tau(V)$ we denote the tree in (V, E) rooted in IV. Additionally, by $\bar{\tau}(V)$ we denote the subset of nodes of $\tau(V)$ that are labeled by a correctly padded message.
In addition, the simulator maintains a database Δ. This database will consist of future query inputs $x \in \mathbb{Z}_2^s$ of which the simulator knows that they correspond to the extraction phase of the parazoa execution (1c). Associated to each $x \in \Delta$ is a tuple $(i, P_{i+1} \cdots P_l)$ with $i \in \{1, \ldots, l-1\}$. Essentially, i denotes the number of executions of g that are already simulated for this specific path, and $P_{i+1}, \ldots, P_l$ denote the output values of the subsequent executions of g, determined by the simulator before. The idea behind Δ is further explained in Sect. 4.3.

4.3 Intuition

As is common in indifferentiability proofs, the simulator needs to be constructed in such a way that the answers from the oracles $(\mathcal{H}^\pi, \pi)$ and $(\mathsf{RO}, \mathsf{S}^{\mathsf{RO}})$ are close to identically distributed. In other words, the oracle answers made by the simulator need to be in consistency with the random oracle, in such a sense that any relation among the query answers in real world, holds in the simulated world as well. In particular, the simulator needs to pay attention to the following scenario: suppose a path $\mathsf{IV} \xrightarrow{\bar{M}} v_k$ is in the graph, for $\bar{M} \in \mathsf{rng}(\mathsf{pad})$ (i.e. $v_k \in \bar{\tau}(V)$). Suppose moreover that $v_k, \ldots, v_{k+l-1} \in \mathsf{dom}(\text{Sim-}\pi)$, where $P_i = \mathsf{L}_{\mathrm{ex}}(v_{k+i-1})$ and $v_{k+i} = \text{Sim-}\pi(v_{k+i-1})$ (for $i = 1, \ldots, l$). Then, these values $(P_1, \ldots, P_l)$ should satisfy $\mathsf{fin}(P_1, \ldots, P_l) = \mathsf{RO}(\mathsf{depad}(\bar{M}))$ in order for the simulator to maintain consistency. However, in general the simulator can only guarantee this equation to hold if the P_i's are decided *after* $\bar{M}$ is known, but *before* v_k is known (notice that v_k determines $P_1 = \mathsf{L}_{\mathrm{ex}}(v_k)$ deterministically). The simulator of Fig. 5 handles this problem in a smart way: in the query where the last edge (v_{k-1} to v_k) is added, the simulator decides on $(P_1, \ldots, P_l)$ on forehand, and based on these, he fixes the next state value v_k such that $v_k \in \mathsf{L}_{\mathrm{ex}}^{-1}(P_1)$. This value equals the input to the next execution of π in the chaining. It stores $(v_k; 1, P_2, \ldots, P_l)$ in its database Δ. As soon as the simulator is then queried v_k, the simulator will apply the same trick: the answer $v_{k+1} \leftarrow \text{Sim-}\pi(v_k)$ will be generated such that $v_{k+1} \in \mathsf{L}_{\mathrm{ex}}^{-1}(P_2)$. This value v_{k+1} equals the input to the next execution of π in the chaining. Subsequently it replaces the corresponding entry in Δ with $(v_{k+1}; 2; P_3, \ldots, P_l)$.
Before describing the simulator in more detail, we note that it includes several **GOTO**-statements. These statements guarantee the randomly generated values to satisfy certain properties. Associated to the simulator is a constant $C > 0$: the simulator aborts if a certain **GOTO**-statement is executed C times consecutively. Due to the inclusion of this parameter C, the simulator operates in polynomial time, rather than in expected polynomial time. In the security proof (App. B) this constant is fixed to a certain value.
The simulator will answer its queries such that the tree $\tau(V)$ grows as little as possible: indeed, any path in the tree may emerge in the need of an extra element in Δ, and eventually in an evaluatable query. However, as mentioned before, one query pair (x, y) defines at most 2^{p+d} edges in the graph,

```
Forward Query $\mathsf{S}(x)$

000 if $x \in \mathsf{dom}(\text{Sim-}\pi)$ :
001     return $y = \text{Sim-}\pi(x)$
002 if $x \in \varDelta$ assoc. with some $(i; P_{i+1} \cdots P_l)$ :
003     $v_{\text{nxt}} \xleftarrow{\$} \mathsf{L}_{\text{ex}}^{-1}(P_{i+1}) \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \varDelta\big)$
004     if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
005         GOTO 003
006     $y \leftarrow v_{\text{nxt}}$
007     if $y \in \mathsf{rng}(\text{Sim-}\pi)$ :  GOTO 003
008     if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
009         $\begin{cases} \text{find unique } v \in \tau(V) \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x \end{cases}$
010         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \big(\bigcup_{x \in \varDelta} \mathcal{C}(x)\big)$
011         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
012             GOTO 003
013     end if
014     $\varDelta \leftarrow \varDelta \backslash \{(x; i, P_{i+1} \cdots P_l)\}$
015     if $i + 1 < l$ :  $\varDelta \leftarrow \varDelta \cup \{(v_{\text{nxt}}; i+1, P_{i+2} \cdots P_l)\}$
016 else if $\mathcal{C}(x) \cap \tau(V) = \emptyset$ :
017     $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
018 else if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
019     $\begin{cases} \text{find unique } v \in \tau(V), \bar{M} \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x \text{ and } \mathsf{IV} \xrightarrow{\bar{M}} v \end{cases}$
020     if $\bar{M} \| M \notin \mathsf{rng}(\mathsf{pad})$ :
021         $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
022         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \big(\bigcup_{x \in \varDelta} \mathcal{C}(x)\big)$
023         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
024             GOTO 021
025     else if $\bar{M} \| M \in \mathsf{rng}(\mathsf{pad})$ :
026         $h \xleftarrow{\$} \mathsf{RO}(\mathsf{depad}(\bar{M} \| M))$
027         $(P_1, \ldots, P_l) \xleftarrow{\$} \mathsf{fin}^{-1}(h)$
028         $v_{\text{nxt}} \xleftarrow{\$} \mathsf{L}_{\text{ex}}^{-1}(P_1) \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \varDelta\big)$
029         if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
030             GOTO 028
031         $y \leftarrow \mathsf{L}_{\text{out}}^{-1}[v, M](v_{\text{nxt}})$
032         if $y \in \mathsf{rng}(\text{Sim-}\pi)$ :  GOTO 028
033         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \big(\bigcup_{x \in \varDelta} \mathcal{C}(x)\big)$
034         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
035             GOTO 028
036         if $1 < l$ :  $\varDelta \leftarrow \varDelta \cup \{(v_{\text{nxt}}; 1, P_2 \cdots P_l)\}$
037     end if
038 end if
039 return $\text{Sim-}\pi(x) \leftarrow y$
```

```
Inverse Query $\mathsf{S}^{-1}(y)$

100 if $y \in \mathsf{rng}(\text{Sim-}\pi)$ :
101     return $x = \text{Sim-}\pi^{-1}(y)$
102 $x \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{dom}(\text{Sim-}\pi)$
103 if $x \in \varDelta$ or $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
104     GOTO 102
105 return $\text{Sim-}\pi^{-1}(y) \leftarrow x$
```

Fig. 5. The simulator S for π used in the proof of Thm. 1. The simulator aborts if a certain **GOTO**-statement is executed $C > 0$ times consecutively, for some constant C.

leaving from the nodes in the set $\mathcal{C}(x)$. The simulator will answer its queries so as to satisfy the following properties concerning the growth of the graph:

(a) Out of all newly added edges, at most one will be added to the tree. More generally, the simulator assures the following property at any time in the execution:

$$|\mathcal{C}(x) \cap \tau(V)| \leq 1 \text{ for any } x \in \mathbb{Z}_2^s; \tag{4}$$

(b) When a query adds a new edge to the tree, its end node has no outgoing edge. Together with (a), this implies that per query at most one edge is added to the tree;
(c) The tree does not contain any colliding paths.

Notice that a query pair (x, y) adds a new edge to the tree *if and only if* $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$. Indeed, $\mathcal{C}(x)$ corresponds to all nodes with an outgoing edge defined by the query pair (x, y). In this case, the simulator needs to assure that properties (a)-(c) are satisfied. Secondly if $x \in \Delta$, the simulator needs to handle as described above. Ideally, the value x satisfies $x \notin \Delta$ and $\mathcal{C}(x) \cap \tau(V) = \emptyset$, which explains the algorithm for S^{-1}. In forward queries to S, however, x is chosen by the distinguisher, and it may be possible that $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ or $x \in \Delta$. We will now explain the algorithm for a forward query x to S, based on the above observations.
In case $\mathcal{C}(x) \cap \tau(V) = \emptyset$ (lines 016-017), no edge will be added to the tree, and (a)-(c) are trivially satisfied. In case $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$, by (4) and the definition of L_{in}, there exists one unique couple $v \in \tau(V)$ and $M \in \mathbb{Z}_2^m$ such that $\mathsf{L}_{\text{in}}(v, M) = x$. By construction, the current query adds the edge $v \xrightarrow{M} w$ to the tree, where $w = \mathsf{L}_{\text{out}}(y, v, M)$. In lines 022 and 033, the simulator assures that this is the only edge added to the tree, i.e. that y is chosen such that $w \notin V_{\text{out}}$ (there is no outgoing edge from w) and $w \notin \mathcal{C}(x)$ (no outgoing edge from w will accidentally be added in the current round). Additionally, requirements (a) and (c) are covered by requiring that w does not share a capacity set with a node already in $\tau(V)$: lines 023 and 034.
Hence, a query adds at most one edge to the tree, and in particular, in case of evaluatable queries, the last edge $v_{k-1} \xrightarrow{M} v_k$ is really added at last. We still need to consider this specific case that $\bar{\tau}(V)$ is increased. Additionally, we still need to explain the case of $x \in \Delta$.

$\bar{\tau}(V)$ **gets increased but** $x \notin \Delta$**.** This specific case corresponds to the **else**-clause of line 025: the simulator will proceed as previously described: the next state value v_{nxt} (which equals the first permutation input of the extraction phase) is generated, and the original answer y is generated accordingly in line 031. Notice that we need to assure that no collision in Sim-π occurs (line 032). In line 036, the node v_{nxt} is added to Δ, provided $1 < l$. Note that, by (c), the path to v_k is unique, and $(P_1, \ldots, P_l)$ is generated in a non-ambiguous way;

$\bar{\tau}(V)$ **does not get increased and** $x \in \Delta$**.** This case corresponds to the **if**-clause of line 002: again, the simulator will generate the next state value v_{nxt} (which equals the next permutation input of the extraction phase), and generate the original answer y accordingly (line 006). It will update Δ in lines 014-015. Note that in this if-clause it may still be possible that $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$. Then, the simulator assures properties (a)-(c) as mentioned before (by lines 010 and 011);

$\bar{\tau}(V)$ **gets increased and** $x \in \Delta$**.** It may be the case that a query x to S is an element of Δ, and moreover adds an edge to the tree (this is checked in the **if**-clause of line 008). However, as we will now explain, the message block that labels this edge can never be the last block of a padded message, and hence, $\bar{\tau}(V)$ will not be increased.
As $x \in \Delta$, this specific value is fixed by the simulator on forehand (in the previous query of the extraction phase defined as v_{nxt}). In particular, it is generated in the query where $(x', y') \in$ Sim-π is generated such that either $\mathsf{L}_{\text{out}}(y', v', M') = x$ for some v', M' (line 031, or such that $y' = x$ (line 006). However, x had been generated such that $\mathcal{C}(x)$, all start-nodes of the edges defined by x, had an empty intersection with $\tau(V)$ (lines 004 and 029)[2]. Also, in all future queries, it is assured

[2] Notice that, if $l = 1$, one does not need to assure this property. In particular, line 004 and 029 become unnecessary. In the proof, these are the lines that make us require the second property concerning the capacity loss d (cf. Sect. 4).

that new queries cannot make the link to this specific x (due to "$\bigcup_{x\in\Delta}\mathcal{C}(x)$" in lines 010, 022 and 033). As a consequence, if the query (x, y) adds an edge to the tree, this only happens for an edge leaving from the end node of an edge defined by (x', y') (as this is the only query round in which $\mathcal{C}(x)$ is not avoided as end node of a newly added edge to the tree). In other words, after the forward query x to S, we have the path $\mathsf{IV} \xrightarrow{\bar{M}} v_0 \xrightarrow{M_1} v_1 \xrightarrow{M_2} v_2$ in the tree for $\bar{M} \in (\mathbb{Z}_2^m)^*$ and $M_1, M_2 \in \mathbb{Z}_2^m$, where the edge (v_0, v_1) is defined by (x', y') and the edge (v_1, v_2) by (x, y). However, the value M_2 satisfying this path, particularly satisfies $x = \mathsf{L}_{\text{in}}(v_1, M_2)$, where $v_1 = \mathsf{L}_{\text{out}}(y', v', M')$. Given the specific property of x (in the beginning of this paragraph), it either satisfies

$$x = \mathsf{L}_{\text{in}}(x, M_2), \text{ or } x = \mathsf{L}_{\text{in}}(\mathsf{L}_{\text{out}}(x, v', M'), M_2),$$

for some $(v', M') \in \mathbb{Z}_2^s \times \mathbb{Z}_2^m$. By the requirement in Sect. 3.4, we either have $l = 1$ (which means that $\Delta = \emptyset$ at all time), or that M_2 can never be the last block of a padded message. Summarizing, a query $x \in \Delta$ *never* increases $\bar{\tau}(V)$.

The full proof of Thm. 1 is given in App. B.

5 Concluding Remarks

We now present some comments on the parazoa hash function design.

Ambiguity. The design as described in Sect. 3 allows for ambiguous interpretations. In particular, it is straightforward to construct schemes that can be described as a parazoa function in different ways:

- Let $\mathcal{P}$ be any s-bit permutation. For any parazoa design with $l = 1$, the same design is described if $\mathsf{L}_{\text{in}}, \mathsf{L}_{\text{out}}$ and L_{ex} are replaced by $\mathsf{L}'_{\text{in}} = \mathsf{L}_{\text{in}} \circ \mathcal{P}^{-1}$, $\mathsf{L}'_{\text{out}} = \mathcal{P} \circ \mathsf{L}_{\text{out}}$ and $\mathsf{L}'_{\text{ex}} = \mathsf{L}_{\text{ex}} \circ \mathcal{P}^{-1}$, and with the initial chaining value re-defined as $\mathsf{IV}' = \mathcal{P}(\mathsf{IV})$. Although this modification does not harm the security of the described parazoa design, the obtained indifferentiability bound may differ. In particular, this modification may affect the value d (cf. Sect. 4);
- Consider a parazoa design where $\mathsf{L}_{\text{ex}}(v) = \mathsf{chop}_{s-p}(v)$, and $\mathsf{fin}(P_1, \ldots, P_l) = P_1\| \cdots \|P_l$. Then, the same design is described if these functions are replaced by $\mathsf{L}'_{\text{ex}}(v) = v$ and $\mathsf{fin}'(P_1, \ldots, P_l) = \mathsf{chop}_{s-p}(P_1)\| \cdots \|\mathsf{chop}_{s-p}(P_l)$. However, our proof fails for the second description, whereas this need not be the case for the first description. This paradoxical ambiguity is because the parazoa design allows for *any* type of finalization function, and therefore, we cannot base security of the parazoa function on specific properties of the finalization. For instance, our scheme does not make any distinction between fin' and $\mathsf{fin}''(P_1, \ldots, P_l) = \mathsf{chop}_{sl-pl}(P_1\| \cdots \|P_l)$.

In general, different descriptions of a parazoa design may result in different bound, and the best bound naturally applies.

Generalization of g. It is possible to consider the parazoa hash function design with a more complicated function g, namely to define it as $g(v_{k+i-1}) = (v_{k+i}, P_i)$, where $(x, P_i) \leftarrow \mathsf{M}_{\text{in}}(v_{k+i-1})$, $y \leftarrow \pi(x)$ and $v_{k+i} \leftarrow \mathsf{M}_{\text{out}}(y, v_{k+i-1})$ (cf. Fig. 6). It is straightforward to generalize the simulator and the proof to this case. The simulator of Fig. 5 is modified mainly in the lines 003 and 028 (one randomly generates x_{nxt} as input to the next permutation and generates v_{nxt} randomly from $\mathsf{M}_{\text{in}}^{-1}(x_{\text{nxt}}, P_{i+1})$ rather than $\mathsf{L}_{\text{ex}}^{-1}(P_{i+1})$) and in line 006 (one deterministically finds the previous state value v_{prev} and computes y such that $v_{\text{nxt}} = \mathsf{M}_{\text{out}}(y, v_{\text{prev}})$). The proof results in the same bound. For this proof, we would require both M_{in} and M_{out} to be bijections on the state, and additionally that, restricted to the extract P_i, the function M_{in} is balanced.

Generalization to (multiple) different ideal primitives. The description of the parazoa design is based on one permutation π, which is utilized by both f and g. We notice that in the design and the proof, π can easily be replaced by a random function. Also, it is straightforward to consider two different permutations π_1, π_2 (e.g., f is build on π_1 and g on π_2). We notice that, in some cases, the usage of

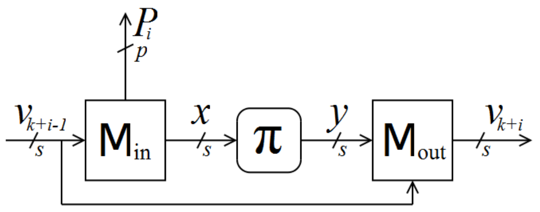

Fig. 6. A generalized variant of the extraction function g.

different permutations may improve the indifferentiability bound. For instance, if $l = 1$, and the last execution of f is defined to use a different permutation, the variable d (cf. Sect. 4) becomes superfluous.

Remarks on the applications. In Sect. 1.1, we sketched several applications of parazoa functions, and we will briefly elaborate on this. The original **Sponge** follows the parazoa design with $m = p$; if we relax this requirement to allow for $p \neq m$, the indifferentiability bound results in $O((Kq)^2/2^{s-\max\{p,m\}})$ ($d = \max\{m - p, 0\}$), where s is the state size of the sponge. The indifferentiability bounds for the lightweight hash functions **Quark**, **PHOTON** and **SPONGENT** follow directly [3, 14, 19]. The **Grindahl** hash function (simplified, with zero final blank rounds) satisfies the parazoa function design with $d = m$, which is caused by the fact that L_{in} inserts the message in the leftmost part, while L_{ex} outputs the rightmost part of the state. With respect to NIST's SHA-3 hash function competition, we can consider the following second round candidates. **CubeHash** consists of a permutation P executed 16 times iteratively, and the result holds if P^{16} is assumed to be a random permutation. In its final transformation, one bit in the chaining is swapped, which results in $d = 1$. **Hamsi** [23] employs two different permutations, one of which is exclusively used in the last compression function. We note that, even though the permutation and compression function input sizes of Hamsi differ, the design can be described as a parazoa function: instead of chopping half of the state at the end of a compression function evaluation f, and concatenating the remaining state with the expanded message in the subsequent call to f, one can just as well leave the state unchopped, and overwrite the redundant part of the state with the expanded message in the next evaluation of f. A simplified version of Hamsi, where each compression function employs *one* single permutation, would be insecure as the attack of App. A would apply. However, it is fair to believe that for the original Hamsi a better bound can be obtained (see previous paragraph). The **Fugue** design can be considered to have a final compression function execution based on a permutation π' differing from the permutation π used in the iteration (for instance, for Fugue-256 we have $\pi = (\textbf{SMIX} \circ \textbf{CMIX} \circ \textbf{ROR3})^2$ and $\pi' = \textbf{G}$ [20]). Still, the parazoa design can handle Fugue, and the results carry over. The **Luffa** compression function consists of a linear function operating on the state, and multiple smaller permutations executed in parallel. The results hold under the assumption that this parallel execution behaves like *one* random permutation.[3] For both Fugue and Luffa, our result has been confirmed by Bhattacharyya and Mandal [11].

Acknowledgments. This work has been funded in part by the IAP Program P6/26 BCRYPT of the Belgian State (Belgian Science Policy), in part by the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II, and in part by the Research Council K.U.Leuven: GOA TENSE. The first author was supported by a Ph.D. Fellowship from the Flemish Research Foundation (FWO-Vlaanderen). The second author is supported by a Ph.D. Fellowship from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

[1] Andreeva, E., Mennink, B., Preneel, B.: Security reductions of the second round SHA-3 candidates. In: Burmester, M., Tsudik, G., Magliveras, S., Ilic, I. (eds.) ISC 2010: 13th International Conference on Information Security.

[3] For Luffa-384 and Luffa-512, which satisfy the parazoa design with $l = 2$, equation (2) says that the last block of a padded message should satisfy $\mathsf{L}_{\text{in}}(x, M) \neq x$ for all $x \in \mathbb{Z}_2^s$. However, the last block of a padded message of Luffa-384 or Luffa-512 is a zero-block, and for both versions of Luffa we have $\mathsf{L}_{\text{in}}(0^s, 0^m) = 0^s$. It is straightforward to see that indifferentiability can be re-obtained by artificially assuring that $\tau(V) \cap \mathcal{C}(0^s) = \emptyset$ throughout (for instance, this is satisfied if the simulator defines a query pair $(0^s, y)$ for randomly chosen $y \in \mathbb{Z}_2^s$ in its initialization phase).

Lecture Notes in Computer Science, vol. 6531, pp. 39–53. Springer (Oct 2010)

[2] Andreeva, E., Neven, G., Preneel, B., Shrimpton, T.: Seven-property-preserving iterated hashing: ROX. In: Kurosawa, K. (ed.) Advances in Cryptology – ASIACRYPT 2007. Lecture Notes in Computer Science, vol. 4833, pp. 130–146. Springer (Dec 2007)

[3] Aumasson, J.P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: A lightweight hash. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2010. Lecture Notes in Computer Science, vol. 6225, pp. 1–15. Springer, Santa Barbara, California, USA (Aug 17–20, 2010)

[4] Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) Advances in Cryptology – CRYPTO'96. Lecture Notes in Computer Science, vol. 1109, pp. 1–15. Springer (Aug 18–22, 1996)

[5] Bellare, M., Ristenpart, T.: Multi-property-preserving hash domain extension and the EMD transform. In: Lai, X., Chen, K. (eds.) Advances in Cryptology – ASIACRYPT 2006. Lecture Notes in Computer Science, vol. 4284, pp. 299–314. Springer (Dec 2006)

[6] Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Ashby, V. (ed.) ACM CCS 93: 1st Conference on Computer and Communications Security. pp. 62–73. ACM Press (Nov 1993)

[7] Bernstein, D.: CubeHash specification (2009), submission to NIST's SHA-3 competition

[8] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Smart, N.P. (ed.) Advances in Cryptology – EUROCRYPT 2008. Lecture Notes in Computer Science, vol. 4965, pp. 181–197. Springer (Apr 2008)

[9] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The KECCAK sponge function family (2009), submission to NIST's SHA-3 competition

[10] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Sponge functions (ECRYPT Hash Function Workshop 2007)

[11] Bhattacharyya, R., Mandal, A.: On the indifferentiability of Fugue and Luffa. In: Lopez, J., Tsudik, G. (eds.) ACNS 11: 9th International Conference on Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 6715, pp. 479–497. Springer, Nerja, Spain (Jun 7–10, 2011)

[12] Bhattacharyya, R., Mandal, A., Nandi, M.: Security analysis of the mode of JH hash function. In: Hong, S., Iwata, T. (eds.) Fast Software Encryption – FSE 2010. Lecture Notes in Computer Science, vol. 6147, pp. 168–191. Springer (Feb 2010)

[13] Biham, E., Dunkelman, O.: A framework for iterative hash functions – HAIFA. Cryptology ePrint Archive, Report 2007/278 (2007)

[14] Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: SPONGENT: A lightweight hash function. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2011. Lecture Notes in Computer Science, vol. 6917, pp. 312–325. Springer, Nara, Japan (Sep 28 – Oct 1, 2011)

[15] Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård revisited: How to construct a hash function. In: Shoup, V. (ed.) Advances in Cryptology – CRYPTO 2005. Lecture Notes in Computer Science, vol. 3621, pp. 430–448. Springer (Aug 14–18, 2005)

[16] Damgård, I.: A design principle for hash functions. In: Brassard, G. (ed.) Advances in Cryptology – CRYPTO'89. Lecture Notes in Computer Science, vol. 435, pp. 416–427. Springer (Aug 20–24, 1990)

[17] De Cannière, C., Sato, H., Watanabe, D.: Hash Function Luffa (2009), submission to NIST's SHA-3 competition

[18] Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., Rabin, T.: Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In: Franklin, M. (ed.) Advances in Cryptology – CRYPTO 2004. Lecture Notes in Computer Science, vol. 3152, pp. 494–510. Springer (Aug 15–19, 2004)

[19] Guo, J., Peyrin, T., Poschmann, A.: The PHOTON family of lightweight hash functions. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 222–239. Springer (Aug 14–18, 2011)

[20] Halevi, S., Hall, W., Jutla, C.: The Hash Function "Fugue" (2009), submission to NIST's SHA-3 competition

[21] Halevi, S., Hall, W., Jutla, C., Roy, A.: Weak ideal functionalities for designing random oracles with applications to Fugue (2010)

[22] Knudsen, L.R., Rechberger, C., Thomsen, S.S.: The Grindahl hash functions. In: Biryukov, A. (ed.) Fast Software Encryption – FSE 2007. Lecture Notes in Computer Science, vol. 4593, pp. 39–57. Springer (Mar 2007)

[23] Küçük, Ö.: The Hash Function Hamsi (2009), submission to NIST's SHA-3 competition

[24] Lai, X., Massey, J.L.: Hash function based on block ciphers. In: Rueppel, R.A. (ed.) Advances in Cryptology – EUROCRYPT'92. Lecture Notes in Computer Science, vol. 658, pp. 55–70. Springer (May 1992)

[25] Maurer, U.M., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: Naor, M. (ed.) TCC 2004: 1st Theory of Cryptography Conference. Lecture Notes in Computer Science, vol. 2951, pp. 21–39. Springer (Feb 2004)

[26] Merkle, R.C.: One way hash functions and DES. In: Brassard, G. (ed.) Advances in Cryptology – CRYPTO'89. Lecture Notes in Computer Science, vol. 435, pp. 428–446. Springer (Aug 20–24, 1990)

[27] National Institute for Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA3) Family (nov 2007)

[28] Stam, M.: Blockcipher-based hashing revisited. In: Dunkelman, O. (ed.) Fast Software Encryption – FSE 2009. Lecture Notes in Computer Science, vol. 5665, pp. 67–83. Springer (Feb 2009)

[29] Wikipedia: Parazoa (jan 2012), `http://en.wikipedia.org/wiki/Parazoa`

[30] Wu, H.: The Hash Function JH (2009), submission to NIST's SHA-3 competition

A Differentiability of a “Sponge-Like” Function

In this appendix we show that a simple modification of the original sponge design can render it insecure with respect to indifferentiability. To this end, we construct the following sponge-like design.
Consider the following hash function $\mathcal{H} : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$, that has a state size $2n$, and processes message blocks of n bits (see also Fig. 7). It is based on a $2n$-bit permutation π, and uses a simple injective padding function pad to process messages of arbitrary length: $\mathsf{pad}(M) = M\|1\|0^{-|M|-1 \bmod n}$, parsed into message blocks of n bits. For an initial value $\mathsf{IV}_1\|\mathsf{IV}_2$, the hash function $\mathcal{H}$ processes a message M as follows: $\mathcal{H}(M)$ outputs the n rightmost bits of h_k, where $(M_1, \ldots, M_k) \leftarrow \mathsf{pad}(M)$, $h_0 \leftarrow \mathsf{IV}_1\|\mathsf{IV}_2$, and $h_i \leftarrow \pi(h_{i-1} \oplus (M\|0^n))$ for $i = 1, \ldots, k$. Notice that this design follows the original sponge design, with a small modification that the message digest is defined by the other half of the state (but we stress that this observation does not invalidate the security of the sponge function design). We construct a distinguisher $\mathcal{D}$ that can distinguish $(\mathcal{H}^\pi, \pi)$ from $(\mathsf{RO}, \mathsf{S}^{\mathsf{RO}})$, for any simulator S.

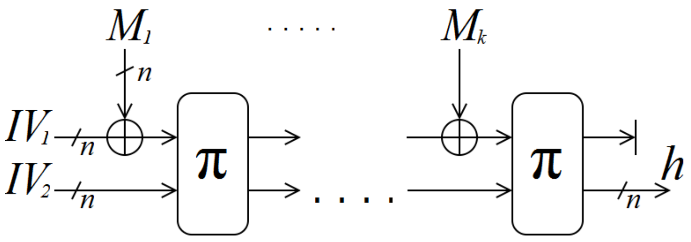

Fig. 7. The “sponge-like” hash function $\mathcal{H}$ described in App. A.

- First, $\mathcal{D}$ decides on an arbitrary message M of length $0 < |M| < n$. It defines $M_1 = \mathsf{pad}(M)$ and $M_3 = 1\|0^{n-1}$. Notice that $M_1\|M_2\|M_3 = \mathsf{pad}(M_1\|M_2)$ for any $M_2 \in \mathbb{Z}_2^n$;
- $\mathcal{D}$ queries M to the left oracle, to obtain $h_1 \leftarrow L(M)$;
- $\mathcal{D}$ queries $0\|h_1$ to the right oracle, to obtain $x_1\|y_1 \leftarrow R(0\|h_1)$;
- $\mathcal{D}$ queries $(x_1 \oplus M_3)\|y_1$ to the right oracle, to obtain $x_2\|y_2 \leftarrow R((x_1 \oplus M_3)\|y_1)$;
- $\mathcal{D}$ queries $(\mathsf{IV}_1 \oplus M_1)\|\mathsf{IV}_2$ to the right oracle, to obtain $x_3\|y_3 \leftarrow R((\mathsf{IV}_1 \oplus M_1)\|\mathsf{IV}_2)$;
- $\mathcal{D}$ queries $M_1\|x_3$ to the left oracle, to obtain $h_2 \leftarrow L(M_1\|x_3)$.

In the real world, where the distinguisher queries $\mathcal{H}$ and π, the answers of the oracles satisfy $h_1 = y_3$ and $h_2 = y_2$ by construction. In the simulated world however, h_2 equals $\mathsf{RO}(M_1\|x_3)$. As the simulator generated y_2 without any knowledge of M_1, equality $h_2 = y_2$ holds with negligible probability only. We notice that, even though the described hash function is a parazoa design, the counterexample does not contradict Thm. 1. In particular, Thm. 1 results in a $O(1)$ indifferentiability bound (as the design has $d = n$).

B Proof of Thm. 1

Let S be the simulator of Fig. 5, and let $\mathcal{D}$ be any distinguisher that makes at most q_1 left queries of maximal length $(K-1)m$ bits, where $K \geq 1$, and q_2 right queries. Recall from Def. 1 that the goal is to bound:

$$\mathbf{Adv}^{\mathsf{pro}}_{\mathcal{H},\mathsf{S}}(\mathcal{D}) = \left|\mathbf{Pr}\left(\mathcal{D}^{\mathcal{H}^\pi,\pi} = 1\right) - \mathbf{Pr}\left(\mathcal{D}^{\mathsf{RO},\mathsf{S}^{\mathsf{RO}}} = 1\right)\right|. \tag{5}$$

Thm. 1 will be proven via a game-playing argument. In this aspect, our result is fundamentally different from the result obtained by Bertoni et al. for the original sponge design [8]. Each game consists of a left and a right oracle. In the proof, G_1 will equal the simulated world, and G_9 the real world. We will go from game 1 to game 9 stepwise, and obtain a bound on (5) using a hybrid argument. We define $C = 1$, hence the simulator of Fig. 5 aborts if a certain **GOTO**-statement is executed.

Game 1: $G_1 = (L_1, R_1^{L_1})$ **(Fig. 8).** The left oracle L_1 of game 1 is defined to be a lazily-sampled random oracle, and the right oracle R_1 consists of the two interfaces defined by the simulator of Fig. 5, with an additional difference that a failure condition **bad** is added to the **GOTO**-statements in lines 005, 007, 012, 024, 030, 032, 035 and 104. The distinguisher does not see the difference until the adversary in game 1 sets **bad** (recall that we put $C = 1$). We obtain $\left|\mathbf{Pr}\left(\mathcal{D}^{RO,\mathsf{S}^{RO}} = 1\right) - \mathbf{Pr}\left(\mathcal{D}^{G_1} = 1\right)\right| \leq \mathbf{Pr}\left(\mathcal{D}^{G_1} \text{ sets } \mathbf{bad}\right)$.
Game 2: $G_2 = (L_2^{L_1}, R_1^{L_1})$ **(Fig. 8).** The left oracle of game 1 is replaced by a so-called relay oracle L_2 that passes the queries made by the distinguisher to L_1, and returns its responses. The right oracle remains unchanged. The distinguisher has identical views in G_1 and G_2, and we obtain $\mathbf{Pr}\left(\mathcal{D}^{G_1} = 1\right) = \mathbf{Pr}\left(\mathcal{D}^{G_2} = 1\right)$.
Game 3: $G_3 = (L_3^{R_1^{L_1}}, R_1^{L_1})$ **(Fig. 8).** The left oracle of game 2 is now replaced by an implementation of the parazoa function, which moreover uses the right oracle as a subroutine, rather than L_1 directly. The right oracle itself remains unchanged. In Prop. 1 it is proven that, as long as the **bad** flag is not set in any of the two games, both are identical. Formally, we obtain $\left|\mathbf{Pr}\left(\mathcal{D}^{G_2} = 1\right) - \mathbf{Pr}\left(\mathcal{D}^{G_3} = 1\right)\right| \leq \mathbf{Pr}\left(\mathcal{D}^{G_2} \text{ sets } \mathbf{bad}\right) + \mathbf{Pr}\left(\mathcal{D}^{G_3} \text{ sets } \mathbf{bad}\right)$.
Note that in game 3, as well as in all subsequent games, the right oracle will be queried at most $r := (K + l)q_1 + q_2$ times. Indeed, in all of the following games, the left oracle queries the right oracle at most K times in the compressing phase, and l times in the extraction phase. All subsequent right oracles are constructed in such a way that each query to this oracle adds at most 1 element to Sim-π.
Game 4: $G_4 = (L_3^{R_2}, R_2)$**.** The right oracle R_2 of game 4 differs from R_1 of game 3 in the sense that $h \xleftarrow{\$} L_1(\mathsf{depad}(\bar{M}\|M))$; $(P_1, \ldots, P_l) \xleftarrow{\$} \mathsf{fin}^{-1}(h)$ (lines 026 and 027) is replaced by $(P_1, \ldots, P_l) \xleftarrow{\$} \mathbb{Z}_2^{pl}$. Observe that the games are perfectly indistinguishable: in game 3, R_1 is the only algorithm querying L_1, and as the padding is injective, he never queries L_1 twice on the same value. Therefore, he can just as well generate the random values h himself. Then, as the function fin is balanced, we have, for any $\alpha \in \mathbb{Z}_2^{pl}$: $\mathbf{Pr}\left((P_1, \ldots, P_l) = \alpha : h \xleftarrow{\$} \mathbb{Z}_2^n, (P_1, \ldots, P_l) \xleftarrow{\$} \mathsf{fin}^{-1}(h)\right) = 1/2^{pl}$. In other words, the values $(P_1, \ldots, P_l)$ follow the uniform random distribution on pl bits, and therefore the right oracle can just generate them directly. Formally, we have $\mathbf{Pr}\left(\mathcal{D}^{G_3} = 1\right) = \mathbf{Pr}\left(\mathcal{D}^{G_4} = 1\right)$.
Game 5: $G_5 = (L_3^{R_3}, R_3)$ **(Fig. 9).** In game 4, all values $(P_1, \ldots, P_l)$ are randomly generated as soon as the first one is needed. The remaining $l - 1$ values are then associated to a node $x \in \varDelta$ (line 036), and as soon as x is queried, the next value, P_{i+1}, is taken off of the list and processed. In game 5, these values P_i are not anymore generated in advance, but generated when needed. As a consequence, we implicitly adjust the definition of $\varDelta$, in the sense that each element is labeled by an index $i \in \{1, \ldots, l-1\}$ only. As in both cases the values $(P_1, \ldots, P_l)$ are generated uniformly at random, a distinguisher cannot see the difference. Consequently, we obtain $\mathbf{Pr}\left(\mathcal{D}^{G_4} = 1\right) = \mathbf{Pr}\left(\mathcal{D}^{G_5} = 1\right)$.
Game 6: $G_6 = (L_3^{R_4}, R_4)$ **(Fig. 9).** In game 5, the values P_i ($i = 1, \ldots, l$) are taken uniformly at random, and the state values v_{nxt} are taken according to the property that $P_i = \mathsf{L}_{\text{ex}}(v_{\text{nxt}})$ for all $i = 1, \ldots, l$ (lines 004 and 028 in Fig. 9). In game 6, this is the other way around: v_{nxt} (recall that this value equals the input to the next permutation execution in the extraction phase) is taken randomly (permutation-wise), and the value P_i is taken such that $P_i = \mathsf{L}_{\text{ex}}(v_{\text{nxt}})$ still holds. Hence, the only changes are in lines 003-004 and 027-028 of game 5. In Prop. 2 it is proven that $\left|\mathbf{Pr}\left(\mathcal{D}^{G_5} = 1 \mid \mathcal{D}^{G_5} \text{ sets } \neg\mathbf{bad}\right) - \mathbf{Pr}\left(\mathcal{D}^{G_6} = 1 \mid \mathcal{D}^{G_6} \text{ sets } \neg\mathbf{bad}\right)\right| \leq \frac{2r^2}{2^s}$.
Game 7: $G_7 = (L_3^{R_5}, R_5)$ **(Fig. 10).** In game 6, concretely in the blocks 003-008, where the oracle is queried on an x belonging to the extraction phase, and 027-032 where the query answer to x will initiate the extraction phase, the oracle decides on its answer y based on the next state value v_{nxt}. In game 7, the answer y is taken uniformly at random, and the next state v_{nxt} is generated accordingly. The values $P_i = \mathsf{L}_{\text{ex}}(x_i)$ are not used in R_5, and their generation is omitted. In Prop. 3 it is proven that $\left|\mathbf{Pr}\left(\mathcal{D}^{G_6} = 1 \mid \mathcal{D}^{G_6} \text{ sets } \neg\mathbf{bad}\right) - \mathbf{Pr}\left(\mathcal{D}^{G_7} = 1 \mid \mathcal{D}^{G_7} \text{ sets } \neg\mathbf{bad}\right)\right| \leq \frac{2r^2}{2^s}$.
Game 8: $G_8 = (L_3^{R_6}, R_6)$ **(Fig. 10).** The right oracle R_6 in game 8 differs from R_5 in game 7 in

the sense that the **GOTO**-statements that are accompanied with a **bad**-statement are removed. As a consequence, game 7 and 8 proceed identically as long as the **bad** flag is not set in game 7. Formally, we obtain $\left|\mathbf{Pr}\left(\mathcal{D}^{G_7}=1\right)-\mathbf{Pr}\left(\mathcal{D}^{G_8}=1\right)\right| \leq \mathbf{Pr}\left(\mathcal{D}^{G_7} \text{ sets } \mathbf{bad}\right)$.

Game 9: $G_9 = (L_3^{R_7}, R_7)$**.** The right oracle R_7 mimics a lazily-sampled random permutation π, and the left oracle is the parazoa specification querying this right oracle. Hence, $G_9 = (\mathcal{H}^\pi, \pi)$, and thus $\mathbf{Pr}\left(\mathcal{D}^{G_9}=1\right) = \mathbf{Pr}\left(\mathcal{D}^{\mathcal{H}^\pi,\pi}=1\right)$. It turns out that R_6 of game 8 also mimics a lazily-sampled permutation, due to the removal of the **GOTO**-statements. In particular, any forward query to R_6 is answered with a $y \in \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$. As a consequence, we obtain $\mathbf{Pr}\left(\mathcal{D}^{G_8}=1\right) = \mathbf{Pr}\left(\mathcal{D}^{G_9}=1\right)$.

We conclude that (5) reduces to:

$$\mathbf{Adv}_{\mathcal{H},\mathsf{S}}^{\mathsf{pro}}(\mathcal{D}) \leq \sum_{i=1,2,3,5,6,7} \mathbf{Pr}\left(\mathcal{D}^{G_i} \text{ sets } \mathbf{bad}\right) + \frac{4r^2}{2^s}. \tag{6}$$

In the remainder of the proof, we will evaluate the probability that the distinguisher sets bad in game 3. Thereafter, we will elaborate on the other probabilities.

Consider the j^{th} query ($j = 1, \ldots, r$) to R_1. Notice that, by Lem. 1, we have $|\Delta| \leq j-1$ and $|\tau(V)| \leq j$. By the union bound, the probability that **bad** is set in this round, $\mathbf{Pr}_j$, equals the probability that **bad** is set in either of the lines 005, 007, 012, 024, 030, 032, 035 or 104. Denote by $\mathbf{Pr}_j^{\mathtt{xxx}}$ the probability that **bad** is set in the j^{th} query in line xxx. Observe that $\mathbf{Pr}_j^{012} \leq \mathbf{Pr}_j^{024} \leq \mathbf{Pr}_j^{035}$, and the simulator will either execute *one of these lines*. Similar observation holds for $\mathbf{Pr}_j^{005} \leq \mathbf{Pr}_j^{030}$ and $\mathbf{Pr}_j^{007} \leq \mathbf{Pr}_j^{032}$. Therefore, we obtain $\mathbf{Pr}\left(D^{G_3} \text{ sets } \mathbf{bad} \text{ in query } j\right) = \mathbf{Pr}_j \leq \mathbf{Pr}_j^{030\vee 032\vee 035} + \mathbf{Pr}_j^{104}$. We first consider $\mathbf{Pr}_j^{030\vee 032\vee 035}$. Boolean **bad** is set in either of these lines if v_{nxt}, taken uniformly at random from a set of size at least $2^{s-p} - 2r$ (notice that $(P_1, \ldots, P_l)$ are fixed), violates lines 029, 032, 033 or 034.

029. This line is violated if v_{nxt} corresponds to any capacity set already represented in $\tau(V)$. Consider a node $v \in \tau(V)$. By the definition of the capacity loss d in Sect. 4 (as $\mathsf{L}_{\text{ex}}(v_{\text{nxt}}) = P$ is fixed), there are at most 2^d values v_{nxt} such that $v \in \mathcal{C}(v_{\text{nxt}})$. In total, there are at most $j2^d$ possible values for v_{nxt} that make this line violated;

032. Line 032 is violated if $\mathsf{L}_{\text{out}}^{-1}[v, M](v_{\text{nxt}})$ hits a set of size at most $j-1$. Recall that L_{out} forms a bijection on the state (for fixed v, M);

033. Recall that $V_{\text{out}} = \bigcup_{x_0 \in \mathsf{dom}(\text{Sim-}\pi)} \mathcal{C}(x_0)$. Line 033 is violated if v_{nxt} hits any of the sets $\mathcal{C}(x_0)$, for $x_0 \in \mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \Delta$. However, by the definition of d, for each x_0, there are at most 2^d values v_{nxt} that would satisfy $v_{\text{nxt}} \in \mathcal{C}(x_0)$. In total, there are at most $(2j-1)2^d$ possible values for v_{nxt} that make this line violated;

034. This line is violated if v_{nxt} shares the same capacity set with any of the elements in $\tau(V)$. Consider a node $v \in \tau(V)$: this element exactly defines *one* capacity set[4], say $\mathcal{C}(x_0)$. By the definition of d (as $\mathsf{L}_{\text{ex}}(v_{\text{nxt}}) = P$ is fixed), there are at most 2^d values v_{nxt} that would satisfy $v_{\text{nxt}} \in \mathcal{C}(x_0)$. In total, there are at most $j2^d$ possible values for v_{nxt} that make this line violated.

Summarizing, we obtain $\mathbf{Pr}_j^{030\vee 032\vee 035} \leq \frac{(4j-1)2^d+j-1}{2^{s-p}-2r}$. Now for $\mathbf{Pr}_j^{104}$: notice that **bad** is set in line 104 if x, taken uniformly at random from a set of size at least $2^s - r$, hits a set of size at most $j2^{p+d} + j - 1$ (by the definition of d, each v is a member of at most 2^{p+d} capacity sets). Concluding, $\mathbf{Pr}_j^{104} \leq \frac{j2^{p+d}+j-1}{2^s-r}$. By the union bound, and under the assumption that $2r < 2^{s-p-1}$, we thus obtain

$$\mathbf{Pr}\left(\mathcal{D}^{G_3} \text{ sets } \mathbf{bad}\right) \leq \frac{7r(r+1)}{2^{s-p-d}}.$$

For games G_1, G_2, the same analysis holds. For games G_5, G_6, G_7, the same bound can be obtained similarly, and we only highlight the major differences: (i) v_{nxt} is now generated randomly from a set of size at least $2^s - 2r$, and (ii) we cannot use the fact that $\mathsf{L}_{\text{ex}}(v_{\text{nxt}}) = P$ is fixed anymore, but still the same analysis holds with 2^d replaced by 2^{p+d}. In general, however, the same bound is obtained for these games. Now, combined with (6), these bounds give the claimed result. □

[4] Here we require the property of L_{in} that all capacity sets are the same if they share one element. It is clear that this requirement can be relaxed at a cost of 2^m, as mentioned in Sect. 3.1.

Proposition 1. *As long as* **bad** *is not set in any of the games 2 and 3, both games are identical. Formally, we have* $\mathbf{Pr}\left(\mathcal{D}^{G_2} = 1 \;\middle|\; \mathcal{D}^{G_2} \text{ sets } \neg\mathbf{bad}\right) = \mathbf{Pr}\left(\mathcal{D}^{G_3} = 1 \;\middle|\; \mathcal{D}^{G_3} \text{ sets } \neg\mathbf{bad}\right)$.

Proof. We need to prove that, until **bad** is set in either one of the two games, the query outcomes in games 2 and 3 are identically distributed. As both games employ the same right oracle, a distinguisher can differentiate game 2 and 3 only based on its answers obtained from the left oracle.
Consider an execution of game 2 or game 3. Recall that Sim-π consists of a list of query pairs to the right oracle, and that (V, E) is the graph defined by these. Denote by $\mathcal{V} = (\mathcal{V}_L, \mathcal{V}_R)$ any view of a distinguisher on an execution of the oracle (G_2 or G_3). Here, $\mathcal{V}_L$ is a list of *different* query pairs (M_i, h_i), and $\mathcal{V}_R$ a list of query pairs for the right world. In game 2, we have $\mathcal{V}_R = \text{Sim-}\pi$, and in game 3, we have $\mathcal{V}_R \subseteq \text{Sim-}\pi$ (by construction). Denote by $(\overline{V}, \overline{E})$ the subgraph of (V, E) generated by the query pairs in $\mathcal{V}_R$. We need to prove that, given any view $\mathcal{V}$, outcomes of new queries to the left oracle are identically distributed in both games. Formally, we need to prove that for any $M \in \mathbb{Z}_2^*$ and any $\alpha \in \mathbb{Z}_2^n$, we have

$$\begin{aligned} &\mathbf{Pr}\left(L_2(M) = \alpha \text{ in } G_2 \mid (\mathcal{V}_L, \mathcal{V}_R);\; M \notin \mathsf{dom}(\mathcal{V}_L);\; \mathcal{D}^{G_2} \text{ sets } \neg\mathbf{bad}\right) \\ = &\,\mathbf{Pr}\left(L_3(M) = \alpha \text{ in } G_3 \mid (\mathcal{V}_L, \mathcal{V}_R);\; M \notin \mathsf{dom}(\mathcal{V}_L);\; \mathcal{D}^{G_3} \text{ sets } \neg\mathbf{bad}\right). \end{aligned} \tag{7}$$

Define $(M_1, \ldots, M_k) = \mathsf{pad}(M)$ to be the padded message of M. We will call the queried message M "**determined**" by $\mathcal{V}_R$ if there exists a path $\mathsf{IV} \xrightarrow{M_1} \cdots \xrightarrow{M_k} v_k$ in $(\overline{V}, \overline{E})$. We will prove that if M is not determined by $\mathcal{V}_R$, both probabilities in (7) equal $1/2^n$. On the other hand, if M is determined by $\mathcal{V}_R$, both properties are still equal (although they may naturally have a higher value).

M **is not determined by** $\mathcal{V}_R$**.** As the tree in $(\overline{V}, \overline{E})$ contains no path labeled by $M_1 \cdots M_k$, and moreover $M \notin \mathsf{dom}(\mathcal{V}_L)$, in both games the oracle L_1 had never been queried on M. Now, in game 2, L_2 passes the query through to L_1, which will generate its answer uniformly at random from $\mathbb{Z}_2^n$ (line `201`). In game 3, the **for**-loop of line `402` will force the right oracle to grow the path $\mathsf{IV} \xrightarrow{M_1} \cdots \xrightarrow{M_k} v_k$ to some node v_k. By Lem. 1 and as M is not determined by $\mathcal{V}_R$, at least the last edge $v_{k-1} \xrightarrow{M_k} v_k$ will be newly added to the tree. By construction, the oracle R_3 will generate the query answer via the **else**-clause of line `025`. It will query a *new* value to L_1, which samples the value $h \xleftarrow{\$} \mathbb{Z}_2^n$. The oracle R_3 will represent this value h in an obfuscated way in a list of l values $(P_1, \ldots, P_l)$ and answer all future queries from L_3 such that this oracle will exactly extract the same values $(P_1, \ldots, P_l)$. Consequently, the value h outputted by L_3 in line `411`, exactly equals the one randomly sampled;

M **is determined by** $\mathcal{V}_R$**.** By construction, the path $\mathsf{IV} \xrightarrow{M_1} \cdots v_{k-1} \xrightarrow{M_k} v_k$ is in the tree, for some v_{k-1}, v_k. But by Lem. 1, the tree is only increased with one edge at a time, and as a consequence, the edge (v_{k-1}, v_k) labeled by M_k must have been added to the tree *after* $M_1 \cdots M_{k-1}$ are known. Additionally, the tree is never increased in inverse queries (Lem. 1), and a determinatable path is never created in queries for $x \in \Delta$ (Lem. 2). In other words, this specific edge had been added in a forward query via the **else**-clause of line `025`. In particular, the oracle R_1 already generated $(P_1, \ldots, P_l)$ such that $\mathsf{fin}(P_1, \ldots, P_l) = L_1(M)$ (by Lem. 1 there are no collisions in the tree, and thus the oracle indeed queried L_1 on M). He additionally defined the node v_k as the next state in the extraction phase corresponding to the above-described path. Additionally, he saved $(v_k; 1, P_2, \ldots, P_l)$ in Δ. By construction, this value v_k satisfies $\mathsf{L}_{\text{ex}}(v_k) = P_1$. In particular, $\mathcal{V}_R$ and Δ jointly deterministically define $(P_1, \ldots, P_l)$, and therewith $L_1(M)$. However, the distinguisher does not know Δ. Let $i^* \in \{0, \ldots, l-1\}$ be the maximal index such that $v_k, \ldots, v_{k+i^*-1} \in \mathsf{dom}(\mathcal{V}_R)$. Recall that we have $P_i = \mathsf{L}_{\text{ex}}(v_{k+i-1})$ and $v_{k+i} = \text{Sim-}\pi(v_{k+i-1})$, for $i = 1, \ldots, l$. Then, using this equation, the distinguisher can deterministically determine $P_1, \ldots, P_{i^*}$ from $\mathcal{V}_R$. As $\text{Sim-}\pi(v_{k+i^*})$ is unknown to the distinguisher, and all other values in $(\mathcal{V}_L, \mathcal{V}_R)$ are unrelated[5], the distinguisher is oblivious to the values $P_{i^*+1}, \ldots, P_l$ such that $\mathsf{fin}(P_1, \ldots, P_l) = L_1(M)$. This analysis holds for

[5] It may be the case that $v_{k+i} \in \mathsf{dom}(\mathcal{V}_R)$ for some $i > i^*$, but this only happens with negligible probability. Moreover, from the view of the distinguisher, this edge is equally likely to be a part of the tail as any other edge.

both games: in game 2, the oracle L_2 will output this pre-determined h. In game 3, by construction of the **if**-clause of line 002, the oracle L_3 will output the same pre-determined h. □

Proposition 2. *As long as* **bad** *is not set in any of the games 5 and 6 (Fig. 9), both games are statistically indistinguishable. Formally, we have:*

$$\left|\mathbf{Pr}\left(\mathcal{D}^{G_5}=1 \mid \mathcal{D}^{G_5} \textit{ sets } \neg\mathbf{bad}\right)-\mathbf{Pr}\left(\mathcal{D}^{G_6}=1 \mid \mathcal{D}^{G_6} \textit{ sets } \neg\mathbf{bad}\right)\right| \leq \frac{2r^2}{2^s}.$$

Proof. For the purpose of the proof, we construct two new games 5a and 6a. Game 5a differs from game 5 in the sense that line 028 is replaced by:

```
028a $v_{\mathrm{nxt}} \xleftarrow{\$} \mathsf{L}_{\mathrm{ex}}^{-1}(P_1)$
028b if $v_{\mathrm{nxt}} \in \mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \Delta$ :
028c     $\mathbf{bad}' \leftarrow \mathbf{true}$; GOTO 028a
```

and similar for line 004 of game 5. $\mathcal{D}$ does not see the difference, and hence $\mathbf{Pr}\left(\mathcal{D}^{G_5}=1\right) = \mathbf{Pr}\left(\mathcal{D}^{G_{5a}}=1\right)$. Game 6a differs from game 6 in the sense that line 027 is replaced by:

```
027a $v_{\mathrm{nxt}} \xleftarrow{\$} \mathbb{Z}_2^s$
027b if $v_{\mathrm{nxt}} \in \mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \Delta$ :
027c     $\mathbf{bad}' \leftarrow \mathbf{true}$; GOTO 027a
```

and similar for line 003 of game 6. Similarly as before, $\mathbf{Pr}\left(\mathcal{D}^{G_6}=1\right) = \mathbf{Pr}\left(\mathcal{D}^{G_{6a}}=1\right)$, and the problem reduces to bounding $\left|\mathbf{Pr}\left(\mathcal{D}^{G_{5a}}=1\right)-\mathbf{Pr}\left(\mathcal{D}^{G_{6a}}=1\right)\right|$. By the properties of L_{ex}, both games distribute the (P_i, v_{nxt})'s identically as long as **bad** and **bad**$'$ are not set. As a consequence, we obtain:

$$\begin{aligned}\Big|\mathbf{Pr}\left(\mathcal{D}^{G_{5a}}=1 \mid \mathcal{D}^{G_{5a}} \text{ sets } \neg\mathbf{bad}\right)-\mathbf{Pr}\left(\mathcal{D}^{G_{6a}}=1 \mid \mathcal{D}^{G_{6a}} \text{ sets } \neg\mathbf{bad}\right)\Big| \\ \leq \mathbf{Pr}\left(\mathcal{D}^{G_{5a}} \text{ sets } \mathbf{bad}'\right)+\mathbf{Pr}\left(\mathcal{D}^{G_{6a}} \text{ sets } \mathbf{bad}'\right).\end{aligned}$$

Consider game 5a, and assume $\mathcal{D}^{G_{5a}}$ did not set **bad**. Suppose the right oracle has been queried $j-1$ times, and consider the j^{th} query x to the right oracle. As the simulator will either execute line 004 or 028 (not both), and will set **bad**$'$ with a higher probability in 028c, it suffices to consider this line only. Notice that we have $|\mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \Delta| \leq 2j-1$ (by Lem. 1). The probability that **bad**$'$ is set in 028c equals the probability that a value v_{nxt}, randomly sampled from a set of size 2^s (recall that line 028 is preceded by $P_1 \xleftarrow{\$} \mathbb{Z}_2^p$ and that L_{ex} is balanced), hits a value in a set of size at most $2j-1$. As a consequence, **bad**$'$ is set in the j^{th} query with probability at most $\frac{2j-1}{2^s}$. Summed over j, we obtain $\mathbf{Pr}\left(\mathcal{D}^{G_{5a}} \text{ sets } \mathbf{bad}'\right) \leq \frac{r^2}{2^s}$. Similarly, we obtain $\mathbf{Pr}\left(\mathcal{D}^{G_{6a}} \text{ sets } \mathbf{bad}'\right) \leq \frac{r^2}{2^s}$. □

Proposition 3. *As long as* **bad** *is not set in any of the games 6 and 7 (Figs. 9 and 10), both games are statistically indistinguishable. Formally, we have:*

$$\left|\mathbf{Pr}\left(\mathcal{D}^{G_6}=1 \mid \mathcal{D}^{G_6} \textit{ sets } \neg\mathbf{bad}\right)-\mathbf{Pr}\left(\mathcal{D}^{G_7}=1 \mid \mathcal{D}^{G_7} \textit{ sets } \neg\mathbf{bad}\right)\right| \leq \frac{2r^2}{2^s}.$$

Proof. The proof is similar to the one of Prop. 2, and skipped for conciseness. □

Lemma 1. *For games G_i ($i = 1, \dots, 8$), the following holds. Let (V, E) be the graph generated during the execution of the games, and let $(\tau(V), \tau(E))$ be the subgraph spanned by the nodes in $\tau(V)$. Let $(\tau(V)_j, \tau(E)_j)$ be the subgraph of $(\tau(V), \tau(E))$ after the j^{th} query. Let Δ_j denote the set Δ right after the j^{th} query. Under the condition that in the execution of the game,* **bad** *is not set, the following properties are satisfied for any $j \geq 0$.*

(i) *We have $|\mathcal{C}(x) \cap \tau(V)_j| \leq 1$ for any $x \in \mathbb{Z}_2^s$;*
(ii) *We have $|\tau(V)_j| \leq j+1$ and $|\tau(E)_j| \leq j$. In particular, $\tau(V)$ is a tree in (V, E);*
(iii) *We have $|\Delta_j| \leq j$.*

Proof (only for game 7). First of all, (iii) is satisfied as any query to the R_5 adds at most one element to Δ. The proof of the other properties is done by mathematical induction. Before the first query to R_5 is made, we have $|\tau(V)_0| = 1$ and $|\tau(E)_0| = 0$ and the claims are naturally satisfied. Now, assume the claims hold after $j-1$ queries, and consider the j^{th} query. We distinguish between forward and inverse queries.
Inverse query. On input of y, the simulator outputs some value x. This query adds at most 2^{p+d} edges to the graph, leaving from the vertices in $\mathcal{C}(x)$. As the simulator did not set **bad** in line `104`, we have $|\mathcal{C}(x) \cap \tau(V)_{j-1}| = 0$, and as a consequence, the size of the tree is not increased. The claims now follow by the induction hypothesis.
Forward query. On input of x, the simulator outputs some value y, and this is the only query pair added to Sim-π. This query adds at most 2^{p+d} edges to the graph, leaving from the vertices in $\mathcal{C}(x)$. If $|\mathcal{C}(x) \cap \tau(V)_{j-1}| = 0$, the same happens as for inverse queries, and all properties are satisfied by induction. Suppose $\mathcal{C}(x) \cap \tau(V)_{j-1} \neq \emptyset$. By the induction hypothesis for (i), there is exactly *one* value v in this intersection, which by the properties of L_{in} uniquely defines M such that an edge $v \xrightarrow{M} w$ to $w = \mathsf{L}_{\text{out}}(y, v, M)$ will be added. Thus, this (x, y)-pair defines exactly *one* new edge in the tree starting from a node in $\tau(V)_{j-1}$. In terms of Fig. 10, either the **if**-clause of line `008`, or the **else**-clause of `018` will be executed. In both cases, as **bad** is not set, y is generated in the same manner, and for simplicity we will analyze the run of the **if**-clause of line `008` only. As **bad** is not set via line `010`, the end node w is *no* element of $V_{\text{out}} \cup \mathcal{C}(x)$. In other words, w has no outgoing edge in the updated graph. As a consequence, $\tau(E)_j = \tau(E)_{j-1} \cup \{(v, w)\}$. Also as **bad** is not set via line `011`, the end node w is not yet in the tree $\tau(V)_{j-1}$, and will henceforth be newly added. These two observations prove property (ii). Remains to prove that the first property is satisfied. To the contrary, suppose $|\mathcal{C} \cap \tau(V)_j| = 2$ for some capacity set $\mathcal{C}$. This implies that $v', w \in \mathcal{C}$ for some $v' \in \tau(V)_{j-1}$, which contradicts with the fact that **bad** is not set via line `011`. □

Lemma 2. *Consider games 2 and 3 (Fig. 8), and assume that* **bad** *is not set. A query $R_1(x)$, with $x \in \Delta$, never increases $\bar{\tau}(V)$.*

Proof. First of all, if $l = 1$, lines `015` and `036` never increase Δ, and the claim is naturally satisfied as $\Delta = \emptyset$ throughout. Otherwise, by virtue of the condition in Sect. 3.4, the last block of a padded message always satisfies (2). We will show that, if a query $R_1(x)$ for $x \in \Delta$ adds an edge to the tree, the message block M that labels this path satisfies $\mathsf{L}_{\text{in}}(x, M) = x$ or $\mathsf{L}_{\text{in}}(\mathsf{L}_{\text{out}}(x, v', M'), M) = x$ for some (v', M'), therewith proving the claim.
Inverse queries never increase the tree (Lem. 1), and we will ignore them for simplicity. For readability, we will add a subscripts to the values x and the tree. Denote by $\tau(V)_j$ the tree in (V, E) after the j^{th} query, for $j = 1, \ldots, r$. Suppose x_j is the j^{th} query to the right oracle, and suppose $x_j \in \Delta$ (notice that x_j occurs exactly once in Δ, due to lines `003` and `028`). The node x_j had been added to Δ in either line `015` or `036`, in one of the previous queries, say the i^{th} query, for $i < j$. By lines `004` and `029`, x_j had been generated such that $\mathcal{C}(x_j) \cap \tau(V)_{i-1} = \emptyset$. Due to the statement "$\bigcup_{x \in \Delta} \mathcal{C}(x)$" in lines `010`, `022` and `033`, it is assured that any new node added to the tree in the $(i+1)^{\text{th}}$ up to the $(j-1)^{\text{th}}$ query, is not in $\mathcal{C}(x_j)$. Formally, we have $\mathcal{C}(x_j) \cap \big(\tau(V)_{j-1} \backslash \tau(V)_i\big) = \emptyset$. As moreover $\mathcal{C}(x_j) \cap \tau(V)_{i-1} = \emptyset$, this means that the query x_j to R_1 increases $\tau(V)$ *only if* $\mathsf{L}_{\text{out}}(y_i, v', M') \in \mathcal{C}(x_j)$ for some v', M' (coming from the i^{th} execution). By the properties of L_{in}, these values uniquely define the message block M that labels the new edge created by the query x_j, namely: M satisfies $\mathsf{L}_{\text{in}}(\mathsf{L}_{\text{out}}(y_i, v', M'), M) = x_j$. However, by construction (lines `006` or `031`), we either have $y_i = x_j$ or $y_i = \mathsf{L}_{\text{out}}^{-1}[v', M'](x_j)$. Consequently, the message block violates the (2). Therefore, M cannot be the last block of any padded message, and in particular $\bar{\tau}(V)$ is not increased. □

```
Forward Query $R_1(x)$
000 if $x \in \mathsf{dom}(\text{Sim-}\pi)$ :
001     return $y = \text{Sim-}\pi(x)$
002 if $x \in \Delta$ assoc. with some $(i; P_{i+1} \cdots P_l)$ :
003     $v_{\text{nxt}} \xleftarrow{\$} \mathsf{L}_{\text{ex}}^{-1}(P_{i+1}) \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \Delta\big)$
004     if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
005         bad $\leftarrow$ true; GOTO 003
006     $y \leftarrow v_{\text{nxt}}$
007     if $y \in \mathsf{rng}(\text{Sim-}\pi)$ : bad $\leftarrow$ true; GOTO 003
008     if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
009         $\begin{cases} \text{find unique } v \in \tau(V) \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x \end{cases}$
010         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \big(\bigcup_{x \in \Delta} \mathcal{C}(x)\big)$
011         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
012             bad $\leftarrow$ true; GOTO 003
013     end if
014     $\Delta \leftarrow \Delta \backslash \{(x; i, P_{i+1} \cdots P_l)\}$
015     if $i + 1 < l$ : $\Delta \leftarrow \Delta \cup \{(v_{\text{nxt}}; i+1, P_{i+2} \cdots P_l)\}$
016 else if $\mathcal{C}(x) \cap \tau(V) = \emptyset$ :
017     $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
018 else if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
019     $\begin{cases} \text{find unique } v \in \tau(V), \bar{M} \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x \text{ and } \mathsf{IV} \xrightarrow{\bar{M}} v \end{cases}$
020     if $\bar{M} \| M \notin \mathsf{rng}(\mathsf{pad})$ :
021         $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
022         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \big(\bigcup_{x \in \Delta} \mathcal{C}(x)\big)$
023         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
024             bad $\leftarrow$ true; GOTO 021
025     else if $\bar{M} \| M \in \mathsf{rng}(\mathsf{pad})$ :
026         $h \xleftarrow{\$} L_1(\mathsf{depad}(\bar{M} \| M))$
027         $(P_1, \ldots, P_l) \xleftarrow{\$} \mathsf{fin}^{-1}(h)$
028         $v_{\text{nxt}} \xleftarrow{\$} \mathsf{L}_{\text{ex}}^{-1}(P_1) \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \Delta\big)$
029         if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
030             bad $\leftarrow$ true; GOTO 028
031         $y \leftarrow \mathsf{L}_{\text{out}}^{-1}[v, M](v_{\text{nxt}})$
032         if $y \in \mathsf{rng}(\text{Sim-}\pi)$ : bad $\leftarrow$ true; GOTO 028
033         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \big(\bigcup_{x \in \Delta} \mathcal{C}(x)\big)$
034         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
035             bad $\leftarrow$ true; GOTO 028
036         if $1 < l$ : $\Delta \leftarrow \Delta \cup \{(v_{\text{nxt}}; 1, P_2 \cdots P_l)\}$
037     end if
038 end if
039 return $\text{Sim-}\pi(x) \leftarrow y$
```

```
Inverse Query $R_1^{-1}(y)$
100 if $y \in \mathsf{rng}(\text{Sim-}\pi)$ :
101     return $x = \text{Sim-}\pi^{-1}(y)$
102 $x \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{dom}(\text{Sim-}\pi)$
103 if $x \in \Delta$ or $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
104     bad $\leftarrow$ true; GOTO 102
105 return $\text{Sim-}\pi^{-1}(y) \leftarrow x$
```

```
Query $L_1(M)$
200 if $M \in \mathsf{dom}(\mathcal{H})$ return $h = \mathcal{H}(M)$
201 $h \xleftarrow{\$} \mathbb{Z}_2^n$
202 return $\mathcal{H}(M) \leftarrow h$
```

```
Query $L_2(M)$
300 return $h \leftarrow L_1(M)$
```

```
Query $L_3(M)$
400 $(M_1, \ldots, M_k) \leftarrow \mathsf{pad}(M)$
401 $v_0 \leftarrow \mathsf{IV}$
402 for $i = 1, \ldots, k$ :
403     $x \leftarrow \mathsf{L}_{\text{in}}(v_{i-1}, M_i)$
404     $y \leftarrow R_1(x)$
405     $v_i \leftarrow \mathsf{L}_{\text{out}}(y, v_{i-1}, M_i)$
406 end for
407 for $i = 1, \ldots, l$ :
408     $P_i \leftarrow \mathsf{L}_{\text{ex}}(v_{k+i-1})$
409     $v_{k+i} \leftarrow R_1(v_{k+i-1})$
410 end for
411 $h \leftarrow \mathsf{fin}(P_1, \ldots, P_l)$
412 return $h$
```

Fig. 8. Games 1, 2 and 3. In game 1, the distinguisher has access to $L_1, R_1^{L_1}$ (oracles L_2, L_3 are ignored). In game 2, the distinguisher has access to $L_2^{L_1}, R_1^{L_1}$ (oracle L_3 is ignored). In game 3, the distinguisher has access to $L_3^{R_1^{L_1}}, R_1^{L_1}$ (oracle L_2 is ignored). Oracle L_1 maintains an initially empty table $\mathcal{H}$.

Forward Query $R_\alpha(x)$

```
000 if $x \in \mathsf{dom}(\text{Sim-}\pi)$ :
001     return $y = \text{Sim-}\pi(x)$
002 if $x \in \varDelta$ assoc. with some $i$ :
    for $\alpha = 3$ :
003     $P_{i+1} \xleftarrow{\$} \mathbb{Z}_2^p$
004     $v_{\text{nxt}} \xleftarrow{\$} \mathsf{L}_{\text{ex}}^{-1}(P_{i+1}) \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \varDelta\big)$
    for $\alpha = 4$ :
003     $v_{\text{nxt}} \xleftarrow{\$} \mathbb{Z}_2^s \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \varDelta\big)$
004     $P_{i+1} \leftarrow \mathsf{L}_{\text{ex}}(v_{\text{nxt}})$
005     if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
006         bad $\leftarrow$ true; GOTO 004
007     $y \leftarrow v_{\text{nxt}}$
008     if $y \in \mathsf{rng}(\text{Sim-}\pi)$ : bad $\leftarrow$ true; GOTO 004
009     if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
010         $\begin{cases} \text{find unique } v \in \tau(V) \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x \end{cases}$
011         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \big(\bigcup_{x \in \varDelta} \mathcal{C}(x)\big)$
012         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
013             bad $\leftarrow$ true; GOTO 004
014     end if
015     $\varDelta \leftarrow \varDelta \backslash \{(x; i)\}$
016     if $i + 1 < l$ : $\varDelta \leftarrow \varDelta \cup \{(v_{\text{nxt}}; i + 1)\}$
017 else if $\mathcal{C}(x) \cap \tau(V) = \emptyset$ :
018     $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
019 else if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
020     $\begin{cases} \text{find unique } v \in \tau(V), \bar{M} \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x \text{ and } \mathsf{IV} \xrightarrow{\bar{M}} v \end{cases}$
021     if $\bar{M} \| M \notin \mathsf{rng}(\mathsf{pad})$ :
022         $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
023         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \big(\bigcup_{x \in \varDelta} \mathcal{C}(x)\big)$
024         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
025             bad $\leftarrow$ true; GOTO 022
026     else if $\bar{M} \| M \in \mathsf{rng}(\mathsf{pad})$ :
    for $\alpha = 3$ :
027         $P_1 \xleftarrow{\$} \mathbb{Z}_2^p$
028         $v_{\text{nxt}} \xleftarrow{\$} \mathsf{L}_{\text{ex}}^{-1}(P_1) \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \varDelta\big)$
    for $\alpha = 4$ :
027         $v_{\text{nxt}} \xleftarrow{\$} \mathbb{Z}_2^s \backslash \big(\mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \varDelta\big)$
028         $P_1 \leftarrow \mathsf{L}_{\text{ex}}(v_{\text{nxt}})$
029         if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
030             bad $\leftarrow$ true; GOTO 028
031         $y \leftarrow \mathsf{L}_{\text{out}}^{-1}[v, M](v_{\text{nxt}})$
032         if $y \in \mathsf{rng}(\text{Sim-}\pi)$ : bad $\leftarrow$ true; GOTO 028
033         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \big(\bigcup_{x \in \varDelta} \mathcal{C}(x)\big)$
034         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
035             bad $\leftarrow$ true; GOTO 028
036         if $1 < l$ : $\varDelta \leftarrow \varDelta \cup \{(v_{\text{nxt}}; 1)\}$
037     end if
038 end if
039 return $\text{Sim-}\pi(x) \leftarrow y$
```

Inverse Query $R_\alpha^{-1}(y)$

```
100 if $y \in \mathsf{rng}(\text{Sim-}\pi)$ :
101     return $x = \text{Sim-}\pi^{-1}(y)$
102 $x \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{dom}(\text{Sim-}\pi)$
103 if $x \in \varDelta$ or $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
104     bad $\leftarrow$ true; GOTO 102
105 return $\text{Sim-}\pi^{-1}(y) \leftarrow x$
```

Query $L_3(M)$

```
400 $(M_1, \ldots, M_k) \leftarrow \mathsf{pad}(M)$
401 $v_0 \leftarrow \mathsf{IV}$
402 for $i = 1, \ldots, k$ :
403     $x \leftarrow \mathsf{L}_{\text{in}}(v_{i-1}, M_i)$
404     $y \leftarrow R_\alpha(x)$
405     $v_i \leftarrow \mathsf{L}_{\text{out}}(y, v_{i-1}, M_i)$
406 end for
407 for $i = 1, \ldots, l$ :
408     $P_i \leftarrow \mathsf{L}_{\text{ex}}(v_{k+i-1})$
409     $v_{k+i} \leftarrow R_\alpha(v_{k+i-1})$
410 end for
411 $h \leftarrow \mathsf{fin}(P_1, \ldots, P_l)$
412 return $h$
```

Fig. 9. Game 5 (for $\alpha = 3$) and game 6 (for $\alpha = 4$). In both games, the distinguisher has access to $L_3^{R_\alpha}, R_\alpha$.

```
Forward Query $R_\alpha(x)$
000 if $x \in \mathsf{dom}(\text{Sim-}\pi)$ :
001     return $y = \text{Sim-}\pi(x)$
002 if $x \in \Delta$ assoc. with some $i$ :
003     $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
004     $v_{\text{nxt}} \leftarrow y$
005     if $v_{\text{nxt}} \in \mathsf{dom}(\text{Sim-}\pi) \cup \Delta$
006     or if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
007         bad $\leftarrow$ true; $\boxed{\textbf{GOTO } 003}$
008     if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
009         $\left\{\begin{array}{l}\text{find unique } v \in \tau(V) \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x\end{array}\right.$
010         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \left(\bigcup_{x \in \Delta} \mathcal{C}(x)\right)$
011         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
012             bad $\leftarrow$ true; $\boxed{\textbf{GOTO } 003}$
013     end if
014     $\Delta \leftarrow \Delta \backslash \{(x; i)\}$
015     if $i + 1 < l$ :  $\Delta \leftarrow \Delta \cup \{(v_{\text{nxt}}; i + 1)\}$
016 else if $\mathcal{C}(x) \cap \tau(V) = \emptyset$ :
017     $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
018 else if $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
019     $\left\{\begin{array}{l}\text{find unique } v \in \tau(V), \bar{M} \text{ and } M \\ \text{s.t. } \mathsf{L}_{\text{in}}(v, M) = x \text{ and } \mathsf{IV} \xrightarrow{\bar{M}} v\end{array}\right.$
020     if $\bar{M} \| M \notin \mathsf{rng}(\mathsf{pad})$ :
021         $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
022         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \left(\bigcup_{x \in \Delta} \mathcal{C}(x)\right)$
023         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
024             bad $\leftarrow$ true; $\boxed{\textbf{GOTO } 021}$
025     else if $\bar{M} \| M \in \mathsf{rng}(\mathsf{pad})$ :
026         $y \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{rng}(\text{Sim-}\pi)$
027         $v_{\text{nxt}} \leftarrow \mathsf{L}_{\text{out}}(y, v, M)$
028         if $v_{\text{nxt}} \in \mathsf{dom}(\text{Sim-}\pi) \cup \{x\} \cup \Delta$
029         or if $\mathcal{C}(v_{\text{nxt}}) \cap \tau(V) \neq \emptyset$ :
030             bad $\leftarrow$ true; $\boxed{\textbf{GOTO } 026}$
031         if $\mathsf{L}_{\text{out}}(y, v, M) \in V_{\text{out}} \cup \mathcal{C}(x) \cup \left(\bigcup_{x \in \Delta} \mathcal{C}(x)\right)$
032         or if $\exists\, v' \in \tau(V) : \mathsf{same}\mathcal{C}(v', \mathsf{L}_{\text{out}}(y, v, M))$ :
033             bad $\leftarrow$ true; $\boxed{\textbf{GOTO } 026}$
034         if $1 < l$ :  $\Delta \leftarrow \Delta \cup \{(v_{\text{nxt}}; 1)\}$
035     end if
036 end if
037 return $\text{Sim-}\pi(x) \leftarrow y$
```

```
Inverse Query $R_\alpha^{-1}(y)$
100 if $y \in \mathsf{rng}(\text{Sim-}\pi)$ :
101     return $x = \text{Sim-}\pi^{-1}(y)$
102 $x \xleftarrow{\$} \mathbb{Z}_2^s \backslash \mathsf{dom}(\text{Sim-}\pi)$
103 if $x \in \Delta$ or $\mathcal{C}(x) \cap \tau(V) \neq \emptyset$ :
104     bad $\leftarrow$ true; $\boxed{\textbf{GOTO } 102}$
105 return $\text{Sim-}\pi^{-1}(y) \leftarrow x$
```

```
Query $L_3(M)$
400 $(M_1, \ldots, M_k) \leftarrow \mathsf{pad}(M)$
401 $v_0 \leftarrow \mathsf{IV}$
402 for $i = 1, \ldots, k$ :
403     $x \leftarrow \mathsf{L}_{\text{in}}(v_{i-1}, M_i)$
404     $y \leftarrow R_\alpha(x)$
405     $v_i \leftarrow \mathsf{L}_{\text{out}}(y, v_{i-1}, M_i)$
406 end for
407 for $i = 1, \ldots, l$ :
408     $P_i \leftarrow \mathsf{L}_{\text{ex}}(v_{k+i-1})$
409     $v_{k+i} \leftarrow R_\alpha(v_{k+i-1})$
410 end for
411 $h \leftarrow \mathsf{fin}(P_1, \ldots, P_l)$
412 return $h$
```

Fig. 10. Game 7 (for $\alpha = 5$, including the boxed statements) and game 8 (for $\alpha = 6$, with the boxed statements removed). In both games, the distinguisher has access to $L_3^{R_\alpha}, R_\alpha$.