Alternating Sponge: A Low-Memory Hash Function with Beyond-Birthday-Bound Security

Ziyang Luo[1], Yaobin Shen[1], Hailun Yan[2], Lei Wang[3], and Dawu Gu[3]

[1] School of Informatics, Xiamen University, Xiamen, China
ziyangluo@stu.xmu.edu.cn, yaobin.shen@xmu.edu.cn
[2] School of Cryptology, University of Chinese Academy of Sciences, Beijing 100049, China
[3] Shanghai Jiao Tong University, Shanghai, China

Abstract. We propose the *Alternating Sponge* (ASP), a permutation-based hashing mode that explores how to obtain beyond-birthday-bound security without duplicating the full capacity state. The main design idea is to replace spatial state duplication by temporal alternation: ASP keeps two r-bit rate branches but lets them share a single c-bit capacity chain, and updates the two branches through alternating calls to two independent $(r + c)$-bit permutations. This yields a state size of $2r + c$ bits and outputs $2r$ bits per mode-level squeezing round.
We prove that ASP is indifferentiable from a random oracle in the ideal permutation model. For the base primitive-call budget $q = q_P + \lambda$, the bound gives a capacity-driven term of order $O(q^3/2^{2c})$, together with a primitive-space term of order $O(q^2/2^{r+c})$. This yields the asymptotic security level $\min\{\frac{2c}{3}, \frac{r+c}{2}\}$. In particular, for $r \geq \frac{c}{3}$, ASP reaches the $\frac{2c}{3}$ capacity-driven exponent while saving one c-bit capacity component compared with a fully duplicated two-branch design. We also discuss generic distinguishing attacks, including a construction-level state-merging birthday attack and a simulator-specific attack, which clarify the current proof–attack gap.

Keywords: permutation-based hashing · indifferentiability · beyond-birthday-bound security · low-memory hashing · lightweight cryptography

1 Introduction

1.1 Background and Motivation

Permutation-based hashing has become a central design paradigm in modern symmetric cryptography [3,7,14]. The sponge construction shows that a public permutation, combined with a simple rate–capacity split, can be used to build hash functions and extendable-output functions with a clean and unified interface [3]. Such constructions avoid key schedules, admit compact implementations, and can be analyzed in the random permutation model through indifferentiability from a random oracle [12].

A fundamental limitation of the standard sponge is that its generic indifferentiability security is governed by the capacity birthday bound. More precisely, for capacity c, the sponge is tightly indifferentiable from a random oracle up to roughly $2^{c/2}$ queries. This raises a natural question: can permutation-based hashing go beyond the birthday bound while preserving a simple and efficiently implementable mode?

Several lines of work have explored how to go beyond the sponge interface[2,9,5]. A first route uses wider internal states or wide-pipe designs, as represented by Grøstl and JH [8,15]. These modes are important permutation-based hash designs, but their provable indifferentiability bounds and best known attacks do not coincide with the standard sponge setting.

A second route studies generalized sponge families[2]. Recent work on the porifera family gives new attacks on FP- and JH-like designs and proposes VFB, a near-optimal construction whose bound $V(c)$ approaches c in the high-rate regime $r \geq c$, using a distinct finalization and output-expansion procedure [11]. This line shows that stronger bounds are possible, but it also relies on a different state organization and output interface.

The route closest to our work is Double Sponge [10]. It uses two $(r + c)$-bit permutations and maintains two complete sponge-like states, obtaining a capacity-driven bound of roughly $2^{2c/3}$ queries. Thus, existing work shows that the $2^{c/2}$ barrier of the standard sponge is not inherent to all keyless permutation-based hashing modes; the remaining question is what state organization is needed to obtain such beyond-birthday behavior.

The improvement of Double Sponge, however, comes from spatial duplication. Its state size is $2r + 2c$ bits, so both the rate part and the capacity part are duplicated. This motivates the central question of this work: Can beyond-birthday-bound indifferentiability be achieved without duplicating the full capacity state of the Double Sponge?

We answer this question by proposing an alternating design principle: *temporal alternation replaces spatial duplication*. Instead of maintaining two complete sponge states, our construction keeps two rate branches but lets them share a single capacity chain. The two branches interact over time through alternating permutation calls, rather than through two fully duplicated capacity components. This turns the problem from a purely implementation-driven memory reduction into a structural question about the design space of beyond-birthday-bound permutation-based hashing.

1.2 Our Construction and Results

We introduce the *Alternating Sponge* (ASP), a permutation-based hashing mode built from two independent $(r + c)$-bit permutations. ASP maintains two r-bit rate branches and one shared c-bit capacity chain, and therefore uses a state of size $2r + c$. Compared with Double Sponge, whose state size is $2r + 2c$, ASP saves one c-bit capacity component while retaining a two-branch structure.

At a high level, ASP replaces full spatial duplication by temporal alternation. Each message block is processed through two sequential permutation calls

sharing the same capacity path, and the two rate branches are alternately combined during squeezing. We measure the squeezing rate as the number of output bits exposed per mode-level squeezing round, where one such round consists of the two permutation calls used during squeezing. Under this measure, Double Sponge exposes one r-bit block per round, whereas ASP exposes two r-bit fragments per round. Thus ASP improves the squeezing rate from r to $2r$ bits per round relative to Double Sponge.

Our contributions can be summarized as follows.

- **A shared-capacity alternating mode.** We define ASP as a two-permutation hashing mode with state size $2r+c$. The construction keeps two rate branches but shares a single capacity chain, thereby reducing the duplicated capacity state of Double Sponge.
- **Indifferentiability bound.** We prove that ASP is indifferentiable from a random oracle in the ideal permutation model. Let $q = q_P + \lambda$ be the base primitive-call budget, where q_P is the number of direct primitive queries and λ is the number of primitive calls induced by hash queries expanded as ASP evaluations. The dominant terms in our bound are
$$O\left(\frac{q^3}{2^{2c}}\right), \qquad O\left(\frac{q^2}{2^{r+c}}\right).$$
The full bound is stated in Theorem 1.
- **Parameter interpretation.** The bound gives an asymptotic security level of
$$\min\left\{\frac{2c}{3}, \frac{r+c}{2}\right\}.$$
The two terms are balanced at $r = c/3$. For $r \geq c/3$, ASP reaches the capacity-driven $2c/3$ level of Double Sponge while using c fewer state bits. For $r < c/3$, the primitive-space term $(r+c)/2$ becomes the limiting factor.
- **Simulator for the shared-capacity structure.** The proof develops a simulator tailored to the alternating ASP structure. Instead of maintaining two independent full sponge states, the simulator tracks rooted absorbing and squeezing states, identifies activated contexts, imposes local rate constraints for random-oracle consistency, and uses lazy closure to complete only transitions forced by existing table entries.
- **Generic distinguishing attack analysis.** We also provide generic distinguishing attacks to clarify the current proof–attack gap of ASP. This analysis is used to position ASP among existing permutation-based hashing modes, rather than to claim that the indifferentiability bound is tight in every parameter regime.

1.3 Comparison with Existing Modes

ASP does not dominate all existing permutation-based hashing modes. Instead, it offers a different point in the design space: two rate branches, one shared

Table 1. Comparison with representative permutation-based hashing modes. The security bounds are in bits. The attack column reports the best known generic upper bound when available. For fixed-output modes such as Grøstl and JH, the rate column refers to the message-processing rate; for sponge-type modes, it refers to the squeezing rate per mode-level squeezing round.

Mode	Primitives per Round	Memory Size	Proven Security Bound	Best Known Attack	Rate / Output per Round	Reference
Sponge	1	$r+c$	$\frac{c}{2}$	$\frac{c}{2}$	r	[3,4]
Grøstl	2	$3b$	$\frac{b}{2}$	$\frac{b}{2}$	b	[8,1]
JH	2	b	$\frac{b}{4}$	$\frac{b}{3}$	$\frac{b}{2}$	[15,13,11]
Double Sponge	2	$2r+2c$	$\frac{2c}{3}$	$\frac{2c+r}{3}$	r	[10]
VFB	1	$r+3c$[1]	$V(c)$, approaching c	c	r	[11]
Alternating Sponge	2	$\mathbf{2r+c}$	$\min\{\frac{2c}{3}, \frac{r+c}{2}\}$	$\min\{\frac{r+c}{2}$ (constr.), $\frac{2c+r}{3}$ (sim.)$\}$[2]	$\mathbf{2r}$	This work

[1] For VFB, the original work reports the internal/chaining state size under its own accounting. The $r+3c$ expression used here follows an implementation-level working-memory accounting for comparison with low-memory hashing modes; it counts the permutation working state and the additional c-bit values retained around the feed-forward/finalization mechanism.

[2] The $(2c+r)/3$ term is simulator-specific: it is a distinguisher against the simulator used in the proof, not a construction-level attack on ASP.

capacity chain, a reduced working state compared with Double Sponge, and a beyond-birthday-type indifferentiability bound in the relevant parameter regime.

Table 1 compares ASP with representative sponge-type and permutation-based hashing architectures. The table reports memory size, proven security bound, and squeezing rate per mode-level round. The attack column is discussed later together with the tightness analysis.

With respect to the standard sponge, ASP uses two permutation calls and a larger state, and therefore it should not be viewed as a replacement for the simplest sponge construction. The standard sponge remains the natural baseline when one wants a single-permutation mode with minimal state. ASP instead targets the two-permutation setting, where the goal is to improve the tradeoff among security, state size, and squeezing behavior.

Double Sponge is the closest baseline. It achieves a $2c/3$-type beyond-birthday indifferentiability bound by maintaining two full $(r+c)$-bit sponge states. ASP keeps two rate branches while sharing the capacity part, reducing the state from $2r+2c$ to $2r+c$. Our security bound becomes

$$\min\left\{\frac{2c}{3}, \frac{r+c}{2}\right\},$$

Thus, when $r < c/3$, ASP is governed by the primitive-space term $(r+c)/2$, which is still beyond the birthday bound. When $r \geq c/3$, ASP reaches the same capacity-driven $2c/3$-order bound as Double Sponge.

VFB represents a recent high-rate direction in generalized sponge-family designs, achieving a stronger asymptotic bound in the regime $r \geq c$ through a distinct finalization and output-expansion mechanism. ASP explores a different part of the design space: it keeps the iterative sponge-like interface, allows

flexible (r, c) choices. Grøstl and JH follow yet another route, relying on wide-pipe or wide-state organizations. In contrast, ASP obtains its tradeoff from two alternating rate branches coupled by a single capacity chain.

In summary, ASP shows that in beyond-birthday-bound permutation-based hashing, full capacity duplication is not the only way to obtain stronger generic security. A shared capacity chain, combined with temporal alternation between two rate branches, can support an executable simulator proof and yield a concrete security–state–squeezing tradeoff.

2 Preliminaries

2.1 Notation

This subsection provides the fundamental notations and basic definitions used throughout this paper.

- r, c: denotes the rate and capacity size in bits.
- f_1, f_2: denote two independent permutations over $\{0,1\}^{r+c}$.
- σ: denotes the linear permutation applied to message blocks before the second-stage absorption.
- $\|$: denotes bit-string concatenation.
- $\oplus$: denotes bitwise XOR.
- $|X|$: denotes the bit length of a bit string X.
- $X[a:b]$: denotes the substring of X from position a to position b, inclusive.
- $\mathsf{rate}(Y), \mathsf{cap}(Y)$: denote the r-bit rate part and the c-bit capacity part of $Y \in \{0,1\}^{r+c}$, respectively, so that $Y = \mathsf{rate}(Y) \parallel \mathsf{cap}(Y)$.
- $\mathcal{RO}(M)[j]$: denotes the j-th r-bit output block of the random oracle on input M, indexed from 0.
- T_s, T_s^{-1}: denote the simulator's partial table for f_s and its inverse table, respectively, where $s \in \{1,2\}$.
- $\mathrm{Dom}(T_s), \mathrm{Img}(T_s)$: denote the domain and image of the partial table T_s, respectively.
- $\perp$: denotes the absence of a rate constraint.

3 The Alternating Sponge Construction

3.1 Algorithm Framework

State Partitioning and Permutation Process Design ASP maintains a fixed-size state of $2r + c$ bits, written as $(S_{\mathsf{up}}, S_{\mathsf{mid}}, C)$, where $S_{\mathsf{up}}, S_{\mathsf{mid}} \in \{0,1\}^r$ are two rate branches and $C \in \{0,1\}^c$ is the shared capacity part. The construction uses two independent permutations f_1, f_2 over $\{0,1\}^{r+c}$, which are applied sequentially to update the two rate branches through the same capacity chain.

For each padded message block $M_i \in \{0,1\}^r$, the absorbing update is defined by

$$(S'_{\mathsf{up}}, C') = f_1(S_{\mathsf{up}} \oplus M_i \| C), \qquad (S'_{\mathsf{mid}}, C'') = f_2(S_{\mathsf{mid}} \oplus \sigma(M_i) \| C'),$$

followed by

$$(S_{\mathsf{up}}, S_{\mathsf{mid}}, C) \leftarrow (S'_{\mathsf{up}}, S'_{\mathsf{mid}}, C'').$$

Here σ is a linear permutation on r-bit message blocks, used to separate the two message injections. Our proof does not require any additional property of σ; in an efficiency-oriented implementation, it may be chosen as the identity map. This two-stage absorbing update realizes the shared-capacity structure of ASP: two rate branches are updated sequentially through a single capacity chain, and each primitive call acts on $r + c$ bits.

Squeezing Phase and Output Mechanism After absorption, ASP reuses f_1 and f_2 to generate an arbitrary-length output string $Z = z_0 \| z_1 \| z_2 \| \cdots$. Let the final absorbing state be $(S_{\mathsf{up}}, S_{\mathsf{mid}}, C)$. The initial output fragment is

$$z_0 = S_{\mathsf{up}} \oplus S_{\mathsf{mid}}.$$

For each squeezing round $i \geq 0$, given the current state $(S_{\mathsf{up}}, S_{\mathsf{mid}}, C)$ and the even fragment z_{2i}, ASP computes

$$(S'_{\mathsf{up}}, C') = f_1(z_{2i} \| C), \qquad z_{2i+1} = S'_{\mathsf{up}} \oplus S_{\mathsf{mid}},$$

then applies the second permutation and updates the state by

$$(S'_{\mathsf{mid}}, C'') = f_2(z_{2i+1} \| C'), \qquad z_{2i+2} = S'_{\mathsf{up}} \oplus S'_{\mathsf{mid}},$$
$$(S_{\mathsf{up}}, S_{\mathsf{mid}}, C) \leftarrow (z_{2i+2}, S'_{\mathsf{mid}}, C'').$$

Thus each squeezing round, consisting of one f_1 call and one f_2 call, produces two r-bit fragments and advances the shared-capacity state. The construction therefore outputs $2r$ bits per squeezing round.

To clearly illustrate the state partitioning and permutation flow of the Alternating Sponge, Figure 1 presents its overall architecture. Dashed lines indicate state transfers rather than direct inputs to the current-round permutation. A more formal description is provided in the pseudocode (Algorithm 1), where the absorbing and squeezing phases are presented together in a single main procedure for a step-by-step overview of the process.

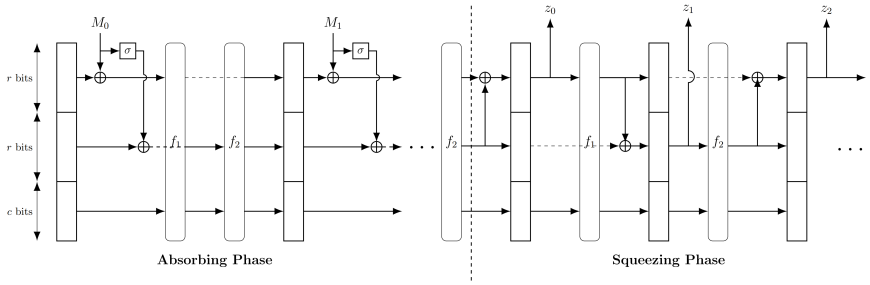


Fig. 1. Structure diagram of ASP

Algorithm 1 Alternating Sponge Operation Process

Require: Message M, desired output length l, rate r, capacity c
Ensure: Hash digest D

// **1. Padding**
$M' \leftarrow \text{Pad}(M, r)$
// **2. State Initialization**
$S_{up} \leftarrow 0^r; S_{mid} \leftarrow 0^r; C \leftarrow 0^c$
// **3. Absorbing Phase**
$num_absorb_blocks \leftarrow \text{length}(M')/r$
for i from 0 to $num_absorb_blocks - 1$ **do**
    $M_i \leftarrow M'[(i \cdot r) : ((i+1) \cdot r - 1)]$
    $(S'_{up}, C') \leftarrow f_1(S_{up} \oplus M_i \parallel C)$
    $(S'_{mid}, C'') \leftarrow f_2(S_{mid} \oplus \sigma(M_i) \parallel C')$
    // **State Update**
    $S_{up} \leftarrow S'_{up}; S_{mid} \leftarrow S'_{mid}; C \leftarrow C''$
end for
// **4. Squeezing Phase**
$output_digest \leftarrow$ empty string
$S_{up} \leftarrow S_{up} \oplus S_{mid}$ // **Initial XOR blending (z_0)**
$t \leftarrow 0$
while length($output_digest$) $<$ l **do**
    $z_{2t} \leftarrow S_{up}$ // **Current even fragment**
    $(S'_{up}, C') \leftarrow f_1(z_{2t} \parallel C)$
    $z_{2t+1} \leftarrow S'_{up} \oplus S_{mid}$ // **Extraction of odd fragment**
    $(S'_{mid}, C'') \leftarrow f_2(z_{2t+1} \parallel C')$
    // **State Update and Next Even Fragment (z_{2t+2})**
    $S_{up} \leftarrow S'_{up} \oplus S'_{mid}$
    $S_{mid} \leftarrow S'_{mid}; C \leftarrow C''$
    // Append fragments to the digest
    $output_digest \leftarrow output_digest \parallel z_{2t} \parallel z_{2t+1}$
    $t \leftarrow t + 1$
end while
// **5. Return the result truncated to l bits**
return $output_digest[0 : l - 1]$

3.2 Message Padding Mechanism

To ensure that every input can be parsed into r-bit blocks, ASP uses the standard 10^* padding rule. Given a message M, the padding appends one bit 1, followed by the minimum number of 0 bits required so that the padded message length becomes a multiple of r. We denote the resulting padded message by $M' = \mathsf{pad}(M)$.

This padding rule is injective: distinct messages yield distinct padded encodings. In particular, the final padded block is never the all-zero block, which prevents trivial ambiguities in the absorbing input. Since the absorbing phase processes r-bit blocks, the padded message M' is parsed uniquely as $M' = M_0 \parallel$

$\cdots \parallel M_{\ell-1}$, where $M_i \in \{0,1\}^r$. The two-stage absorbing procedure of ASP is then applied to these blocks in order.

4 Security Analysis

In this section, we prove that the Alternating Sponge is indifferentiable from a random oracle $\mathcal{RO}$. The advantage bound consists of a bad-event term, a sampling-distance term, and primitive-space collision terms. In particular, with $\Gamma = 17q + 1$, the dominant terms are of order $\Gamma^3/2^{2c}$, $\Gamma^{3/2}/2^c$, and $\Gamma^2/2^{r+c}$, yielding beyond-birthday security with respect to the capacity c.

4.1 Indifferentiability Security Model

This subsection formally defines the indifferentiability security concept that we adopt, and clarifies the adversary model, laying the foundation for the subsequent security proof.

Definition 1 (Indifferentiability from a Random Oracle). *Let $\mathcal{P} = (f_1, f_2)$, where f_1 and f_2 are independent random permutations over $\{0,1\}^{r+c}$. Let $\mathcal{H}^{\mathcal{P}}$ denote the ASP construction instantiated with $\mathcal{P}$, and let $\mathcal{RO}$ be a random oracle returning an infinite sequence of r-bit output blocks for each message.*

In the real experiment, the distinguisher $\mathcal{D}$ has oracle access to

$$(\mathcal{H}^{\mathcal{P}}, f_1^{\pm}, f_2^{\pm}),$$

where $f_s^{\pm}$ denotes both the forward and inverse interfaces of f_s, for $s \in \{1,2\}$.

In the ideal experiment, the distinguisher $\mathcal{D}$ has oracle access to

$$(\mathcal{RO}, \mathcal{S}^{\mathcal{RO}}),$$

where $\mathcal{S}^{\mathcal{RO}}$ is a simulator that answers forward and inverse primitive queries to f_1 and f_2, while having oracle access to $\mathcal{RO}$.

For a fixed simulator $\mathcal{S}$, the indifferentiability advantage of $\mathcal{D}$ is defined as

$$\mathrm{Adv}^{\mathrm{indiff}}_{\mathcal{H},\mathcal{S}}(\mathcal{D}) := \left|\Pr\left[\mathcal{D}^{\mathcal{H}^{\mathcal{P}}, f_1^{\pm}, f_2^{\pm}} = 1\right] - \Pr\left[\mathcal{D}^{\mathcal{RO}, \mathcal{S}^{\mathcal{RO}}} = 1\right]\right|.$$

We say that $\mathcal{H}^{\mathcal{P}}$ is indifferentiable from $\mathcal{RO}$ up to a given query bound if there exists a simulator $\mathcal{S}$ such that, for every distinguisher $\mathcal{D}$ within that query bound, the above advantage is bounded by the claimed security expression.

Adversary Model The distinguisher $\mathcal{D}$ is adaptive and may issue hash queries as well as forward and inverse primitive queries to f_1 and f_2. We write q_P for the number of direct primitive queries made by $\mathcal{D}$.

Hash queries are evaluated on padded message blocks and may induce primitive calls when they are expanded as an ASP computation in the intermediate worlds. Let λ be an upper bound on the number of primitive calls induced by

all hash evaluations considered in the proof. We define the base primitive-call budget by

$$q := q_P + \lambda.$$

The additional table entries created by the closure procedure are not included in q; their number is controlled separately by the closure-complexity bad events **FanOut** and **ThreeRounds**.

For the probability analysis, we use

$$\Gamma := 17q + 1,$$

which bounds the size of the closed bookkeeping graph before the first bad event. Throughout the proof we assume

$$\Gamma < 2^c$$

so that every rate bucket contains at least one unused value whenever the simulator performs a constrained assignment. We also assume $\Gamma < 2^{r+c}$, which is implied by $\Gamma < 2^c$, since $r \geq 0$.

Theorem 1 (Indifferentiability of ASP). *Let $\mathcal{P} = (f_1, f_2)$, where f_1 and f_2 are independent random permutations over $\{0,1\}^{r+c}$, and let $\mathcal{H}^{\mathcal{P}}$ be the ASP construction instantiated with $\mathcal{P}$. Let $\mathcal{S}$ be the simulator defined in Section 4.2. Consider any adaptive distinguisher $\mathcal{D}$ that makes q_P direct forward or inverse primitive queries, and whose hash queries induce at most λ primitive calls when expanded as ASP evaluations. Let*

$$q := q_P + \lambda, \qquad \Gamma := 17q + 1,$$

and assume $\Gamma < 2^c$. Then

$$\mathrm{Adv}^{\mathrm{indiff}}_{\mathcal{H},\mathcal{S}}(\mathcal{D}) \leq \frac{192\Gamma^3}{(2^c - \Gamma)^2} + \frac{34\Gamma^2 + 4q\Gamma}{2^{r+c} - \Gamma} + \frac{\Gamma^{3/2}}{2^c - \Gamma} + \frac{\Gamma^2}{2^r(2^c - \Gamma)}.$$

The proof proceeds as follows. Section 4.2 gives an executable simulator that maintains partial permutation tables, rooted absorbing and squeezing states, activated contexts, rate constraints, and a lazy closure procedure. Section 4.3 defines the bad events under which these invariants may fail. Section 4.4 decomposes the indifferentiability advantage into four terms: the probability of bad events, the no-bad equivalence of W_I and W_{IM1}, the sampling distance between W_{IM1} and W_{IM2}, and the output-wise equivalence of W_{IM2} and W_R. Finally, Section 5 bounds the two nonzero terms, namely the bad-event probability and the rate-bucket sampling distance, which together yield the theorem.

4.2 Simulator Construction

In this section, we specify the ideal-world simulator. It answers forward and inverse queries to the ideal permutations f_1 and f_2, while preserving consistency

with the random oracle $\mathcal{RO}$ whenever the values fixed in the permutation tables determine an output block of the ASP construction.

We write every $(r+c)$-bit string as $R\|C$, with $R \in \{0,1\}^r$ the rate part and $C \in \{0,1\}^c$ the capacity part. For $Y \in \{0,1\}^{r+c}$, let $\mathsf{rate}(Y)$ and $\mathsf{cap}(Y)$ denote its rate and capacity components. We also write ASP states as (A, B, C), where A and B correspond to S_{up} and S_{mid}, respectively.

Tables and Rooted States The simulator maintains two partial permutation tables $T_1, T_2 : \{0,1\}^{r+c} \rightharpoonup \{0,1\}^{r+c}$, together with their inverse tables T_1^{-1}, T_2^{-1}. These tables are maintained so as to satisfy permutation consistency: for every $s \in \{1,2\}$, $T_s(X) = Y \iff T_s^{-1}(Y) = X$.

The simulator also maintains a set of absorbing rooted states, denoted by $\mathsf{AbsRoot}$.

An absorbing rooted state is a tuple $N = (A, B, C, \mu) \in \mathsf{AbsRoot}$, where $A, B \in \{0,1\}^r$, $C \in \{0,1\}^c$, and $\mu \in (\{0,1\}^r)^*$ is the absorbed block sequence. It represents that ASP reaches state (A, B, C) after absorbing μ. Initially, $(0^r, 0^r, 0^c, \epsilon) \in \mathsf{AbsRoot}$.

Let $N = (A, B, C, \mu) \in \mathsf{AbsRoot}$ and let $m \in \{0,1\}^r$. An absorbing transition from N on input block m is defined whenever both table entries

$$T_1(A \oplus m\|C) = A'\|C' \quad \textit{and} \quad T_2(B \oplus \sigma(m)\|C') = B'\|C''$$

are already defined. In this case, the corresponding successor rooted state is $N' = (A', B', C'', \mu\|m)$. This is precisely the absorbing-phase transition of the ASP construction.

In addition, the simulator maintains a set of squeezing rooted states, denoted by SqRoot.

A squeezing rooted state is a tuple $Q = (M, t, E, B, C) \in \mathsf{SqRoot}$, where $M \in \{0,1\}^*$, $t \in \mathbb{N}$, $E, B \in \{0,1\}^r$, and $C \in \{0,1\}^c$. It represents the beginning of squeezing round t for message M, with $E = \mathcal{RO}(M)[2t]$ and current state (E, B, C).

Whenever $N = (A, B, C, \mu) \in \mathsf{AbsRoot}$ satisfies $\mu = \mathsf{pad}(M)$, it induces $Q_0 = (M, 0, A \oplus B, B, C) \in \mathsf{SqRoot}$. The simulator then enforces $A \oplus B = \mathcal{RO}(M)[0]$, so the first output block agrees with the random oracle.

Activated Contexts Activated contexts connect the rooted-state bookkeeping with the rate-constraint mechanism. For a forward query (s, X), where $X = R\|C_x$, the simulator enumerates all ways in which X can match the next input to f_s in a partial ASP execution already represented by the rooted states. Each such match may later impose a rate constraint on the value assigned to $T_s(X)$.

The possible contexts are listed in Table 2. The conditions are read existentially: every satisfying choice of rooted state, and of m where applicable, creates one activated context. The two $\mathsf{Abs1}$ and $\mathsf{Sq1}$ contexts are considered only for queries to f_1, while $\mathsf{Abs2}$ and $\mathsf{Sq2}$ are considered only for queries to f_2.

Let $\mathsf{Ctx}(s, X)$ be the set of contexts of the appropriate oracle type in Table 2 triggered by (s, X). The query is *wild* if $\mathsf{Ctx}(s, X) = \emptyset$, and *coherent* otherwise.

Table 2. Activated contexts for a forward query on $X = R\|C_x$.

Type	Call	Trigger condition	Context
Abs1	Abs., f_1	$N = (A, B, C, \mu) \in \mathsf{AbsRoot}$, $C = C_x$, and $m = R \oplus A$.	$\mathsf{Abs1}(N, m)$
Abs2	Abs., f_2	$N = (A, B, C, \mu) \in \mathsf{AbsRoot}$, $m \in \{0,1\}^r$, $T_1(A \oplus m\|C) = A'\|C_x$ is defined, and $R = B \oplus \sigma(m)$.	$\mathsf{Abs2}(N, m, A', C_x)$
Sq1	Sq., f_1	$Q = (M, t, E, B, C) \in \mathsf{SqRoot}$, $R = E$, and $C_x = C$.	$\mathsf{Sq1}(Q)$
Sq2	Sq., f_2	$Q = (M, t, E, B, C) \in \mathsf{SqRoot}$, $T_1(E\|C) = U\|C_x$ is defined, and $R = U \oplus B$.	$\mathsf{Sq2}(Q, U, C_x)$

Coherent queries are precisely those for which the simulator may have to answer from a restricted rate bucket to remain consistent with the rooted ASP executions.

These are input-side definitions. Hence inverse queries need no separate context types: on $f_s^{-1}(Y)$, the simulator first determines the candidate preimage X, either from the existing table if $Y \in \mathrm{Img}(T_s)$, or by sampling a fresh unused X otherwise, and then computes $\mathsf{Ctx}(s, X)$ by the same rules.

Rate Constraints Activated contexts may force the rate part of the value assigned to a primitive input to take a prescribed value. For a primitive input (s, X) and a set $\mathcal{C} \subseteq \mathsf{Ctx}(s, X)$ of activated contexts, define

$$\mathsf{RateConstr}(s, X, \mathcal{C}) = \{\rho(\mathfrak{c}) : \mathfrak{c} \in \mathcal{C},\ \rho(\mathfrak{c}) \neq \bot\} \subseteq \{0,1\}^r,$$

where $\rho(\mathfrak{c})$ is the rate value required by context $\mathfrak{c}$, and $\rho(\mathfrak{c}) = \bot$ means that the context imposes no rate constraint. Equivalently, each $\rho \in \mathsf{RateConstr}(s, X, \mathcal{C})$ represents the requirement $\mathsf{rate}(T_s(X)) = \rho$.

The constraints are listed in Table 3. Contexts not covered by the table contribute $\bot$.

Table 3. Rate constraints imposed by activated contexts.

Context	When it constrains	Required rate value
$\mathsf{Abs2}(N, m, A', C_x)$, where $N = (A, B, C, \mu)$	$\mu\|m = \mathsf{pad}(M)$. The context completes absorption, and the first output block is $z_0 = A' \oplus B'$, where $T_2(X) = B'\|C''$.	$\rho = A' \oplus \mathcal{RO}(M)[0]$
$\mathsf{Sq1}(Q)$, where $Q = (M, t, E, B, C)$	The context is the first squeezing call. If $T_1(X) = U\|C'$, then $z_{2t+1} = U \oplus B$.	$\rho = B \oplus \mathcal{RO}(M)[2t+1]$
$\mathsf{Sq2}(Q, U, C_x)$, where $Q = (M, t, E, B, C)$	The context is the second squeezing call. If $T_2(X) = V\|C''$, then $z_{2t+2} = U \oplus V$.	$\rho = U \oplus \mathcal{RO}(M)[2t+2]$

If $|\mathsf{RateConstr}(s, X, \mathcal{C})| > 1$, then the activated contexts impose incompatible rate requirements on the same table entry, and the simulator declares a bad event. If $\mathsf{RateConstr}(s, X, \mathcal{C}) = \{\rho\}$ and $T_s(X)$ is already defined with

$\mathsf{rate}(T_s(X)) \neq \rho$, then the existing table value violates the required rate constraint, and the simulator also declares a bad event. These bad events are formalized in the next subsection.

Value Assignment We use two assignment procedures to write entries into the partial permutation tables. In both procedures, $\rho \in \{0,1\}^r \cup \{\perp\}$ denotes the required rate value, where $\rho = \perp$ means that no rate constraint is imposed.

Forward-side assignment. $\mathsf{AssignF}(s, X, \rho)$ is used when the domain point X is fixed and the simulator has to determine the image value. If $X \in \mathrm{Dom}(T_s)$, let $Y = T_s(X)$. If $\rho \neq \perp$ and $\mathsf{rate}(Y) \neq \rho$, the simulator declares a bad event; otherwise it returns Y. If $X \notin \mathrm{Dom}(T_s)$, the simulator samples

$$Y \xleftarrow{\$} \{Y' \in \{0,1\}^{r+c} \setminus \mathrm{Img}(T_s) : \rho = \perp \text{ or } \mathsf{rate}(Y') = \rho\}.$$

If the sampling set is empty, the simulator declares a bad event. Otherwise, it records $T_s(X) = Y$ and $T_s^{-1}(Y) = X$, and returns Y.

Inverse-side assignment. $\mathsf{AssignI}(s, X, Y, \rho)$ is used after an inverse query has prescribed the image value Y and the simulator has sampled a fresh unused domain point X. If $\rho \neq \perp$ and $\mathsf{rate}(Y) \neq \rho$, the simulator declares a bad event. Otherwise, it records $T_s(X) = Y$ and $T_s^{-1}(Y) = X$, and returns X.

Simulator Query Procedures We now describe how the simulator answers permutation queries.

Bad-event convention. The flag **BAD** is internal to the analysis and is never revealed to the distinguisher. Declaring a bad event does not make the simulator abort, nor does it cause the oracle to return a special symbol. Once **BAD** is set, the simulator answers the current and all subsequent primitive queries by ordinary lazy sampling of the corresponding permutation table, preserving only the consistency between T_s and T_s^{-1}. In this fallback mode, existing table entries are returned unchanged; fresh forward or inverse queries are sampled uniformly from the unused image or domain, respectively. The simulator then ignores activated contexts, rate constraints, and closure rules. Thus the oracle interface remains well-defined even after **BAD**. All transcript-equivalence claims below are made only for good executions before **BAD** is set, while the probability of entering **BAD** is bounded separately. In the sequel, whenever we say that the simulator declares a bad event, we mean that it sets the internal flag $\mathbf{BAD} \leftarrow 1$ and follows the fallback lazy-sampling rule described above.

For a primitive input (s, X), the simulator computes $\mathcal{C} = \mathsf{Ctx}(s, X)$ and $\mathcal{R} = \mathsf{RateConstr}(s, X, \mathcal{C})$. If $|\mathcal{R}| > 1$, the simulator declares a bad event. Otherwise, write $\rho(\mathcal{R}) = \perp$ when $\mathcal{R} = \emptyset$, and $\rho(\mathcal{R}) = \rho'$ when $\mathcal{R} = \{\rho'\}$. After every newly created table entry, the simulator invokes the closure procedure $\mathsf{Close}()$, defined in the next subsection.

Forward queries. On a query $f_s(X)$, if **BAD** has already been set, the simulator answers by the fallback lazy-sampling rule. Otherwise, it computes $\mathcal{C}$, $\mathcal{R}$, and $\rho = \rho(\mathcal{R})$ as above. It then calls $Y \leftarrow \mathsf{AssignF}(s, X, \rho)$. If this call creates a new table entry, the simulator invokes $\mathsf{Close}()$. Finally, it returns Y.

Inverse queries. On a query $f_s^{-1}(Y)$, if **BAD** has already been set, the simulator answers by the fallback lazy-sampling rule. Otherwise, if $Y \in \mathrm{Img}(T_s)$, the preimage has already been fixed and the simulator returns $T_s^{-1}(Y)$. Otherwise, it samples a fresh domain point $X \xleftarrow{\$} \{0,1\}^{r+c} \setminus \mathrm{Dom}(T_s)$, computes $\mathcal{C} = \mathsf{Ctx}(s, X)$, $\mathcal{R} = \mathsf{RateConstr}(s, X, \mathcal{C})$, and $\rho = \rho(\mathcal{R})$ as above. Since the image value Y is prescribed by the inverse query, the simulator cannot choose its rate part; it therefore calls $X \leftarrow \mathsf{AssignI}(s, X, Y, \rho)$. If this call creates a new table entry, the simulator invokes $\mathsf{Close}()$. Finally, it returns X.

Closure Procedure The closure procedure is a lazy fixed-point procedure. Its purpose is to ensure that whenever the current permutation tables already determine an ASP transition from a rooted state, the corresponding successor rooted state is also recorded. To avoid processing the same transition more than once, the simulator maintains two edge sets, $\mathsf{AbsEdge}$ and SqEdge, initially empty.

For a primitive input (s, X), let $\mathsf{CAssign}(s, X)$ denote the following constrained forward assignment. The simulator computes $\mathcal{C} = \mathsf{Ctx}(s, X)$ and $\mathcal{R} = \mathsf{RateConstr}(s, X, \mathcal{C})$. If $|\mathcal{R}| > 1$, it declares a bad event. Otherwise it sets $\rho = \bot$ if $\mathcal{R} = \emptyset$, and $\rho = \rho'$ if $\mathcal{R} = \{\rho'\}$, and returns $\mathsf{AssignF}(s, X, \rho)$.

The procedure $\mathsf{Close}()$ repeatedly applies the following two rules until no new table entry, rooted state, or edge is added. The closure is lazy: it only completes transitions already enabled by existing table entries. In particular, it does not proactively create the first f_1-entry of a newly created squeezing state, which prevents an unbounded squeezing chain.

Absorbing closure. For every $N = (A, B, C, \mu) \in \mathsf{AbsRoot}$ and every defined entry $T_1(R\|C) = A'\|C'$, let $m = R \oplus A$. If the transition from N on m has already been recorded in $\mathsf{AbsEdge}$, the simulator skips it. Otherwise, it sets $X_2 = (B \oplus \sigma(m))\|C'$ and computes $B'\|C'' \leftarrow \mathsf{CAssign}(2, X_2)$. The completed absorbing transition yields $N' = (A', B', C'', \mu\|m)$. Before inserting N', the simulator checks absorbing-root uniqueness: neither the absorbed history $\mu\|m$ nor the state (A', B', C'') may already be associated with a different counterpart. If the check fails, the simulator declares a bad event; otherwise it adds N' to $\mathsf{AbsRoot}$ and records $(N, m, N') \in \mathsf{AbsEdge}$.

If $\mu\|m = \mathsf{pad}(M)$, the completed absorbing path induces the initial squeezing root $Q_0 = (M, 0, A' \oplus B', B', C'')$. The simulator inserts Q_0 subject to the corresponding squeezing-root uniqueness conditions. The final absorbing rate constraint guarantees, unless a bad event occurs, that $A' \oplus B' = \mathcal{RO}(M)[0]$.

Squeezing closure. For every $Q = (M, t, E, B, C) \in \mathsf{SqRoot}$, let $X_1 = E\|C$. If the transition out of Q has already been recorded in SqEdge, or if $X_1 \notin \mathrm{Dom}(T_1)$, this

rule does nothing. Otherwise, the simulator computes $U\|C' \leftarrow \mathsf{CAssign}(1, X_1)$. The squeezing f_1 constraint guarantees, unless a bad event occurs, that $U \oplus B = \mathcal{RO}(M)[2t+1]$.

Next, the simulator sets $X_2 = (U \oplus B)\|C'$ and computes $V\|C'' \leftarrow \mathsf{CAssign}(2, X_2)$. The completed squeezing round yields $Q' = (M, t+1, U \oplus V, V, C'')$. Before inserting Q', the simulator checks squeezing-root uniqueness: neither the pair $(M, t+1)$ nor the internal state $(U \oplus V, V, C'')$ may already be associated with a different counterpart. If the check fails, the simulator declares a bad event; otherwise it adds Q' to SqRoot and records $(Q, Q') \in \mathsf{SqEdge}$. The squeezing f_2-constraint guarantees $U \oplus V = \mathcal{RO}(M)[2t+2]$, so the successor root is consistent with the next random-oracle output block.

Termination and chain reactions. Adding a new rooted state may make previously defined table entries newly interpretable as ASP transitions. Therefore, $\mathsf{Close}()$ is iterated to a fixed point. If the closure process triggered by a single external query creates too many table entries, too many rooted states, or too long a chain of dependent transitions, the simulator declares a bad event. The corresponding chain-reaction events and their bounds are specified in the next subsection.

Simulator Invariants We record the invariants maintained by the simulator. Conditioned on $\neg\mathbf{BAD}$, after every external query and the subsequent execution of $\mathsf{Close}()$, the following properties hold.

1. **Permutation consistency.** For each $s \in \{1, 2\}$, T_s is a partial permutation consistent with its inverse table: $T_s(X) = Y$ iff $T_s^{-1}(Y) = X$. Thus no two domain points map to the same image point, and no domain point receives two images.
2. **Absorbing-root correctness and uniqueness.** Each $N = (A, B, C, \mu) \in \mathsf{AbsRoot}$ represents the state obtained by running the ASP absorbing procedure on the block sequence μ using the current tables T_1, T_2. Moreover, absorbing roots are unique in both directions: the absorbed history μ and the state (A, B, C) determine each other.
3. **Squeezing-root correctness and uniqueness.** Each $Q = (M, t, E, B, C) \in \mathsf{SqRoot}$ satisfies $E = \mathcal{RO}(M)[2t]$, and (E, B, C) is the ASP squeezing state for message M at the beginning of round t. The pair (M, t) and the internal state (E, B, C) determine each other uniquely.
4. **Closure completeness.** After $\mathsf{Close}()$ reaches a fixed point, no enabled ASP transition remains partially completed. Namely, if $N = (A, B, C, \mu) \in \mathsf{AbsRoot}$ and $T_1(A \oplus m\|C) = A'\|C'$ is defined, then $T_2((B \oplus \sigma(m))\|C')$ is also defined and the absorbing edge is recorded in $\mathsf{AbsEdge}$. Similarly, if $Q = (M, t, E, B, C) \in \mathsf{SqRoot}$ and $T_1(E\|C) = U\|C'$ is defined, then $T_2((U \oplus B)\|C')$ is also defined and the squeezing edge is recorded in SqEdge.
5. **Random-oracle consistency.** Every ASP output block determined by the current transcript agrees with the corresponding block of $\mathcal{RO}$. In particular, an initial squeezing root $(M, 0, E, B, C) \in \mathsf{SqRoot}$ satisfies $E = \mathcal{RO}(M)[0]$;

and every recorded squeezing transition from $Q = (M, t, E, B, C)$ to $Q' = (M, t+1, E', B', C')$ produces the blocks $\mathcal{RO}(M)[2t+1]$ and $\mathcal{RO}(M)[2t+2]$, respectively.

The invariants are proved by induction on the number of external queries. The base case follows from initialization. For the induction step, activated contexts identify all affected ASP positions, rate constraints enforce the required random-oracle values, assignments preserve permutation consistency, and $\mathsf{Close}()$ completes all enabled transitions. Hence the invariants are preserved unless one of the bad events occurs.

4.3 Bad Events Analysis

We now define the bad events used in the indifferentiability proof. These events capture the situations in which the simulator may fail to preserve the invariants stated above: uniqueness of rooted states, consistency with $\mathcal{RO}$, partial-permutation consistency of the tables, and bounded closure complexity.

All events are defined with respect to the execution of the simulator described in Section 4.2. We write **BAD** for their union.

Structural Events

1. **BadNode.** This event occurs when the simulator attempts to insert a rooted state that violates the uniqueness of the rooted-state representation.
   For absorbing roots, **BadNode** is triggered if either the same absorbed history is already associated with a different state, or the same state is already associated with a different absorbed history. For squeezing roots, it is triggered if either the same pair (M, t) is already associated with a different squeezing state, or the same internal squeezing state is already associated with a different pair (M, t).
2. **MultiCol.** This event occurs when a single primitive input activates too many rooted ASP contexts, namely when $|\mathsf{Ctx}(s, X)| > 2$ for some $s \in \{1, 2\}$ and $X \in \{0,1\}^{r+c}$. In this case, the same table entry would have to be interpreted as part of more than two rooted computations. Excluding this event ensures that the simulator only needs to reconcile a bounded number of rate requirements for each table entry.

Consistency Events

1. **Inconsistency.** This event captures failures of the rate-constraint mechanism. It occurs whenever, for some primitive input (s, X), the simulator computes $\mathcal{C} \leftarrow \mathsf{Ctx}(s, X)$ and $\mathcal{R} \leftarrow \mathsf{RateConstr}(s, X, \mathcal{C})$, but the activated contexts impose incompatible rate requirements, i.e., $|\mathcal{R}| > 1$. It is also triggered if $\mathcal{R} = \{\rho\}$ while $T_s(X)$ is already defined and $\mathsf{rate}(T_s(X)) \neq \rho$. For inverse queries, **Inconsistency** is triggered if the prescribed image value Y fails to satisfy the unique required rate constraint.

2. **BadAssign.** This event occurs when the simulator cannot sample a fresh image value satisfying a required rate constraint. In particular, it occurs if $\mathsf{AssignF}(s, X, \rho)$ is called with $\rho \neq \perp$, but

$$\left\{Y \in \{0,1\}^{r+c} \setminus \mathrm{Img}(T_s) : \mathsf{rate}(Y) = \rho\right\} = \emptyset.$$

Closure-Complexity Events

1. **FanOut.** This event occurs when the closure process triggered by a single base primitive call branches too much in one closure round. More precisely, it is triggered if some closure round creates more than two new unrecorded enabled transitions or rooted states.
   This event rules out the case where one closure round enables too manyindependent ASP transitions at once. Conditioned on $\neg$**FanOut**, each closure round creates at most two new unrecorded enabled transitions.
2. **ThreeRounds.** This event occurs if the closure process triggered by a single external query requires more than two dependent closure rounds before reaching a fixed point. Equivalently, the query creates a rooted-transition dependency chain of length at least three, such as $N_0 \to N_1 \to N_2 \to N_3$, where each transition becomes enabled only after the previous one has been created.

Finally, we define

$$\begin{aligned}\mathbf{BAD} = \mathbf{BadNode} \vee \mathbf{MultiCol} \vee \mathbf{Inconsistency} \\ \vee \mathbf{BadAssign} \vee \mathbf{FanOut} \vee \mathbf{ThreeRounds}.\end{aligned}$$

Conditioned on $\neg$**BAD**, the simulator preserves the invariants of Section 4.2: the tables remain partial permutations, rooted states are unique, all enabled transitions are closed, and every ASP output block determined by the query history agrees with the corresponding block of $\mathcal{RO}$.

4.4 World Decomposition

We decompose the proof into four worlds: W_I, W_{IM1}, W_{IM2}, and W_R. The first hop replaces the external random oracle by the construction induced by the simulator. The second hop replaces the simulator's rate-constrained sampling by truly lazy-sampled random permutations. The last hop removes bookkeeping, that is invisible to the distinguisher.

Ideal world W_I. In W_I, the distinguisher has access to $\mathcal{RO}$ as the hash interface, and to the simulator $\mathcal{S}$ as the forward and inverse interfaces for f_1 and f_2. The simulator is defined in Section 4.2: it maintains the partial permutation tables, rooted states, closure edges, activated contexts, rate constraints, and closure procedure, and sets **BAD** whenever one of the bad events from Section 4.3 occurs.

Intermediate world W_{IM1}. World W_{IM1} uses the same simulator $\mathcal{S}$ and the same internal random oracle as W_I, but replaces the external hash interface by $\mathcal{H}^{\mathcal{S}}$, namely ASP evaluated through the simulator's primitive interfaces. Thus, $\mathcal{RO}$ is hidden from the distinguisher and is used only internally to impose rate constraints. A hash query may create table entries, rooted states, and closure edges, and may set **BAD**. Conditioned on $\neg$**BAD**, the simulator invariants imply that every output block of $\mathcal{H}^{\mathcal{S}}(M)$ equals the corresponding block of $\mathcal{RO}(M)$; hence W_I and W_{IM1} have identical hash interfaces on no-bad executions.

Intermediate world W_{IM2}. World W_{IM2} answers primitive queries using truly lazy-sampled random permutations f_1, f_2, and answers hash queries by evaluating ASP with these permutations. In parallel, it runs the same bookkeeping as the simulator, but only passively: it tracks rooted states, activated contexts, rate constraints, closure edges, and bad events, while never choosing or modifying primitive values. Therefore, W_{IM1} and W_{IM2} differ only in how fresh primitive values are sampled: W_{IM1} samples constrained values from prescribed rate buckets, whereas W_{IM2} samples true random-permutation values and merely checks whether the constraints fail.

Real world W_R. In W_R, the distinguisher interacts with the real ASP construction $\mathcal{H}^{\mathcal{P}}$, where $\mathcal{P} = (f_1, f_2)$ are two independent lazy-sampled random permutations, and also has forward and inverse access to these permutations. Since the bookkeeping in W_{IM2} is internal and never affects any oracle response, W_{IM2} and W_R are output-wise identical.

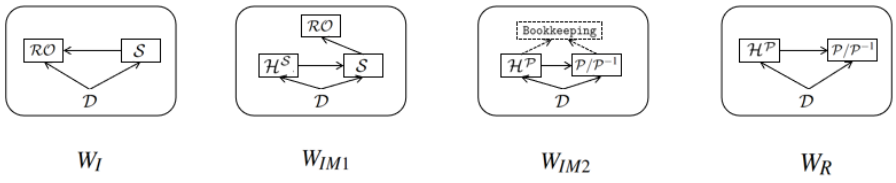


Fig. 2. World decomposition for the indifferentiability proof. Solid arrows denote the hybrid transitions bounded in the proof, and the dashed box in W_{IM2} denotes internal bookkeeping that observes the lazy-permutation transcript but does not affect oracle responses.

Let D^{W_x} denote the output of the distinguisher in world W_x, with $x \in \{I, I_{M1}, I_{M2}, R\}$. The event **BAD** is evaluated in the corresponding world. By the triangle inequality, we first separate the gap through W_{IM2}. We then split the W_I-to-W_{IM2} gap according to **BAD** and insert W_{IM1} on the no-bad executions. This step is purely algebraic; the full derivation is given in the auxiliary supporting material. We obtain

$$\mathrm{Adv}^{\mathrm{indiff}}_{\mathcal{H},\mathcal{S}}(\mathcal{D}) \le \max\{\Pr_{W_I}[\mathbf{BAD}], \Pr_{W_{IM2}}[\mathbf{BAD}]\} \tag{1a}$$

$$+ \left|\Pr[D^{W_I} = 1 \wedge \neg\mathbf{BAD}] - \Pr[D^{W_{IM1}} = 1 \wedge \neg\mathbf{BAD}]\right| \tag{1b}$$

$$+ \left|\Pr[D^{W_{IM1}} = 1 \wedge \neg\mathbf{BAD}] - \Pr[D^{W_{IM2}} = 1 \wedge \neg\mathbf{BAD}]\right| \tag{1c}$$

$$+ \left|\Pr[D^{W_{IM2}} = 1] - \Pr[D^{W_R} = 1]\right|. \tag{1d}$$

We now explain how the four terms in Equations (1a)–(1d) will be bounded. Let q_P be the number of direct forward and inverse primitive queries made by the distinguisher, and let λ be the total number of primitive calls induced by its hash queries when evaluating ASP. We set

$$q := q_P + \lambda.$$

Additional table entries created by closure are not included in q; they are controlled separately by **FanOut** and **ThreeRounds**. Conditioned on $\neg\mathbf{BAD}$, their total number is $O(q)$.

The remainder of the proof bounds the four terms in Equations (1a)-(1d) one by one. We only state the role of each term here; the detailed probability calculations are deferred to Section 5.

- **Term (1a)**: This term accounts for the probability of the **BAD** events. Its upper bound will be computed in Section 5.1.
- **Term (1b)**: This is the no-bad distance between W_I and W_{IM1}. We will show that this term is 0.
- **Term (1c)**: This is the no-bad distance between W_{IM1} and W_{IM2}. It is bounded by the sampling analysis in Section 5.2.
- **Term (1d)**: This is the distance between W_{IM2} and W_R. We show below that this term is 0.

Bad Events. As formalized in Lemma 2 (Section 5.1), the **BAD** events are proven to satisfy the upper bound

$$(1a) \le \frac{192\Gamma^3}{(2^c - \Gamma)^2} + \frac{34\Gamma^2 + 4q\Gamma}{2^{r+c} - \Gamma}$$

No-Bad Equivalence of Worlds W_I and W_{IM1}. The only difference between W_I and W_{IM1} is the hash interface. In W_I, hash queries are answered directly by $\mathcal{RO}$, whereas in W_{IM1} they are answered by the simulator-induced construction $\mathcal{H}^{\mathcal{S}}$. Conditioned on $\neg\mathbf{BAD}$, the simulator invariants guarantee that every output block of $\mathcal{H}^{\mathcal{S}}(M)$ equals the corresponding block of $\mathcal{RO}(M)$ for every hash query M. The primitive interface is the same simulator interface, and the additional table entries created during hash evaluation are consistent with the same closed state. Hence the two worlds induce identical transcripts under $\neg\mathbf{BAD}$, and

$$(1b) = 0.$$

Sampling Distance between W_{IM1} and W_{IM2}. The worlds W_{IM1} and W_{IM2} use the same bookkeeping rules, but their actual bookkeeping states need not coincide because the sampled primitive values may differ. The only relevant difference is how fresh constrained forward assignments are sampled. In W_{IM1}, the simulator samples them from the required rate buckets. In W_{IM2}, the corresponding values are sampled from truly lazy random permutations. This sampling discrepancy is bounded in Section 5.2.

By Lemma 4 (Section 5.2), the resulting good-transcript distance is bounded by

$$(1c) \le \frac{\Gamma^{3/2}}{2^c - \Gamma} + \frac{\Gamma^2}{2^r(2^c - \Gamma)}.$$

Equivalence of Worlds W_{IM2} and W_R. In W_{IM2}, all primitive queries are answered by truly lazy-sampled random permutations, and hash queries are answered by running the ASP construction with these permutations. The additional bookkeeping procedure only maintains internal rooted states, edge sets, context information, and bad-event flags. It never modifies the values returned by the permutation or hash interfaces.

Therefore, the transcript observed by the distinguisher in W_{IM2} is identical to the transcript observed in W_R. Consequently,

$$(1d) = 0.$$

Final Indifferentiability Bound. Combining the bounds for the four terms in (1a)–(1d), and using

$$(1b) = (1d) = 0,$$

we obtain

$$\begin{aligned}\mathbf{Adv}^{\mathrm{indif}}_{\mathcal{H}^P,\mathcal{RO}}(\mathcal{D}) &\le (1a) + (1c) \\ &\le \frac{192\Gamma^3}{(2^c - \Gamma)^2} + \frac{34\Gamma^2 + 4q\Gamma}{2^{r+c} - \Gamma} + \frac{\Gamma^{3/2}}{2^c - \Gamma} + \frac{\Gamma^2}{2^r(2^c - \Gamma)},\end{aligned}$$

where $\Gamma = 17q + 1$.

5 Probability Computation

In this section, we establish upper bounds for the two terms from equations (1a), (1c) in Section 4.4, and (1b), (1d) already proved in Section 4.4.

5.1 Bad Events Probability Analysis

We bound term (1a), namely the probability that one of the bad events occurs. Recall that

$$\begin{aligned}\mathbf{BAD} = \mathbf{BadNode} \vee \mathbf{MultiCol} \vee \mathbf{Inconsistency} \\ \vee \mathbf{BadAssign} \vee \mathbf{FanOut} \vee \mathbf{ThreeRounds}.\end{aligned}$$

The analysis first bounds the size of the internal graph generated by the simulator, and bounds the individual bad events by union bounds over this structure.

Throughout this subsection, we use the query budget q defined above. We call the q primitive calls counted in this budget the *base primitive calls*. These are the direct primitive queries made by the distinguisher together with the primitive calls induced by its hash queries when evaluating ASP. Additional table entries created by $\mathsf{Close}()$ are not included in q; they are controlled by the closure-complexity events.

Closed bookkeeping graph. After each base primitive call and the subsequent execution of $\mathsf{Close}()$, let

$$\mathcal{G}_{\mathrm{cl}} = (T_1, T_2, \mathsf{AbsRoot}, \mathsf{SqRoot}, \mathsf{AbsEdge}, \mathsf{SqEdge})$$

denote the closed bookkeeping graph maintained by the simulator. This graph is not an additional component of the simulator; it is a counting device for the probability analysis. It collects the permutation table entries, rooted states, and recorded closure edges generated so far.

The graph is called closed because, conditioned on $\neg\mathbf{BAD}$, the closure procedure has completed every enabled ASP transition: after $\mathsf{Close}()$ reaches a fixed point, no unrecorded enabled transition remains.

We measure its size by

$$|\mathcal{G}_{\mathrm{cl}}| := |\mathrm{Dom}(T_1)| + |\mathrm{Dom}(T_2)| + |\mathsf{AbsRoot}| + |\mathsf{SqRoot}| + |\mathsf{AbsEdge}| + |\mathsf{SqEdge}|.$$

Closure depth. By definition, **ThreeRounds** occurs if the closure process triggered by a single external query produces a dependency chain of rooted transitions of length at least three. Therefore, conditioned on $\neg\mathbf{ThreeRounds}$, the closure process triggered by one base primitive call has at most two dependent closure rounds.

For the size bound below, we instantiate the fan-out threshold as follows: **FanOut** occurs if one closure round creates more than two new unrecorded enabled transitions. Hence, conditioned on $\neg\mathbf{FanOut}$, each closure round creates at most two such transitions.

Lemma 1 (Size of the closed bookkeeping graph). *Let $\mathcal{G}_{\mathrm{cl}}^{(i)}$ denote the closed bookkeeping graph after the first i base primitive calls and the subsequent closure procedures. For every execution prefix in which no bad event has occurred, we have*

$$|\mathcal{G}_{\mathrm{cl}}^{(i)}| \le 17i + 1.$$

In particular, under $\neg\mathbf{BAD}$, after all q base primitive calls,

$$|\mathcal{G}_{\mathrm{cl}}| \le 17q + 1.$$

Proof. We use this bound only on good prefixes. Fix the execution prefix before the first bad event; on this prefix neither **FanOut** nor **ThreeRounds** has occurred, so their defining thresholds can be used to bound the closure generated by each base primitive call.

Initially, the graph contains only the initial absorbing root, so

$$|\mathcal{G}_{\mathrm{cl}}^{(0)}| = 1.$$

Consider one base primitive call before any bad event has occurred. The call creates at most one new table entry before closure is invoked.

Since **ThreeRounds** has not occurred, the closure process triggered by this call has at most two dependent closure rounds. Since **FanOut** has not occurred, each closure round creates at most two new unrecorded enabled transitions. Hence one base call triggers at most four completed closure transitions.

Each completed closure transition adds at most four objects to $\mathcal{G}_{\mathrm{cl}}$: one complementary table entry, one successor rooted state, one recorded edge, and, only in the final absorbing case, one initial squeezing root. Thus closure contributes at most $4 \cdot 4 = 16$ new objects per base call. Including the table entry created by the base call itself, each base call increases the graph size by at most 17. Therefore,

$$|\mathcal{G}_{\mathrm{cl}}^{(i)}| \le 17i + 1.$$

For later use, define $\Gamma := 17q + 1$. By Lemma 1, in any good prefix of the execution, the closed bookkeeping graph has size at most Γ.

Event-wise Probability Bounds In the event-wise bounds below, we use the first-bad-time convention: when bounding the probability that a bad event occurs first, all previous steps form a good prefix. Hence the bound $|\mathcal{G}_{\mathrm{cl}}| \le \Gamma$ applies immediately before the event.

Lemma 2 (Bad-event probability). *Let $\Gamma := 17q + 1$, and assume $\Gamma < 2^c$. Then term (1a) satisfies*

$$(1a) \le \frac{192\Gamma^3}{(2^c - \Gamma)^2} + \frac{34\Gamma^2 + 4q\Gamma}{2^{r+c} - \Gamma}.$$

Proof. We prove the bound in either W_I or W_{IM2}; the same argument applies to both. We use the first-bad-time convention: when bounding the probability that a bad event occurs first, all previous steps are good. Therefore, by Lemma 1, the closed bookkeeping graph immediately before the event has size at most Γ.

1. BadNode. The event **BadNode** has two possible directions. The direction in which the same history, or the same pair (M, t), is associated with two different states cannot be the first bad event, since the successor state is deterministic once the relevant table entries and closure edges have been fixed. It remains to bound the collision direction: two different histories, or two different pairs (M, t), reaching the same internal state.

We use a conservative bound and charge such a collision to a collision in the underlying primitive space. There are at most Γ^2 candidate pairs of absorbing roots and at most Γ^2 candidate pairs of squeezing roots. For each fixed pair, the

probability of hitting a previously represented primitive value is at most $\frac{1}{2^{r+c}-\Gamma}$. Therefore,

$$\Pr[\textbf{BadNode}] \leq \frac{2\Gamma^2}{2^{r+c}-\Gamma}.$$

2. MultiCol. For the context-based events, we use the following counting observation. Before the first bad event, the current closed bookkeeping graph exposes at most 4Γ context candidates: f_1-contexts are charged to rooted states, while f_2-contexts are charged to recorded closure edges.

The event **MultiCol** occurs if some primitive input activates more than two contexts. If **MultiCol** occurs, three context candidates are activated by the same primitive input. There are at most $(4\Gamma)^3 = 64\Gamma^3$ such triples. For a fixed triple, two capacity-side matches are required, each with probability at most $1/(2^c-\Gamma)$ before the first bad event. Hence

$$\Pr[\textbf{MultiCol}] \leq \frac{64\Gamma^3}{(2^c-\Gamma)^2}.$$

3. Inconsistency. The event **Inconsistency** captures failures of the rate-constraint mechanism. We consider three cases.

First, two activated contexts may impose incompatible rate requirements on the same table entry. There are at most 4Γ candidate constrained context occurrences, so the number of ordered pairs is at most $16\Gamma^2$. For a fixed pair, both contexts must be activated by the same primitive input, which requires a full input match. Since at least $2^{r+c}-\Gamma$ domain points remain unused, this contributes at most

$$\frac{16\Gamma^2}{2^{r+c}-\Gamma}.$$

Second, a table entry may already be defined, and a newly activated context may impose a rate constraint inconsistent with its existing value. The same counting gives another contribution of at most

$$\frac{16\Gamma^2}{2^{r+c}-\Gamma}.$$

Third, for inverse queries, Y may be adversarial, but when $Y \notin \mathrm{Img}(T_s)$, the sampled preimage X is uniform over at least $2^{r+c}-\Gamma$ unused domain points. Conditioned on the current good transcript, the set of constrained primitive inputs is already fixed and has size at most 4Γ. Thus the probability that the fresh X hits this set is at most $4\Gamma/(2^{r+c}-\Gamma)$ per inverse query, and over at most q such queries the contribution is at most

$$\frac{4q\Gamma}{2^{r+c}-\Gamma}.$$

Combining the three cases,

$$\Pr[\textbf{Inconsistency}] \leq \frac{32\Gamma^2 + 4q\Gamma}{2^{r+c}-\Gamma}.$$

4. BadAssign. The event **BadAssign** occurs when $\mathsf{AssignF}(s, X, \rho)$ is called with $\rho \neq \perp$, but there is no unused image value with rate part ρ. For any fixed rate value ρ, there are exactly 2^c image values. Before the first bad event, each table has used at most Γ image values. Since $\Gamma < 2^c$, this bucket cannot be exhausted. Therefore,

$$\Pr[\textbf{BadAssign}] = 0.$$

5. FanOut. The event **FanOut** is triggered when one closure round creates more than two new unrecorded enabled transitions. Thus, if some closure round creates at least three such transitions. By the counting observation above, there are at most $64\Gamma^3$ candidate triples. For a fixed triple, two capacity-side matches are needed; by sequential conditioning, this has probability at most $\frac{1}{(2^c-\Gamma)^2}$. Therefore,

$$\Pr[\textbf{FanOut}] \leq \frac{64\Gamma^3}{(2^c - \Gamma)^2}.$$

6. ThreeRounds. The event **ThreeRounds** occurs when the closure process triggered by a single external query produces a dependency chain of rooted transitions of length at least three:

$$N_0 \to N_1 \to N_2 \to N_3.$$

Such a chain requires two successive newly created rooted states to enable further transitions. Again there are at most $64\Gamma^3$ candidate transition triples, and a fixed triple requires two capacity-side matches. Hence

$$\Pr[\textbf{ThreeRounds}] \leq \frac{64\Gamma^3}{(2^c - \Gamma)^2}.$$

Finally, by the union bound,

$$\begin{aligned}(1a) &\leq \Pr[\textbf{BadNode}] + \Pr[\textbf{MultiCol}] + \Pr[\textbf{Inconsistency}] \\ &\quad + \Pr[\textbf{BadAssign}] + \Pr[\textbf{FanOut}] + \Pr[\textbf{ThreeRounds}] \\ &\leq \frac{192\Gamma^3}{(2^c - \Gamma)^2} + \frac{34\Gamma^2 + 4q\Gamma}{2^{r+c} - \Gamma}.\end{aligned}$$

5.2 Sampling Distance between W_{IM1} and W_{IM2}

This subsection bounds term (1c), the distance between the $\neg$**BAD** parts of W_{IM1} and W_{IM2}. The only remaining difference between these worlds is how fresh constrained primitive values are sampled: W_{IM1} samples them from the required rate buckets, whereas W_{IM2} obtains them from truly lazy random permutations.

Both worlds use the same bookkeeping rules, but their actual bookkeeping states need not coincide, since the sampled primitive values may differ. Unconstrained forward assignments are sampled uniformly from unused permutation values in both worlds and can be coupled perfectly. Fresh inverse assignments

can also be coupled perfectly, since both worlds sample a uniformly random unused preimage for the prescribed image and then apply the same consistency checks. Such assignments do not directly contribute to the sampling distance, but their prescribed images may affect later bucket occupancies. This effect is accounted for in the bucket-sampling bound below. Thus the direct sampling discrepancy comes only from fresh constrained forward assignments.

For a required rate value ρ, define the corresponding rate bucket by

$$\mathcal{B}_\rho = \{Y \in \{0,1\}^{r+c} : \mathsf{rate}(Y) = \rho\}.$$

Since the r-bit rate part is fixed and the c-bit capacity part is free,

$$|\mathcal{B}_\rho| = 2^c.$$

Whenever the rate label of a fresh constrained assignment is first defined, it is uniform in the joint transcript, since it is obtained by xoring a random-oracle block with values fixed by the current transcript. Later reuses of the same label are already determined by the transcript and are not counted as fresh constrained assignments. Thus the sampling problem reduces to comparing bucket sampling with true lazy permutation sampling, in the presence of possibly skewed bucket occupancies caused by previously coupled assignments.

Lemma 3 (Bucket-sampling bound). *Let a set of size 2^{r+c} be partitioned into $N = 2^r$ rate buckets, each of size $M = 2^c$. Suppose that the table receives m fresh constrained assignments and at most B other assignments whose images may be distributed arbitrarily among the buckets. Assume that at all times at most Γ elements have been used and $\Gamma < M$.*

In the bucket process, each fresh constrained assignment uses a uniformly random bucket label and then samples uniformly from the unused elements of that bucket. In the real process, each such assignment samples uniformly from all unused elements. Then the good-transcript distance between the two processes is at most

$$\frac{m^{3/2}}{M-\Gamma} + \frac{mB}{N(M-\Gamma)}.$$

Equivalently,

$$\frac{m^{3/2}}{2^c-\Gamma} + \frac{mB}{2^r(2^c-\Gamma)}.$$

Proof. The proof uses a standard sequential coupling over the fresh constrained assignments, in the spirit of the step-by-step indistinguishability analyses underlying the chi-squared method [6]. At each step, the distance from uniform lazy-permutation sampling is controlled by the current imbalance of the rate buckets. Cauchy's inequality bounds the imbalance from constrained samples, while the other B assignments are handled by a worst-case bucket-occupancy bound. The full calculation is deferred to the auxiliary supporting material.

Lemma 4 (Distance between W_{IM1} and W_{IM2}). *Let $\Gamma = 17q + 1$, and assume $\Gamma < 2^c$. The good-transcript distance in term (1c) satisfies*

$$(1c) \le \frac{\Gamma^{3/2}}{2^c - \Gamma} + \frac{\Gamma^2}{2^r(2^c - \Gamma)}.$$

Proof. For $s \in \{1, 2\}$, let m_s be the number of fresh constrained forward assignments to table T_s in a good transcript. Let B_s be the number of other assignments to T_s, including unconstrained forward assignments and fresh inverse assignments. Each such assignment creates a table entry in the closed bookkeeping graph. Hence

$$m_1 + m_2 \le \Gamma, \qquad B_1 + B_2 \le \Gamma.$$

We compare the two permutation tables separately and apply the triangle inequality. By Lemma 3, the contribution of table T_s is at most

$$\Delta_s \le \frac{m_s^{3/2}}{2^c - \Gamma} + \frac{m_s B_s}{2^r(2^c - \Gamma)}.$$

Therefore,

$$\begin{aligned}(1c) &\le \Delta_1 + \Delta_2 \\ &\le \frac{m_1^{3/2} + m_2^{3/2}}{2^c - \Gamma} + \frac{m_1 B_1 + m_2 B_2}{2^r(2^c - \Gamma)}.\end{aligned}$$

Since

$$m_1^{3/2} + m_2^{3/2} \le (m_1 + m_2)^{3/2} \le \Gamma^{3/2},$$

and

$$m_1 B_1 + m_2 B_2 \le (m_1 + m_2)(B_1 + B_2) \le \Gamma^2,$$

we obtain

$$(1c) \le \frac{\Gamma^{3/2}}{2^c - \Gamma} + \frac{\Gamma^2}{2^r(2^c - \Gamma)}.$$

6 Generic Distinguishing Attacks and Proof–Attack Gap

This section discusses generic distinguishing attacks for ASP and clarifies the gap between Theorem 1 and the attacks currently known to us. We present two attacks. The first is a construction-level state-merging birthday attack with complexity about $2^{(r+c)/2}$, matching the primitive-space term in our bound. The second is simulator-specific: it targets the present rate-constraint simulator rather than ASP itself. Its complexity is about $2^{(2c+r)/3}$, exposing a limitation of our current proof framework but not matching the capacity-driven $2c/3$ exponent.

6.1 A Construction-Level State-Merging Birthday Attack

We first give a construction-level attack that is independent of the simulator. The attack does not look for two already equal full states. Instead, it searches for two rooted ASP states that can be merged into the same next full state by choosing suitable next message blocks.

Let $N_i = (A_i, B_i, C_i)$ and $N_j = (A_j, B_j, C_j)$ be two rooted ASP states. The goal is to choose message blocks $m_i, m_j \in \{0,1\}^r$ such that the next absorbing step maps both states to the same full state.

For the first permutation call, it suffices to have $C_i = C_j$ and to choose $m_j = m_i \oplus A_i \oplus A_j$, which makes $A_i \oplus m_i = A_j \oplus m_j$. Hence the two f_1-inputs coincide.

Since the two f_1-outputs are then identical, the second permutation inputs have the same capacity part. The remaining condition is equality of their rate parts:

$$B_i \oplus \sigma(m_i) = B_j \oplus \sigma(m_j).$$

By linearity of σ, this is equivalent to $B_i \oplus B_j = \sigma(A_i \oplus A_j)$. Therefore, a useful pair of rooted states is characterized by

$$C_i = C_j, \qquad B_i \oplus B_j = \sigma(A_i \oplus A_j).$$

These impose $c + r$ bits of constraints. Hence, by the birthday paradox, after about $2^{(r+c)/2}$ rooted states, such a pair appears with constant probability.

Once the pair is found, appending m_i and m_j to the two corresponding prefixes makes the real ASP computations reach the same full state. All subsequent outputs coincide. For a random oracle, the two complete messages are distinct and their outputs are independent except with negligible probability. This yields a generic distinguishing attack of complexity about $2^{(r+c)/2}$.

Consequently, the $(r+c)/2$ term in Theorem 1 is unavoidable. In particular, when $r \le c/3$, this attack matches the dominant exponent of the proof.

6.2 A Simulator-Specific Attack

We next describe a simulator-specific attack against the present rate-constraint simulator.

Consider an absorbing rooted state $N = (A, B, C, \mu)$, and two already fixed table entries

$$T_1(X_1) = A' \parallel C', \qquad T_2(X_2) = B' \parallel C''.$$

Write $X_1 = R_1 \parallel C_1$ and $X_2 = R_2 \parallel C_2$. These two entries form a completed ASP absorbing transition from N if there exists a block m such that

$$X_1 = (A \oplus m) \parallel C, \qquad X_2 = (B \oplus \sigma(m)) \parallel C'.$$

Equivalently, the following three conditions must hold:

$$C_1 = C, \qquad C_2 = C', \qquad R_2 = B \oplus \sigma(R_1 \oplus A).$$

These impose $2c + r$ bits of constraints. Hence, among q rooted states, q entries in T_1, and q entries in T_2, such a configuration appears with probability about $\frac{q^3}{2^{2c+r}}$.

The relevant query order is important. The distinguisher first fixes the corresponding T_1- and T_2-entries while they are still unconstrained. Only later do these entries accidentally become part of a rooted absorbing transition that determines a random-oracle output block. If the corresponding context had been activated before the second table entry was sampled, then the simulator would have sampled that entry under the required rate constraint, and this distinguisher would not apply.

Suppose such a configuration is found and the corresponding history $\mu \parallel m$ determines an output block $\mathcal{RO}(M)[j]$. The already fixed table entries determine the ASP output block; denote it by $z^\star$. In the real world, the construction necessarily outputs $z^\star$. In the ideal world with the present simulator, however, $\mathcal{RO}(M)[j]$ is independent of these previously fixed entries. Thus

$$\Pr[\mathcal{RO}(M)[j] = z^\star] = 2^{-r}.$$

Conditioned on the above configuration, the attack distinguishes the two worlds with advantage roughly $1 - 2^{-r}$.

Overall, the attack succeeds with non-negligible advantage when $\frac{q^3}{2^{2c+r}} \approx 1$, when

$$q \approx 2^{(2c+r)/3}.$$

This does not contradict Theorem 1; it only shows that the current simulator-specific analysis leaves a proof–attack gap from the capacity-driven $2c/3$ exponent.

6.3 Discussion of Parameter Regimes

The attacks above clarify which part of the bound is matched by known generic attacks and where a gap remains. The proof gives the exponent

$$\kappa_{\mathsf{proof}} = \min\left\{\frac{2c}{3}, \frac{r+c}{2}\right\}.$$

The attacks discussed in this section give the attack exponent

$$\kappa_{\mathsf{atk}} = \min\left\{\frac{r+c}{2}, \frac{2c+r}{3}\right\},$$

where the first term comes from the construction-level state-merging birthday attack and the second term comes from the simulator-specific attack.

The natural threshold in the proof is $\frac{2c}{3} = \frac{r+c}{2}$, namely $r = c/3$. When $r \leq c/3$, the proof is dominated by the $(r + c)/2$ term. In this regime, the birthday attack of Section 6.1 matches the proven exponent, so the bound is tight with respect to this structural collision attack.

When $r > c/3$, the proof is dominated by the capacity-driven exponent $2c/3$. The attacks known to us do not match this exponent. More precisely, for $c/3 < r < c$, the birthday attack gives exponent $(r+c)/2$, leaving a gap $\frac{r+c}{2} - \frac{2c}{3} = \frac{3r-c}{6}$. For $r \geq c$, the simulator-specific distinguisher gives exponent $(2c+r)/3$, leaving a formal gap of $r/3$. At $r = c$, this complexity is already at the 2^c boundary, while the theorem assumes $\Gamma = 17q + 1 < 2^c$. For $r > c$, the complexity is beyond 2^c. Thus, in this range, the simulator-specific distinguisher should be viewed mainly as an indication of the current proof framework's limitation, not as a tightness statement within the theorem's admissible query range.

7 Conclusion

This paper introduced the *Alternating Sponge* (ASP), a permutation-based hashing mode that explores whether beyond-birthday-bound security can be obtained without duplicating the full capacity state of the Double Sponge. ASP replaces full spatial duplication by temporal alternation: it keeps two r-bit rate branches, but lets them share a single c-bit capacity chain. As a result, the state size is reduced from $2r+2c$ bits to $2r+c$ bits, while the mode still uses two independent $(r+c)$-bit permutations.

We proved that ASP is indifferentiable from a random oracle in the ideal permutation model. For the base primitive-call budget $q = q_P + \lambda$, the dominant terms in the bound are

$$O\left(\frac{q^3}{2^{2c}}\right), \qquad O\left(\frac{q^{3/2}}{2^c}\right), \qquad O\left(\frac{q^2}{2^{r+c}}\right).$$

Thus the proof gives the asymptotic security level

$$\min\left\{\frac{2c}{3}, \frac{r+c}{2}\right\}.$$

In particular, when $r \geq c/3$, ASP reaches the same capacity-driven $2c/3$ exponent as the Double Sponge while saving one c-bit capacity component; when $r < c/3$, the primitive-space term becomes the limiting factor.

We also discussed generic attacks and the current proof–attack gap. A construction-level state-merging birthday attack matches the $(r+c)/2$ term, showing that this part of the bound is unavoidable. In the capacity-dominated regime, however, the known attacks do not match the $2c/3$ proof exponent. Closing this gap, either by finding stronger generic attacks or by sharpening the simulator analysis, remains an important open problem.

Overall, ASP provides a new design point for beyond-birthday-bound permutation-based hashing. It shows that full capacity duplication is not the only route to stronger generic security: a shared capacity chain combined with temporal alternation can support an executable indifferentiability proof and yield a concrete tradeoff among state size, squeezing behavior, and provable security. Future work includes tightening the proof–attack gap, studying instantiations from concrete permutations, and understanding whether related alternating designs can achieve stronger bounds or lower implementation cost.

References

1. Andreeva, E., Mennink, B., Preneel, B.: On the indifferentiability of the grøstl hash function. In: Security and Cryptography for Networks – SCN 2010. Lecture Notes in Computer Science, vol. 6280, pp. 88–105. Springer (2010)
2. Andreeva, E., Mennink, B., Preneel, B.: The parazoa family: Generalizing the sponge hash functions. International Journal of Information Security **11**(3), 149–165 (2012). https://doi.org/10.1007/s10207-012-0157-6
3. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Sponge functions. In: Bertoni, G., Daemen, J., Peeters, M., Assche, G.V. (eds.) ECRYPT Hash Workshop 2007. pp. 1–20 (2007)
4. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: On the indifferentiability of the sponge construction. In: Advances in Cryptology – EUROCRYPT 2008. Lecture Notes in Computer Science, vol. 4965, pp. 181–197. Springer (2008)
5. Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: SPONGENT: A lightweight hash function. In: Cryptographic Hardware and Embedded Systems – CHES 2011. Lecture Notes in Computer Science, vol. 6917, pp. 312–325. Springer (2011). https://doi.org/10.1007/978-3-642-23951-9_21
6. Dai, W., Hoang, V.T., Tessaro, S.: Information-theoretic indistinguishability via the chi-squared method. In: Advances in Cryptology — CRYPTO 2017. Lecture Notes in Computer Science, vol. 10403, pp. 497–523. Springer (2017)
7. Dworkin, M.J.: Sha-3 standard: Permutation-based hash and extendable-output functions. Tech. Rep. 202, Federal Information Processing Standards (NIST FIPS) (2015), https://doi.org/10.6028/NIST.FIPS.202
8. Gauravaram, P., Knudsen, L.R., Leander, G., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (Round 3) (2011), official design document
9. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON family of lightweight hash functions. In: Advances in Cryptology – CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 222–239. Springer (2011). https://doi.org/10.1007/978-3-642-22792-9_13
10. Lefevre, C., Mennink, B.: Permutation-based hashing beyond the birthday bound. IACR Transactions on Symmetric Cryptology **2024 No. 1**, 71–113 (2024). https://doi.org/10.46586/tosc.v2024.i1.71-113, https://tosc.iacr.org/index.php/ToSC/article/view/11402
11. Li, H., Li, C., Hu, H., Liu, F.: Indifferentiability of the sponge hash family: New attacks and the optimal construction. IACR Transactions on Symmetric Cryptology **2026**(1), 31–75 (2026). https://doi.org/10.46586/tosc.v2026.i1.31-75
12. Maurer, U.M., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: Naor, M. (ed.) Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004. Lecture Notes in Computer Science, vol. 2951, pp. 21–39. Springer (2004). https://doi.org/10.1007/978-3-540-24638-1_2
13. Moody, D., Paul, S., Smith-Tone, D.: Improved indifferentiability security bound for the jh mode. Designs, Codes and Cryptography **79**(2), 237–259 (2016)
14. National Institute of Standards and Technology: Ascon-Based Lightweight Cryptography Standards for Constrained Devices: Authenticated Encryption, Hash, and Extendable Output Functions. NIST Special Publication 800-232, National Institute of Standards and Technology (2025). https://doi.org/10.6028/NIST.SP.800-232, https://doi.org/10.6028/NIST.SP.800-232

15. Wu, H.: The hash function JH. Submission to NIST SHA-3 competition (2011), round 3 submission

Supplementary Material

A Derivation of the World Decomposition

We justify the algebraic decomposition used in the main text. Let D^{W_x} denote the output of the distinguisher in world W_x, where $x \in \{I, IM1, IM2, R\}$. By definition,

$$\mathrm{Adv}^{\mathrm{indiff}}_{\mathcal{H},\mathcal{S}}(\mathcal{D}) = \left|\Pr[D^{W_I} = 1] - \Pr[D^{W_R} = 1]\right|. \tag{A.1}$$

By the triangle inequality, inserting W_{IM2} gives

$$\begin{aligned}\mathrm{Adv}^{\mathrm{indiff}}_{\mathcal{H},\mathcal{S}}(\mathcal{D}) \le{}& \left|\Pr[D^{W_I} = 1] - \Pr[D^{W_{IM2}} = 1]\right| && \text{(A.1a)}\\ &+ \left|\Pr[D^{W_{IM2}} = 1] - \Pr[D^{W_R} = 1]\right|. && \text{(A.1b)}\end{aligned}$$

For the distance in (A.1a), we split according to **BAD**, and insert W_{IM1} only in the no-bad part:

$$\begin{aligned}\text{(A.1a)} = {}& \Big| \Pr[D^{W_I} = 1 \wedge \mathbf{BAD}] + \Pr[D^{W_I} = 1 \wedge \neg\mathbf{BAD}]\\ &\quad - \Pr[D^{W_{IM1}} = 1 \wedge \neg\mathbf{BAD}] + \Pr[D^{W_{IM1}} = 1 \wedge \neg\mathbf{BAD}]\\ &\quad - \Pr[D^{W_{IM2}} = 1 \wedge \mathbf{BAD}] - \Pr[D^{W_{IM2}} = 1 \wedge \neg\mathbf{BAD}]\Big|\\ \le{}& \left|\Pr[D^{W_I} = 1 \wedge \mathbf{BAD}] - \Pr[D^{W_{IM2}} = 1 \wedge \mathbf{BAD}]\right| && \text{(A.2a)}\\ &+ \left|\Pr[D^{W_I} = 1 \wedge \neg\mathbf{BAD}] - \Pr[D^{W_{IM1}} = 1 \wedge \neg\mathbf{BAD}]\right| && \text{(A.2b)}\\ &+ \left|\Pr[D^{W_{IM1}} = 1 \wedge \neg\mathbf{BAD}] - \Pr[D^{W_{IM2}} = 1 \wedge \neg\mathbf{BAD}]\right|. && \text{(A.2c)}\end{aligned}$$

The bad part in (A.2a) is bounded by

$$\text{(A.2a)} \le \max\{\Pr_{W_I}[\mathbf{BAD}], \Pr_{W_{IM2}}[\mathbf{BAD}]\}. \tag{A.3}$$

Combining (A.1b), (A.2b), (A.2c), and (A.3) yields the four terms $(1a)$–$(1d)$ in the main text.

B Proof of the Bucket-Sampling Bound

Proof. Let $N = 2^r$ and $M = 2^c$. After i constrained samples, let x_ρ be the number of constrained samples that have fallen in bucket ρ, and let b_ρ be the number of other used images in bucket ρ. Write

$$x = \sum_\rho x_\rho = i, \qquad b = \sum_\rho b_\rho \le B, \qquad a_\rho = x_\rho + b_\rho.$$

Thus the total number of used elements is $a = i + b \le \Gamma$.

In the bucket process, an unused element $Y \in \mathcal{B}_\rho$ is sampled with probability

$$\frac{1}{N} \cdot \frac{1}{M - a_\rho},$$

whereas in the real process it is sampled with probability

$$\frac{1}{NM - a}.$$

Grouping unused elements by their buckets, the one-step statistical distance is

$$\begin{aligned}\delta_i &= \frac{1}{2} \sum_\rho (M - a_\rho) \left| \frac{1}{N} \cdot \frac{1}{M - a_\rho} - \frac{1}{NM - a} \right| \\ &= \frac{1}{2(NM - a)} \sum_\rho \left| a_\rho - \frac{a}{N} \right|.\end{aligned}$$

Using $a_\rho = x_\rho + b_\rho$, we have

$$\sum_\rho \left| a_\rho - \frac{a}{N} \right| \le \sum_\rho \left| x_\rho - \frac{i}{N} \right| + \sum_\rho \left| b_\rho - \frac{b}{N} \right|.$$

For the constrained samples, the bucket labels are uniform. Hence, under the bucket process,

$$\mathbb{E}\left[\sum_\rho \left(x_\rho - \frac{i}{N} \right)^2 \right] \le i.$$

By Cauchy's inequality,

$$\mathbb{E}\left[\sum_\rho \left| x_\rho - \frac{i}{N} \right| \right] \le \sqrt{Ni}.$$

For the other used images, we use the worst-case bound

$$\sum_\rho \left| b_\rho - \frac{b}{N} \right| \le 2b \le 2B.$$

Since $a \le \Gamma$, we have $NM - a \ge N(M - \Gamma)$. Therefore,

$$\mathbb{E}[\delta_i] \le \frac{\sqrt{Ni} + 2B}{2N(M - \Gamma)} \le \frac{\sqrt{i}}{2(M - \Gamma)} + \frac{B}{N(M - \Gamma)}.$$

Using the standard sequential hybrid bound for adaptive sampling,

$$\Delta \le \sum_{i=0}^{m-1} \mathbb{E}[\delta_i] \le \frac{1}{2(M - \Gamma)} \sum_{i=0}^{m-1} \sqrt{i} + \frac{mB}{N(M - \Gamma)}.$$

Since

$$\sum_{i=0}^{m-1} \sqrt{i} \leq 2m^{3/2},$$

we obtain

$$\Delta \leq \frac{m^{3/2}}{M - \Gamma} + \frac{mB}{N(M - \Gamma)}.$$

Substituting $N = 2^r$ and $M = 2^c$ gives the claim.